Contents

online-majority-element-in-subarray.py

```
# online-majority-element-in-subarra is not found.
# Time:  ctor:  O(n)
#        query: O(klogn), k = log2(Q/ERROR_RATE)
# Space: O(n)

import collections
import random
import bisect


class MajorityChecker(object):

    def __init__(self, arr):
        """
        :type arr: List[int]
        """
        Q, ERROR_RATE = 10000, 0.001
        self.__K = int(Q/ERROR_RATE).bit_length()  # floor(log2(Q/ERROR_RATE))+1 = 24
        self.__arr = arr
        self.__inv_idx = collections.defaultdict(list)
        for i, x in enumerate(self.__arr):
            self.__inv_idx[x].append(i)

    def query(self, left, right, threshold):
        """
        :type left: int
```

```
        :type right: int
        :type threshold: int
        :rtype: int
        """
        def count(inv_idx, m, left, right):
            return bisect.bisect_right(inv_idx[m], right) - \
                   bisect.bisect_left(inv_idx[m], left)

        for _ in xrange(self.__K):
            m = self.__arr[random.randint(left, right)]
            if count(self.__inv_idx, m, left, right) >= threshold:
                return m
        return -1


# Time:  ctor:  O(n)
#        query: O(sqrt(n)*logn)
# Space: O(n)
import collections
import bisect


class MajorityChecker2(object):

    def __init__(self, arr):
        """
        :type arr: List[int]
        """
        self.__arr = arr
        self.__inv_idx = collections.defaultdict(list)
        for i, x in enumerate(self.__arr):
            self.__inv_idx[x].append(i)
        self.__bound = int(round((len(arr)**0.5)))
        self.__majorities = [i for i, group in self.__inv_idx.iteritems() if len(group) >= self.__bound]

    def query(self, left, right, threshold):
        """
        :type left: int
        :type right: int
        :type threshold: int
        :rtype: int
        """
        def count(inv_idx, m, left, right):
            return bisect.bisect_right(inv_idx[m], right) - \
                   bisect.bisect_left(inv_idx[m], left)

        def boyer_moore_majority_vote(nums, left, right):
            m, cnt = nums[left], 1
            for i in xrange(left+1, right+1):
                if m == nums[i]:
                    cnt += 1
                else:
                    cnt -= 1
                    if cnt == 0:
                        m = nums[i]
                        cnt = 1
            return m

        if right-left+1 < self.__bound:
            m = boyer_moore_majority_vote(self.__arr, left, right)
```

```
            if count(self.__inv_idx, m, left, right) >= threshold:
                return m
        else:
            for m in self.__majorities:
                if count(self.__inv_idx, m, left, right) >= threshold:
                    return m
        return -1


# Time:  ctor:  O(nlogn)
#        query: O((logn)^2)
# Space: O(n)
import functools


class SegmentTreeRecu(object):  # 0-based index
    def __init__(self, nums, count):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        N = len(nums)
        self.__original_length = N
        self.__tree_length = 2**(N.bit_length() + (N&(N-1) != 0))-1
        self.__tree = [-1 for _ in range(self.__tree_length)]
        self.__count = count
        self.__constructTree(nums, 0, self.__original_length-1, 0)

    def query(self, i, j):
        return self.__queryRange(i, j, 0, self.__original_length-1, 0)

    def __constructTree(self, nums, left, right, idx):
        if left > right:
            return
        if left == right:
            self.__tree[idx] = nums[left]
            return
        mid = left + (right-left)//2
        self.__constructTree(nums, left, mid, idx*2 + 1)
        self.__constructTree(nums, mid+1, right, idx*2 + 2)
        if self.__tree[idx*2 + 1] != -1 and \
           self.__count(self.__tree[idx*2 + 1], left, right)*2 > right-left+1:
            self.__tree[idx] = self.__tree[idx*2 + 1]
        elif self.__tree[idx*2 + 2] != -1 and \
             self.__count(self.__tree[idx*2 + 2], left, right)*2 > right-left+1:
            self.__tree[idx] = self.__tree[idx*2 + 2]

    def __queryRange(self, range_left, range_right, left, right, idx):
        if left > right:
            return (-1, -1)
        if right < range_left or left > range_right:
            return (-1, -1)
        if range_left <= left and right <= range_right:
            if self.__tree[idx] != -1:
                c = self.__count(self.__tree[idx], range_left, range_right)
                if c*2 > range_right-range_left+1:
                    return (self.__tree[idx], c)
        else:
            mid = left + (right-left)//2
            result = self.__queryRange(range_left, range_right, left, mid, idx*2 + 1)
```

```
            if result[0] != -1:
                return result
            result = self.__queryRange(range_left, range_right, mid + 1, right, idx*2 + 2)
            if result[0] != -1:
                return result
        return (-1, -1)


class MajorityChecker3(object):

    def __init__(self, arr):
        """
        :type arr: List[int]
        """
        def count(inv_idx, m, left, right):
            return bisect.bisect_right(inv_idx[m], right) - \
                   bisect.bisect_left(inv_idx[m], left)

        self.__arr = arr
        self.__inv_idx = collections.defaultdict(list)
        for i, x in enumerate(self.__arr):
            self.__inv_idx[x].append(i)
        self.__segment_tree = SegmentTreeRecu(arr, functools.partial(count, self.__inv_idx))

    def query(self, left, right, threshold):
        """
        :type left: int
        :type right: int
        :type threshold: int
        :rtype: int
        """
        result = self.__segment_tree.query(left, right)
        if result[1] >= threshold:
            return result[0]
        return -1


# Time:  ctor:  O(n)
#        query: O(sqrt(n)*logn)
# Space: O(n)
import collections
import bisect


class MajorityChecker4(object):

    def __init__(self, arr):
        """
        :type arr: List[int]
        """
        self.__arr = arr
        self.__inv_idx = collections.defaultdict(list)
        for i, x in enumerate(self.__arr):
            self.__inv_idx[x].append(i)
        self.__bucket_size = int(round((len(arr)**0.5)))
        self.__bucket_majorities = []
        for left in xrange(0, len(self.__arr), self.__bucket_size):
            right = min(left+self.__bucket_size-1, len(self.__arr)-1)
            self.__bucket_majorities.append(self.__boyer_moore_majority_vote(self.__arr, left, right))
```

```
def query(self, left, right, threshold):
    """
    :type left: int
    :type right: int
    :type threshold: int
    :rtype: int
    """
    def count(inv_idx, m, left, right):
        return bisect.bisect_right(inv_idx[m], right) - \
               bisect.bisect_left(inv_idx[m], left)

    l, r = left//self.__bucket_size, right//self.__bucket_size;
    if l == r:
        m = self.__boyer_moore_majority_vote(self.__arr, left, right)
        if count(self.__inv_idx, m, left, right) >= threshold:
            return m
        return -1
    else:
        m = self.__boyer_moore_majority_vote(self.__arr, left, (l+1)*self.__bucket_size-1)
        if count(self.__inv_idx, m, left, right) >= threshold:
            return m
        m = self.__boyer_moore_majority_vote(self.__arr, r*self.__bucket_size, right)
        if count(self.__inv_idx, m, left, right) >= threshold:
            return m;
        for i in xrange(l+1, r):
            if count(self.__inv_idx, self.__bucket_majorities[i], left, right) >= threshold:
                return self.__bucket_majorities[i]
        return -1

def __boyer_moore_majority_vote(self, nums, left, right):
        m, cnt = nums[left], 1
        for i in xrange(left+1, right+1):
            if m == nums[i]:
                cnt += 1
            else:
                cnt -= 1
                if cnt == 0:
                    m = nums[i]
                    cnt = 1
        return m
```

increasing-order-search-tree.py

```
# DESC
# Constraints:
# Given a binary search tree, rearrange the tree in in-order so that the leftmost
# node in the tree is now the root of the tree, and every node has no left child a
# nd only 1 right child.

# NOTE
# The number of nodes in the given tree will be between 1 and 100.
# Each node will have a unique integer value from 0 to 1000.

# EXAMPLE
# Example 1:
# Input: [5,3,6,2,4,null,8,1,null,null,null,7,9]
#
#        5
#       / \
#
#   3    6
#    / \    \
#   2   4    8
#  /        / \
# 1        7   9
#
# Output: [1,null
# ,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]
#
#  1
#   \
#    2
#     \
#      3
#
#       \
#        4
#         \
#          5
#           \
#            6
#             \
#
#          7
#               \
#                8
#                 \
#                  9

# 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 increasingBST(self, root):
```

```
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        def increasingBSTHelper(root, tail):
            if not root:
                return tail
            result = increasingBSTHelper(root.left, root)
            root.left = None
            root.right = increasingBSTHelper(root.right, tail)
            return result
        return increasingBSTHelper(root, None)
```

add-bold-tag-in-string.py

```
# add-bold-tag-in-string is not found.
# Time:  O(n * d * l), l is the average string length
# Space: O(n)

import collections
import functools


# 59ms
class Solution(object):
    def addBoldTag(self, s, dict):
        """
        :type s: str
        :type dict: List[str]
        :rtype: str
        """
        lookup = [0] * len(s)
        for d in dict:
            pos = s.find(d)
            while pos != -1:
                lookup[pos:pos+len(d)] = [1] * len(d)
                pos = s.find(d, pos + 1)

        result = []
        for i in xrange(len(s)):
            if lookup[i] and (i == 0 or not lookup[i-1]):
                result.append("<b>")
            result.append(s[i])
            if lookup[i] and (i == len(s)-1 or not lookup[i+1]):
                result.append("</b>")
        return "".join(result)


# Time:  O(n * l), l is the average string length
# Space: O(t)    , t is the size of trie
# trie solution, 439ms
class Solution2(object):
    def addBoldTag(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: str
        """
        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        for i, word in enumerate(words):
            functools.reduce(dict.__getitem__, word, trie).setdefault("_end")

        lookup = [False] * len(s)
        for i in xrange(len(s)):
            curr = trie
            k = -1
            for j in xrange(i, len(s)):
                if s[j] not in curr:
                    break
                curr = curr[s[j]]
                if "_end" in curr:
                    k = j
```

```
    for j in xrange(i, k+1):
        lookup[j] = True

result = []
for i in xrange(len(s)):
    if lookup[i] and (i == 0 or not lookup[i-1]):
        result.append("<b>")
    result.append(s[i])
    if lookup[i] and (i == len(s)-1 or not lookup[i+1]):
        result.append("</b>")
return "".join(result)
```

reachable-nodes-in-subdivided-graph.py

```
# DESC
# and n is the total number of new nodes on that edge.
# Now, you start at node 0 from the original graph, and in each move, you travel a
# long one edge.
# edges[k]
# Example 1:
# and n+1 new edges (i, x_1), (x_1, x_2), (x_2, x_3), ..., (x_{n-1}, x_n), (x_n, j
# ) are added to the original graph.
# Return how many nodes you can reach in at most M moves.
# Starting with an undirected graph (the "original graph") with nodes from 0 to N-
# 1, subdivisions are made to some of the edges.
# Note:
# The graph is given as follows: edges[k] is a list of integer pairs (i, j, n) suc
# h that (i, j) is an edge of the original graph,
# Then, the edge (i, j) is deleted from the original graph, n new nodes (x_1, x_2,
#  ..., x_n) are added to the original graph,
# Example 2:

# NOTE
# 0 <= M <= 10^9
# 0 <= edges.length <= 10000
# A reachable node is a node that can be travelled to using at most M moves starti
# ng from node 0.
# 0 <= edges[i][0] < edges[i][1] < N
# There does not exist any i != j for which edges[i][0] == edges[j][0] and edges[i
# ][1] == edges[j][1].
# 0 <= edges[i][2] <= 10000
# The original graph has no parallel edges.
# 1 <= N <= 3000

# EXAMPLE
# Input: edges = [[0,1,10],[0,2,1],[1,2,2]], M = 6, N = 3
# Output: 13
# Explanation:
#
# The nodes that are reachable in the final graph after M = 6 moves are indicated
#  below.
# Input: edges = [[0,1,4],[1,2,6],[0,2,8],[1,3,1]], M = 10, N = 4
# Output: 23

# Time:  O((|E| + |V|) * log|V|) = O(|E| * log|V|),
#        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 heapq

class Solution(object):
    def reachableNodes(self, edges, M, N):
        """
        :type edges: List[List[int]]
        :type M: int
        :type N: int
        :rtype: int
        """
        adj = [[] for _ in xrange(N)]
        for u, v, w in edges:
            adj[u].append((v, w))
```

```
    adj[v].append((u, w))

min_heap = [(0, 0)]
best = collections.defaultdict(lambda: float("inf"))
best[0] = 0
count = collections.defaultdict(lambda: collections.defaultdict(int))
result = 0
while min_heap:
    curr_total, u = heapq.heappop(min_heap)  # O(|V|*log|V|) in total
    if best[u] < curr_total:
        continue
    result += 1
    for v, w in adj[u]:
        count[u][v] = min(w, M-curr_total)
        next_total = curr_total+w+1
        if next_total <= M and next_total < best[v]:
            best[v] = next_total
            heapq.heappush(min_heap, (next_total, v))  # binary heap O(|E|*log|V|) in total
                                                       # Fibonacci heap O(|E|) in total
for u, v, w in edges:
    result += min(w, count[u][v]+count[v][u])
return result
```

positions-of-large-groups.py

```
# DESC
# Note:  1 <= S.length <= 1000
# Example 3:
# The final answer should be in lexicographic order.
# Call a group large if it has 3 or more characters.  We would like the starting a
# nd ending positions of every large group.
# For example, a string like S = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z
# " and "yy".
# S = "abbxxxxzyy"
# Example 1:
# In a string S of lowercase letters, these letters form consecutive groups of the
#  same character.
# Example 2:

# NOTE
#

# EXAMPLE
# Input: "abc"
# Output: []
# Explanation: We have "a","b" and "c" but no large group.
# Input: "abcdddeeeeaabbbcd"
# Output: [[3,5],[6,9],[12,14]]
# Input: "abbxxxxzzy"
# Output: [[3,6]]
# Explanation: "xxxx" is the single large grou
# p with starting  3 and ending positions 6.

# Time:  O(n)
# Space: O(1)


class Solution(object):
    def largeGroupPositions(self, S):
        """
        :type S: str
        :rtype: List[List[int]]
        """
        result = []
        i = 0
        for j in xrange(len(S)):
            if j == len(S)-1 or S[j] != S[j+1]:
                if j-i+1 >= 3:
                    result.append([i, j])
                i = j+1
        return result
```

linked-list-in-binary-tree.py

```
# linked-list-in-binary-tree is not found.
# Time:  O(n + l)
# Space: O(h + l)

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


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


# kmp solution
class Solution(object):
    def isSubPath(self, head, root):
        """
        :type head: ListNode
        :type root: TreeNode
        :rtype: bool
        """
        def getPrefix(head):
            pattern, prefix = [head.val], [-1]
            j = -1
            node = head.next
            while node:
                while j+1 and pattern[j+1] != node.val:
                    j = prefix[j]
                if pattern[j+1] == node.val:
                    j += 1
                pattern.append(node.val)
                prefix.append(j)
                node = node.next
            return pattern, prefix

        def dfs(pattern, prefix, root, j):
            if not root:
                return False
            while j+1 and pattern[j+1] != root.val:
                j = prefix[j]
            if pattern[j+1] == root.val:
                j += 1
            if j+1 == len(pattern):
                return True
            return dfs(pattern, prefix, root.left, j) or \
                   dfs(pattern, prefix, root.right, j)

        if not head:
            return True
        pattern, prefix = getPrefix(head)
        return dfs(pattern, prefix, root, -1)
```

```python
# Time:  O(n * min(h, l))
# Space: O(h)
# dfs solution
class Solution2(object):
    def isSubPath(self, head, root):
        """
        :type head: ListNode
        :type root: TreeNode
        :rtype: bool
        """
        def dfs(head, root):
            if not head:
                return True
            if not root:
                return False
            return root.val == head.val and \
                   (dfs(head.next, root.left) or
                    dfs(head.next, root.right))

        if not head:
            return True
        if not root:
            return False
        return dfs(head, root) or \
               self.isSubPath(head, root.left) or \
               self.isSubPath(head, root.right)
```

minimum-difficulty-of-a-job-schedule.py

```python
# minimum-difficulty-of-a-job-schedule is not found.
# Time:  O(d * n^2)
# Space: O(d * n)

class Solution(object):
    def minDifficulty(self, jobDifficulty, d):
        """
        :type jobDifficulty: List[int]
        :type d: int
        :rtype: int
        """
        if len(jobDifficulty) < d:
            return -1;

        dp = [[float("inf")]*len(jobDifficulty) for _ in xrange(d)]
        dp[0][0] = jobDifficulty[0]
        for i in xrange(1, len(jobDifficulty)):
            dp[0][i] = max(dp[0][i-1], jobDifficulty[i])
        for i in xrange(1, d):
            for j in xrange(i, len(jobDifficulty)):
                curr_max = jobDifficulty[j]
                for k in reversed(xrange(i, j+1)):
                    curr_max = max(curr_max, jobDifficulty[k])
                    dp[i][j] = min(dp[i][j], dp[i-1][k-1] + curr_max)
        return dp[d-1][len(jobDifficulty)-1]
```

minimum-number-of-flips-to-convert-binary-matrix-to-zero-matrix.py

```python
# minimum-number-of-flips-to-convert-binary-matrix-to-zero-matrix is not found.
# Time:  O((m * n) * 2^(m * n))
# Space: O((m * n) * 2^(m * n))

import collections


class Solution(object):
    def minFlips(self, mat):
        """
        :type mat: List[List[int]]
        :rtype: int
        """
        directions = [(0, 0), (0, 1), (1, 0), (0, -1), (-1, 0)]
        start = sum(val << r*len(mat[0])+c for r, row in enumerate(mat) for c, val in enumerate(row))
        q = collections.deque([(start, 0)])
        lookup = {start}
        while q:
            state, step = q.popleft()
            if not state:
                return step
            for r in xrange(len(mat)):
                for c in xrange(len(mat[0])):
                    new_state = state
                    for dr, dc in directions:
                        nr, nc = r+dr, c+dc
                        if 0 <= nr < len(mat) and 0 <= nc < len(mat[0]):
                            new_state ^= 1 << nr*len(mat[0])+nc
                    if new_state in lookup:
                        continue
                    lookup.add(new_state)
                    q.append((new_state, step+1))
        return -1
```

diet-plan-performance.py

```python
# diet-plan-performance is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def dietPlanPerformance(self, calories, k, lower, upper):
        """
        :type calories: List[int]
        :type k: int
        :type lower: int
        :type upper: int
        :rtype: int
        """
        total = sum(itertools.islice(calories, 0, k))
        result = int(total > upper)-int(total < lower)
        for i in xrange(k, len(calories)):
            total += calories[i]-calories[i-k]
            result += int(total > upper)-int(total < lower)
        return result
```

uncommon-words-from-two-sentences.py

```
# DESC
# Note:
# We are given two sentences A and B.  (A sentence is a string of space separated
# words.  Each word consists only of lowercase letters.)
# Example 2:
# Return a list of all uncommon words.
# Example 1:
# A word is uncommon if it appears exactly once in one of the sentences, and does
# not appear in the other sentence.
# You may return the list in any order.

# NOTE
# 0 <= B.length <= 200
# 0 <= A.length <= 200
# A and B both contain only spaces and lowercase letters.

# EXAMPLE
# Input: A = "apple apple", B = "baa"
# Output: ["baa"]
# Input: A = "this apple is sweet", B = "this apple is sour"
# Output: ["sweet","sour"]

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

import collections


class Solution(object):
    def uncommonFromSentences(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: List[str]
        """
        count = collections.Counter(A.split())
        count += collections.Counter(B.split())
        return [word for word in count if count[word] == 1]
```

longest-arithmetic-sequence.py

```
# DESC
# Example 2:
# Recall that a subsequence of A is a list A[i_1], A[i_2], ..., A[i_k] with 0 <= i
# _1 < i_2 < ... < i_k <= A.length - 1, and that a sequence B is arithmetic if B[i
# +1] - B[i] are all the same value (for 0 <= i < B.length - 1).
# Given an array A of integers, return the length of the longest arithmetic subseq
# uence in A.
# Example 1:
# Example 3:
# Note:

# NOTE
# 2 <= A.length <= 2000
# 0 <= A[i] <= 10000

# EXAMPLE
# Input: [20,1,15,3,10,5,8]
# Output: 4
# Explanation:
# The longest arithmetic subsequ
# ence is [20,15,10,5].
# Input: [3,6,9,12]
# Output: 4
# Explanation:
# The whole array is an arithmetic seque
# nce with steps of length = 3.
# Input: [9,4,7,2,10]
# Output: 3
# Explanation:
# The longest arithmetic subsequence i
# s [4,7,10].

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

import collections

class Solution(object):
    def longestArithSeqLength(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        dp = collections.defaultdict(int)
        for i in xrange(len(A)-1):
            for j in xrange(i+1, len(A)):
                v =  A[j]-A[i]
                dp[v, j] = max(dp[v, j], dp[v, i]+1)
        return max(dp.values())+1
```

maximum-points-you-can-obtain-from-cards.py

```python
# maximum-points-you-can-obtain-from-cards is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxScore(self, cardPoints, k):
        """
        :type cardPoints: List[int]
        :type k: int
        :rtype: int
        """
        result, total, curr, left = float("inf"), 0, 0, 0
        for right, point in enumerate(cardPoints):
            total += point
            curr += point
            if right-left+1 > len(cardPoints)-k:
                curr -= cardPoints[left]
                left += 1
            if right-left+1 == len(cardPoints)-k:
                result = min(result, curr)
        return total-result
```

top-k-frequent-elements.py

```
# DESC
# Note:
# Example 1:
# Example 2:
# Given a non-empty array of integers, return the k most frequent elements.

# NOTE
# You may assume k is always valid, 1  k  number of unique elements.
# You can return the answer in any order.
# Your algorithm's time complexity must be better than O(n log n), where n is the
# array's size.
# It's guaranteed that the answer is unique, in other words the set of the top k f
# requent elements is unique.

# EXAMPLE
# Input: nums = [1,1,1,2,2,3], k = 2
# Output: [1,2]
# Input: nums = [1], k = 1
# Output: [1]

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

import collections


class Solution(object):
    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:
                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

```
# DESC
# Example 1:
# Given a non-empty string containing an out-of-order English representation of di
# gits 0-9, output the digits in ascending order.
# Example 2:
# Note:

# NOTE
# Input is guaranteed to be valid and can be transformed to its original digits. T
# hat means invalid inputs such as "abc" or "zerone" are not permitted.
# Input contains only lowercase English letters.
# Input length is less than 50,000.

# EXAMPLE
# Input: "fviefuro"
#
# Output: "45"
# Input: "owoztneoer"
#
# Output: "012"

# 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))
```

hexspeak.py

```python
# hexspeak is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def toHexspeak(self, num):
        """
        :type num: str
        :rtype: str
        """
        lookup = {0:'O', 1:'I'}
        for i in xrange(6):
            lookup[10+i] = chr(ord('A')+i)
        result = []
        n = int(num)
        while n:
            n, r = divmod(n, 16)
            if r not in lookup:
                return "ERROR"
            result.append(lookup[r])
        return "".join(reversed(result))


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def toHexspeak(self, num):
        """
        :type num: str
        :rtype: str
        """
        result = hex(int(num)).upper()[2:].replace('0', 'O').replace('1', 'I')
        return result if all(c in "ABCDEFOI" for c in result) else "ERROR"
```

pyramid-transition-matrix.py

```
# DESC
# We start with a bottom row of bottom, represented as a single string. We also st
# art with a list of allowed triples allowed. Each allowed triple is represented a
# s a string of length 3.
# Return true if we can build the pyramid all the way to the top, otherwise false.
# We are allowed to place any color block C on top of two adjacent blocks of color
# s A and B, if and only if ABC is an allowed triple.
# Example 1:
# Example 2:
# Constraints:
# We are stacking blocks to form a pyramid. Each block has a color which is a one
# letter string.

# NOTE
# Letters in all strings will be chosen from the set {'A', 'B', 'C', 'D', 'E', 'F'
# , 'G'}.
# bottom will be a string with length in range [2, 8].
# allowed will have length in range [0, 200].

# EXAMPLE
# Input: bottom = "AABA", allowed = ["AAA", "AAB", "ABA", "ABB", "BAC"]
# Output: fa
# lse
# Explanation:
# We can't stack the pyramid to the top.
# Note that there could be
#  allowed triples (A, B, C) and (A, B, D) with C != D.
# Input: bottom = "BCD", allowed = ["BCG", "CDE", "GEA", "FFF"]
# Output: true
# Expla
# nation:
# We can stack the pyramid like this:
#     A
#    / \
#   G   E
#  / \ / \
# B   C
#   D
#
# We are allowed to place G on top of B and C because BCG is an allowed tripl
# e.  Similarly, we can place E on top of C and D, then A on top of G and E.

# Time:  O((a^(b+1)-a)/(a-1)) = O(a^b) , a is the size of allowed,
#                                        b is the length of bottom
# Space: O((a^(b+1)-a)/(a-1)) = O(a^b)

class Solution(object):
    def pyramidTransition(self, bottom, allowed):
        """
        :type bottom: str
        :type allowed: List[str]
        :rtype: bool
        """
        def pyramidTransitionHelper(bottom, edges, lookup):
            def dfs(bottom, edges, new_bottom, idx, lookup):
                if idx == len(bottom)-1:
                    return pyramidTransitionHelper("".join(new_bottom), edges, lookup)
                for i in edges[ord(bottom[idx])-ord('A')][ord(bottom[idx+1])-ord('A')]:
```

```
            new_bottom[idx] = chr(i+ord('A'))
            if dfs(bottom, edges, new_bottom, idx+1, lookup):
                return True
        return False

    if len(bottom) == 1:
        return True
    if bottom in lookup:
        return False
    lookup.add(bottom)
    for i in xrange(len(bottom)-1):
        if not edges[ord(bottom[i])-ord('A')][ord(bottom[i+1])-ord('A')]:
            return False
    new_bottom = ['A']*(len(bottom)-1)
    return dfs(bottom, edges, new_bottom, 0, lookup)

edges = [[[] for _ in xrange(7)] for _ in xrange(7)]
for s in allowed:
    edges[ord(s[0])-ord('A')][ord(s[1])-ord('A')].append(ord(s[2])-ord('A'))
return pyramidTransitionHelper(bottom, edges, set())
```

compare-version-numbers.py

```
# DESC
# Example 4:
# Example 5:
# 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 numb
# er are both 0.
# Compare two version numbers version1 and version2.
#
# If version1 > version2 retur
# n 1; if version1 < version2 return -1;otherwise return 0.
# You may assume that the version strings are non-empty and contain only digits an
# d the . character.
# The . character does not represent a decimal point and is used to separate numbe
# r sequences.
# Example 3:
# Example 2:
# Example 1:
# 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.
# Note:

# NOTE
# Version strings are composed of numeric strings separated by dots . and this num
# eric strings may have leading zeroes.
# Version strings do not start or end with dots, and they will not be two consecut
# ive dots.

# EXAMPLE
# Input: version1 = "1.0.1", version2 = "1"
# Output: 1
# Input: version1 = "1.0", version2 = "1.0.0"
# Output: 0
# Explanation: The first ver
# sion number does not have a third level revision number, which means its third l
# evel revision number is default to "0"
# Input: version1 = "7.5.2.4", version2 = "7.5.3"
# Output: -1
# Input: version1 = "1.01", version2 = "1.001"
# Output: 0
# Explanation: Ignoring lea
# ding zeroes, both "01" and "001" represent the same number "1"
# Input: version1 = "0.1", version2 = "1.1"
# Output: -1

# Time:  O(n)
# Space: O(1)

import itertools


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

```
# DESC
# Follow up:
#
# What if negative numbers are allowed in the given array?
#
# How does i
# t change the problem?
#
# What limitation we need to add to the question to allow n
# egative numbers?
# Credits:
#
# Special thanks to @pbrother for adding this problem and creating all t
# est cases.
# Example:
# Given an integer array with all positive numbers and no duplicates, find the num
# ber of possible combinations that add up to a positive integer target.

# NOTE
#

# 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 a
# re counted as different combinations.
#
# Therefore the output is 7.

# 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
# DESC
# Example 3:
# Constraints:
# Example 2:
# Example 1:
# Koko can decide her baas-per-hour eating speed of K.  Each hour, she chooses som
# e pile of baas, and eats K baas from that pile.  If the pile has less than K baa
# s, she eats all of them instead, and won't eat any more baas during this hour.
# Koko loves to eat baas.  There are N piles of baas, the i-th pile has piles[i] b
# aas.  The guards have gone and will come back in H hours.
# Koko likes to eat slowly, but still wants to finish eating all the baas before t
# he guards come back.
# Return the minimum integer K such that she can eat all the baas within H hours.

# NOTE
# 1 <= piles.length <= 10^4
# piles.length <= H <= 10^9
# 1 <= piles[i] <= 10^9

# EXAMPLE
# Input: piles = [3,6,7,11], H = 8
# Output: 4
# Input: piles = [30,11,23,4,20], H = 5
# Output: 30
# Input: piles = [30,11,23,4,20], H = 6
# Output: 23

# 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
```

check-if-it-is-a-straight-line.py

```python
# check-if-it-is-a-straight-line is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def checkStraightLine(self, coordinates):
        """
        :type coordinates: List[List[int]]
        :rtype: bool
        """
        i, j = coordinates[:2]
        return all(i[0] * j[1] - j[0] * i[1] +
                   j[0] * k[1] - k[0] * j[1] +
                   k[0] * i[1] - i[0] * k[1] == 0
                   for k in coordinates)
```

maximum-number-of-non-overlapping-substrings.py

```
# maximum-number-of-non-overlapping-substrings is not found.
# Time:  O(n)
# space: O(1)

class Solution(object):
    def maxNumOfSubstrings(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        def find_right_from_left(s, first, last, left):
            right, i = last[ord(s[left])-ord('a')], left
            while i <= right:
                if first[ord(s[i])-ord('a')] < left:
                    return -1
                right = max(right, last[ord(s[i])-ord('a')])
                i += 1
            return right

        first, last = [float("inf")]*26, [float("-inf")]*26
        for i, c in enumerate(s):
            first[ord(c)-ord('a')] = min(first[ord(c)-ord('a')], i)
            last[ord(c)-ord('a')] = max(last[ord(c)-ord('a')], i)
        result = [""]
        right = float("inf")
        for left, c in enumerate(s):
            if left != first[ord(c)-ord('a')]:
                continue
            new_right = find_right_from_left(s, first, last, left)
            if new_right == -1:
                continue
            if left > right:
                result.append("")
            right = new_right
            result[-1] = s[left:right+1]
        return result


# Time:  O(n)
# space: O(1)
class Solution2(object):
    def maxNumOfSubstrings(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        def find_right_from_left(s, first, last, left):
            right, i = last[ord(s[left])-ord('a')], left
            while i <= right:
                if first[ord(s[i])-ord('a')] < left:
                    return -1
                right = max(right, last[ord(s[i])-ord('a')])
                i += 1
            return right

        first, last = [float("inf")]*26, [float("-inf")]*26
        for i, c in enumerate(s):
            first[ord(c)-ord('a')] = min(first[ord(c)-ord('a')], i)
```

```
        last[ord(c)-ord('a')] = max(last[ord(c)-ord('a')], i)
    intervals = []
    for c in xrange(len(first)):
        if first[c] == float("inf"):
            continue
        left, right = first[c], find_right_from_left(s, first, last, first[c])
        if right != -1:
            intervals.append((right, left))
    intervals.sort()  # Time: O(26log26)
    result, prev = [], -1
    for right, left in intervals:
        if left <= prev:
            continue
        result.append(s[left:right+1])
        prev = right
    return result
```

find-all-the-lonely-nodes.py

```python
# find-all-the-lonely-nodes is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution(object):
    def getLonelyNodes(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result = []
        stk = [root]
        while stk:
            node = stk.pop()
            if not node:
                continue
            if node.left and not node.right:
                result.append(node.left.val)
            elif node.right and not node.left:
                result.append(node.right.val)
            stk.append(node.right)
            stk.append(node.left)
        return result


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def getLonelyNodes(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        def dfs(node, result):
            if not node:
                return
            if node.left and not node.right:
                result.append(node.left.val)
            elif node.right and not node.left:
                result.append(node.right.val)
            dfs(node.left, result)
            dfs(node.right, result)

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

random-pick-with-weight.py

```
# DESC
# Constraints:
# Given an array w of positive integers, where w[i] describes the weight of index
# i(0-indexed), write a function pickIndex which randomly picks an index in propor
# tion to its weight.
# Example 2:
# For example, given an input list of values w = [2, 8], when we pick up a number
# out of it, the chance is that 8 times out of 10 we should pick the number 1 as t
# he answer since it's the second element of the array (w[1] = 8).
# Example 1:

# NOTE
# pickIndex will be called at most 10000 times.
# 1 <= w[i] <= 10^5
# 1 <= w.length <= 10000

# EXAMPLE
# 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 elem
# ent.
# 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 sec
# ond element (index = 1) that has probability of 3/4.
# solution.pickIndex(); // re
# turn 1
# solution.pickIndex(); // return 1
# solution.pickIndex(); // return 1
# solut
# ion.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]
# ......
```

```
# an
# d so on.

# 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

```
# DESC
# Constraints:
# Example 1:
# Example 2:
# 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.

# NOTE
# 1 <= arr.length <= 10^4
# 1 <= k <= arr.length
# Absolute value of elements in the array and x will not exceed 104

# EXAMPLE
# Input: arr = [1,2,3,4,5], k = 4, x = -1
# Output: [1,2,3,4]
# Input: arr = [1,2,3,4,5], k = 4, x = 3
# Output: [1,2,3,4]

# 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]
```

maximum-number-of-events-that-can-be-attended.py

```python
# maximum-number-of-events-that-can-be-attended is not found.
# Time:  O(r + nlogn), r is the max end day of events
# Space: O(n)

import heapq


class Solution(object):
    def maxEvents(self, events):
        """
        :type events: List[List[int]]
        :rtype: int
        """
        events.sort(reverse=True)
        min_heap = []
        result = 0
        for d in xrange(1, max(events, key=lambda x:x[1])[1]+1):
            while events and events[-1][0] == d:
                heapq.heappush(min_heap, events.pop()[1])
            while min_heap and min_heap[0] == d-1:
                heapq.heappop(min_heap)
            if not min_heap:
                continue
            heapq.heappop(min_heap)
            result += 1
        return result
```

falling-squares.py

```
# DESC
# The square is dropped with the bottom edge parallel to the number line, and from
#  a higher height than all currently landed squares. We wait for each square to s
# tick before dropping the next.
# After the second drop of positions[1] = [2, 3]: __aaa __aaa __aaa _aa__ _aa__ --
# ------------ The maximum height of any square is 5. The larger square stays on t
# op of the smaller square despite where its center of gravity is, because squares
#  are infinitely sticky on their bottom edge.
# The squares are infinitely sticky on their bottom edge, and will remain fixed to
#  any positive length surface they touch (either the number line or another squar
# e). Squares dropped adjacent to each other will not stick together prematurely.
# On an infinite number line (x-axis), we drop given squares in the order they are
#  given.
# Example 2:
# Note:
# Example 1:
# Return a list ans of heights. Each height ans[i] represents the current highest
# height of any square we have dropped, after dropping squares represented by posi
# tions[0], positions[1], ..., positions[i].
# After the first drop of positions[0] = [1, 2]: _aa _aa ------- The maximum heigh
# t of any square is 2.
# The i-th square dropped (positions[i] = (left, side_length)) is a square with th
# e left-most point being positions[i][0] and sidelength positions[i][1].
# After the third drop of positions[1] = [6, 1]: __aaa __aaa __aaa _aa _aa___a ---
# ----------- The maximum height of any square is still 5. Thus, we return an answ
# er of [2, 5, 5].

# NOTE
# 1 <= positions.length <= 1000.
# 1 <= positions[i][1] <= 10^6.
# 1 <= positions[i][0] <= 10^8.

# EXAMPLE
# Input: [[1, 2], [2, 3], [6, 1]]
# Output: [2, 5, 5]
# Explanation:
# Input: [[100, 100], [200, 100]]
# Output: [100, 100]
# Explanation: Adjacent squares
#  don't get stuck prematurely - only their bottom edge can stick to surfaces.

# Time:  O(n^2), could be improved to O(nlogn) in cpp by ordered map (bst)
# Space: O(n)

import bisect


class Solution(object):
    def fallingSquares(self, positions):
        result = []
        pos = [-1]
        heights = [0]
        maxH = 0
        for left, side in positions:
            l = bisect.bisect_right(pos, left)
            r = bisect.bisect_left(pos, left+side)
            high = max(heights[l-1:r] or [0]) + side
            pos[l:r] = [left, left+side]          # Time: O(n)
```

```
            heights[l:r] = [high, heights[r-1]]  # Time: O(n)
            maxH = max(maxH, high)
            result.append(maxH)
        return result


class SegmentTree(object):
    def __init__(self, N,
                 query_fn=min,
                 update_fn=lambda x, y: y,
                 default_val=float("inf")):
        self.N = N
        self.H = (N-1).bit_length()
        self.query_fn = query_fn
        self.update_fn = update_fn
        self.default_val = default_val
        self.tree = [default_val] * (2 * N)
        self.lazy = [None] * N

    def __apply(self, x, val):
        self.tree[x] = self.update_fn(self.tree[x], val)
        if x < self.N:
            self.lazy[x] = self.update_fn(self.lazy[x], val)

    def update(self, L, R, h):
        def pull(x):
            while x > 1:
                x //= 2
                self.tree[x] = self.query_fn(self.tree[x*2], self.tree[x*2 + 1])
                if self.lazy[x] is not None:
                    self.tree[x] = self.update_fn(self.tree[x], self.lazy[x])

        L += self.N
        R += self.N
        L0, R0 = L, R
        while L <= R:
            if L & 1:
                self.__apply(L, h)
                L += 1
            if R & 1 == 0:
                self.__apply(R, h)
                R -= 1
            L //= 2
            R //= 2
        pull(L0)
        pull(R0)

    def query(self, L, R):
        def push(x):
            n = 2**self.H
            while n != 1:
                y = x // n
                if self.lazy[y] is not None:
                    self.__apply(y*2, self.lazy[y])
                    self.__apply(y*2 + 1, self.lazy[y])
                    self.lazy[y] = None
                n //= 2

        result = self.default_val
        if L > R:
```

```
            return result

        L += self.N
        R += self.N
        push(L)
        push(R)
        while L <= R:
            if L & 1:
                result = self.query_fn(result, self.tree[L])
                L += 1
            if R & 1 == 0:
                result = self.query_fn(result, self.tree[R])
                R -= 1
            L //= 2
            R //= 2
        return result

    def data(self):
        showList = []
        for i in xrange(self.N):
            showList.append(self.query(i, i))
        return showList


class SegmentTree2(object):
    def __init__(self, nums,
                 query_fn=min,
                 update_fn=lambda x, y: y,
                 default_val=float("inf")):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        N = len(nums)
        self.__original_length = N
        self.__tree_length = 2**(N.bit_length() + (N&(N-1) != 0))-1
        self.__query_fn = query_fn
        self.__update_fn = update_fn
        self.__default_val = default_val
        self.__tree = [default_val for _ in range(self.__tree_length)]
        self.__lazy = [None for _ in range(self.__tree_length)]
        self.__constructTree(nums, 0, self.__original_length-1, 0)

    def update(self, i, j, val):
        self.__updateTree(val, i, j, 0, self.__original_length-1, 0)

    def query(self, i, j):
        return self.__queryRange(i, j, 0, self.__original_length-1, 0)

    def __constructTree(self, nums, left, right, idx):
        if left > right:
            return
        if left == right:
            self.__tree[idx] = self.__update_fn(self.__tree[idx], nums[left])
            return
        mid = left + (right-left)//2
        self.__constructTree(nums, left, mid, idx*2 + 1)
        self.__constructTree(nums, mid+1, right, idx*2 + 2)
        self.__tree[idx] = self.__query_fn(self.__tree[idx*2 + 1], self.__tree[idx*2 + 2])
```

```
    def __apply(self, left, right, idx, val):
        self.__tree[idx] = self.__update_fn(self.__tree[idx], val)
        if left != right:
            self.__lazy[idx*2 + 1] = self.__update_fn(self.__lazy[idx*2 + 1], val)
            self.__lazy[idx*2 + 2] = self.__update_fn(self.__lazy[idx*2 + 2], val)

    def __updateTree(self, val, range_left, range_right, left, right, idx):
        if left > right:
            return
        if self.__lazy[idx] is not None:
            self.__apply(left, right, idx, self.__lazy[idx])
            self.__lazy[idx] = None
        if range_left > right or range_right < left:
            return
        if range_left <= left and right <= range_right:
            self.__apply(left, right, idx, val)
            return
        mid = left + (right-left)//2
        self.__updateTree(val, range_left, range_right, left, mid, idx*2 + 1)
        self.__updateTree(val, range_left, range_right, mid+1, right, idx*2 + 2)
        self.__tree[idx] = self.__query_fn(self.__tree[idx*2 + 1],
                                           self.__tree[idx*2 + 2])

    def __queryRange(self, range_left, range_right, left, right, idx):
        if left > right:
            return self.__default_val
        if self.__lazy[idx] is not None:
            self.__apply(left, right, idx, self.__lazy[idx])
            self.__lazy[idx] = None
        if right < range_left or left > range_right:
            return self.__default_val
        if range_left <= left and right <= range_right:
            return self.__tree[idx]
        mid = left + (right-left)//2
        return self.__query_fn(self.__queryRange(range_left, range_right, left, mid, idx*2 + 1),
                               self.__queryRange(range_left, range_right, mid + 1, right, idx*2 + 2))


# Time:  O(nlogn)
# Space: O(n)
# Segment Tree solution.
class Solution2(object):
    def fallingSquares(self, positions):
        index = set()
        for left, size in positions:
            index.add(left)
            index.add(left+size-1)
        index = sorted(list(index))
        tree = SegmentTree(len(index), max, max, 0)
        # tree = SegmentTree2([0]*len(index), max, max, 0)
        max_height = 0
        result = []
        for left, size in positions:
            L, R = bisect.bisect_left(index, left), bisect.bisect_left(index, left+size-1)
            h = tree.query(L, R) + size
            tree.update(L, R, h)
            max_height = max(max_height, h)
            result.append(max_height)
        return result
```

```
# Time:  O(n * sqrt(n))
# Space: O(n)
class Solution3(object):
    def fallingSquares(self, positions):
        def query(heights, left, right, B, blocks, blocks_read):
            result = 0
            while left % B and left <= right:
                result = max(result, heights[left], blocks[left//B])
                left += 1
            while right % B != B-1 and left <= right:
                result = max(result, heights[right], blocks[right//B])
                right -= 1
            while left <= right:
                result = max(result, blocks[left//B], blocks_read[left//B])
                left += B
            return result

        def update(heights, left, right, B, blocks, blocks_read, h):
            while left % B and left <= right:
                heights[left] = max(heights[left], h)
                blocks_read[left//B] = max(blocks_read[left//B], h)
                left += 1
            while right % B != B-1 and left <= right:
                heights[right] = max(heights[right], h)
                blocks_read[right//B] = max(blocks_read[right//B], h)
                right -= 1
            while left <= right:
                blocks[left//B] = max(blocks[left//B], h)
                left += B

        index = set()
        for left, size in positions:
            index.add(left)
            index.add(left+size-1)
        index = sorted(list(index))
        W = len(index)
        B = int(W**.5)
        heights = [0] * W
        blocks = [0] * (B+2)
        blocks_read = [0] * (B+2)

        max_height = 0
        result = []
        for left, size in positions:
            L, R = bisect.bisect_left(index, left), bisect.bisect_left(index, left+size-1)
            h = query(heights, L, R, B, blocks, blocks_read) + size
            update(heights, L, R, B, blocks, blocks_read, h)
            max_height = max(max_height, h)
            result.append(max_height)
        return result


# Time:  O(n^2)
# Space: O(n)
class Solution4(object):
    def fallingSquares(self, positions):
        """
        :type positions: List[List[int]]
        :rtype: List[int]
```

```
    """
    heights = [0] * len(positions)
    for i in xrange(len(positions)):
        left_i, size_i = positions[i]
        right_i = left_i + size_i
        heights[i] += size_i
        for j in xrange(i+1, len(positions)):
            left_j, size_j = positions[j]
            right_j = left_j + size_j
            if left_j < right_i and left_i < right_j:  # intersect
                heights[j] = max(heights[j], heights[i])

    result = []
    for height in heights:
        result.append(max(result[-1], height) if result else height)
    return result
```

jump-game-iii.py

```
# DESC
# Example 3:
# Example 2:
# Notice that you can not jump outside of the array at any time.
# Constraints:
# Example 1:
# Given an array of non-negative integers arr, you are initially positioned at sta
# rt 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.

# NOTE
# 1 <= arr.length <= 5 * 10^4
# 0 <= start < arr.length
# 0 <= arr[i] < arr.length

# EXAMPLE
# Input: arr = [3,0,2,1,2], start = 2
# Output: false
# Explanation: There is no way t
# o reach at index 1 with value 0.
# 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 -> ind
# ex 3
# index 5 -> index 6 -> index 4 -> index 1 -> index 3
# 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 -> inde
# x 3

# 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
```

expression-add-operators.py

```
# DESC
# Example 2:
# Example 3:
# Constraints:
# Example 1:
# Example 5:
# Example 4:
# Given a string that contains only digits 0-9 and a target value, return all poss
# ibilities to add binary operators (not unary) +, -, or * between the digits so t
# hey evaluate to the target value.

# NOTE
# num only contain digits.
# 0 <= num.length <= 10

# EXAMPLE
# Input: num = "105", target = 5
# Output: ["1*0+5","10-5"]
# Input: num = "232", target = 8
# Output: ["2*3+2", "2+3*2"]
# Input: num = "123", target = 6
# Output: ["1+2+3", "1*2*3"]
# Input: num = "3456237490", target = 9191
# Output: []
# Input: num = "00", target = 0
# Output: ["0+0", "0-0", "0*0"]

# Time:  O(4^n)
# Space: O(n)

class Solution(object):
    def addOperators(self, num, target):
        """
        :type num: str
        :type target: int
        :rtype: List[str]
        """
        result, expr = [], []
        val, i = 0, 0
        val_str = ""
        while i < len(num):
            val = val * 10 + ord(num[i]) - ord('0')
            val_str += num[i]
            # Avoid "00...".
            if str(val) != val_str:
                break
            expr.append(val_str)
            self.addOperatorsDFS(num, target, i + 1, 0, val, expr, result)
            expr.pop()
            i += 1
        return result

    def addOperatorsDFS(self, num, target, pos, operand1, operand2, expr, result):
        if pos == len(num) and operand1 + operand2 == target:
            result.append("".join(expr))
        else:
            val, i = 0, pos
            val_str = ""
```

```
        while i < len(num):
            val = val * 10 + ord(num[i]) - ord('0')
            val_str += num[i]
            # Avoid "00...".
            if str(val) != val_str:
                break

            # Case '+':
            expr.append("+" + val_str)
            self.addOperatorsDFS(num, target, i + 1, operand1 + operand2, val, expr, result)
            expr.pop()

            # Case '-':
            expr.append("-" + val_str)
            self.addOperatorsDFS(num, target, i + 1, operand1 + operand2, -val, expr, result)
            expr.pop()

            # Case '*':
            expr.append("*" + val_str)
            self.addOperatorsDFS(num, target, i + 1, operand1, operand2 * val, expr, result)
            expr.pop()

            i += 1
```

longest-chunked-palindrome-decomposition.py

```python
# DESC
# Example 2:
# Example 1:
# Example 4:
# Example 3:
# Constraints:
# Return the largest possible k such that there exists a_1, a_2, ..., a_k such that:

# NOTE
# Their concatenation a_1 + a_2 + ... + a_k is equal to text;
# text consists only of lowercase English characters.
# Each a_i is a non-empty string;
# 1 <= text.length <= 1000
# For all 1 <= i <= k,  a_i = a_{k+1 - i}.

# EXAMPLE
# Input: text = "aaa"
# Output: 3
# Explanation: We can split the string on "(a)(a)(a)".
# Input: text = "merchant"
# Output: 1
# Explanation: We can split the string on "(mer
# chant)".
# Input: text = "antaprezatepzapreanta"
# Output: 11
# Explanation: We can split the s
# tring on "(a)(nt)(a)(pre)(za)(tpe)(za)(pre)(a)(nt)(a)".
# Input: text = "ghiabcdefhelloadamhelloabcdefghi"
# Output: 7
# Explanation: We can s
# plit the string on "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)".

# Time:  O(n)
# Space: O(1)

# Rabin-Karp Algorithm
class Solution(object):
    def longestDecomposition(self, text):
        """
        :type text: str
        :rtype: int
        """
        def compare(text, l, s1, s2):
            for i in xrange(l):
                if text[s1+i] != text[s2+i]:
                    return False
            return True

        MOD = 10**9+7
        D = 26
        result = 0
        left, right, l, pow_D = 0, 0, 0, 1
        for i in xrange(len(text)):
            left = (D*left + (ord(text[i])-ord('a'))) % MOD
            right = (pow_D*(ord(text[-1-i])-ord('a')) + right) % MOD
            l += 1
            pow_D = (pow_D*D) % MOD
            if left == right and compare(text, l, i-l+1, len(text)-1-i):
```

```
        result += 1
        left, right, l, pow_D = 0, 0, 0, 1
return result
```

convert-sorted-array-to-binary-search-tree.py

```
# DESC
# Given an array where elements are sorted in ascending order, convert it to a hei
# ght balanced BST.
# Example:
# For this problem, a height-balanced binary tree is defined as a binary tree in w
# hich the depth of the two subtrees of every node never differ by more than 1.

# NOTE
#

# EXAMPLE
# Given the sorted array: [-10,-3,0,5,9],
#
# One possible answer is: [0,-3,9,-10,nul
# l,5], which represents the following height balanced BST:
#
#       0
#      / \
#    -
# 3   9
#    /   /
#  -10  5

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

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


class Solution(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        return self.sortedArrayToBSTRecu(nums, 0, len(nums))

    def sortedArrayToBSTRecu(self, nums, start, end):
        if start == end:
            return None
        mid = start + self.perfect_tree_pivot(end - start)
        node = TreeNode(nums[mid])
        node.left = self.sortedArrayToBSTRecu(nums, start, mid)
        node.right = self.sortedArrayToBSTRecu(nums, mid + 1, end)
        return node

    def perfect_tree_pivot(self, n):
        """
        Find the point to partition n keys for a perfect binary search tree
        """
        x = 1
        # find a power of 2 <= n//2
        # while x <= n//2:  # this loop could probably be written more elegantly :)
        #     x *= 2
```

```
        x = 1 << (n.bit_length() - 1)  # use the left bit shift, same as multiplying x by 2**n-1

        if x // 2 - 1 <= (n - x):
            return x - 1  # case 1: the left subtree of the root is perfect and the right subtree has less nod
        else:
            return n - x // 2  # case 2 == n - (x//2 - 1) - 1 : the left subtree of the root
                               # has more nodes and the right subtree is perfect.

# Time:  O(n)
# Space: O(logn)
class Solution2(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        self.iterator = iter(nums)
        return self.helper(0, len(nums))

    def helper(self, start, end):
        if start == end:
            return None

        mid = (start + end) // 2
        left = self.helper(start, mid)
        current = TreeNode(next(self.iterator))
        current.left = left
        current.right = self.helper(mid+1, end)
        return current
```

single-number-ii.py

```
# DESC
# Note:
# Example 2:
# Your algorithm should have a linear runtime complexity. Could you implement it w
# ithout using extra memory?
# Example 1:
# Given a non-empty array of integers, every element appears three times except fo
# r one, which appears exactly once. Find that single one.

# NOTE
#

# EXAMPLE
# Input: [0,1,0,1,0,1,99]
# Output: 99
# Input: [2,2,3,2]
# Output: 3

# 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):
        """
        :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
```

n-th-tribonacci-number.py

```
# DESC
# The Tribonacci sequence Tn is defined as follows:
# Example 2:
# Constraints:
# Given n, return the value of Tn.
# Example 1:
# T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.

# NOTE
# The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1.
# 0 <= n <= 37

# EXAMPLE
# Input: n = 25
# Output: 1389537
# Input: n = 4
# Output: 4
# Explanation:
# T_3 = 0 + 1 + 1 = 2
# T_4 = 1 + 1 + 2 = 4

# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def tribonacci(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)) \
                     for col in ZB] for row in A]

        T = [[1, 1, 0],
             [1, 0, 1],
             [1, 0, 0]]
        return matrix_mult([[1, 0, 0]], matrix_expo(T, n))[0][1]  # [a1, a0, a(-1)] * T^n


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

```
    """
    :type n: int
    :rtype: int
    """
    a, b, c = 0, 1, 1
    for _ in xrange(n):
        a, b, c = b, c, a+b+c
    return a
```

array-of-doubled-pairs.py

```
# DESC
# Note:
# Example 2:
# Example 1:
# Given an array of integers A with even length, return true if and only if it is
# possible to reorder it such that A[2 * i + 1] = 2 * A[2 * i] for every 0 <= i <
# len(A) / 2.
# Example 4:
# Example 3:

# NOTE
# A.length is even
# -100000 <= A[i] <= 100000
# 0 <= A.length <= 30000

# EXAMPLE
# Input: [2,1,2,6]
# Output: false
# Input: [3,1,3,6]
# Output: false
# Input: [1,2,4,16,8,4]
# Output: false
# 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].

# 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
```

the-kth-factor-of-n.py

```python
# the-kth-factor-of-n is not found.
# Time:  O(sqrt(n))
# Space: O(1)

class Solution(object):
    def kthFactor(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        def kth_factor(n, k=0):
            mid = None
            i = 1
            while i*i <= n:
                if not n%i:
                    mid = i
                    k -= 1
                    if not k:
                        break
                i += 1
            return mid, -k

        mid, count = kth_factor(n)
        total = 2*count-(mid*mid == n)
        if k > total:
            return -1
        result = kth_factor(n, k if k <= count else total-(k-1))[0]
        return result if k <= count else n//result


# Time:  O(sqrt(n))
# Space: O(sqrt(n))
class Solution2(object):
    def kthFactor(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        result = []
        i = 1
        while i*i <= n:
            if not n%i:
                if i*i != n:
                    result.append(i)
                k -= 1
                if not k:
                    return i
            i += 1
        return -1 if k > len(result) else n//result[-k]
```

sliding-window-maximum.py

```
# DESC
# Example:
# Follow up:
#
# Could you solve it in linear time?
# Constraints:
# Given an array nums, there is a sliding window of size k which is moving from th
# e very left of the array to the very right. You can only see the k numbers in th
# e window. Each time the sliding window moves right by one position. Return the m
# ax sliding window.

# NOTE
# -10^4 <= nums[i] <= 10^4
# 1 <= nums.length <= 10^5
# 1 <= k <= nums.length

# EXAMPLE
# Input: nums = [1,3,-1,-3,5,3,6,7], and k = 3
# Output: [3,3,5,5,6,7]
# Explanation:
#
#
# Window position                Max
# ---------------               -----
# [1  3
#  -1] -3  5  3  6  7       3
#  1 [3  -1  -3] 5  3  6  7       3
#  1  3 [-1  -3  5]
# 3  6  7       5
#  1  3  -1 [-3  5  3] 6  7       5
#  1  3  -1  -3 [5  3  6] 7
#   6
#  1  3  -1  -3  5 [3  6  7]      7

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

from collections import deque


class Solution(object):
    def maxSlidingWindow(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        result, dq = [], deque()
        for i in xrange(len(nums)):
            if dq and i-dq[0] == k:
                dq.popleft()
            while dq and nums[dq[-1]] <= nums[i]:
                dq.pop()
            dq.append(i)
            if i >= k-1:
                result.append(nums[dq[0]])
        return result
```

remove-9.py

```python
# remove-9 is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def newInteger(self, n):
        """
        :type n: int
        :rtype: int
        """
        result, base = 0, 1
        while n > 0:
            result += (n%9) * base
            n /= 9
            base *= 10
        return result
```

binary-tree-vertical-order-traversal.py

```python
# binary-tree-vertical-order-traversal is not found.
# Time:  O(n)
# Space: O(n)

import collections


# BFS + hash solution.
class Solution(object):
    def verticalOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        cols = collections.defaultdict(list)
        queue = [(root, 0)]
        for node, i in queue:
            if node:
                cols[i].append(node.val)
                queue += (node.left, i - 1), (node.right, i + 1)
        return [cols[i] for i in xrange(min(cols.keys()),
                                        max(cols.keys()) + 1)] if cols else []
```

similar-string-groups.py

```
# DESC
# Together, these form two connected groups by similarity: {"tars", "rats", "arts"
# } and {"star"}.  Notice that "tars" and "arts" are in the same group even though
#  they are not similar.  Formally, each group is such that a word is in the group
#  if and only if it is similar to at least one other word in the group.
# Two strings X and Y are similar if we can swap two letters (in different positio
# ns) of X, so that it equals Y. Also two strings X and Y are similar if they are
# equal.
# "tars"
# Constraints:
# Example 1:
# We are given a list A of strings.  Every string in A is an anagram of every othe
# r string in A.  How many groups are there?
# For example, "tars" and "rats" are similar (swapping at positions 0 and 2), and
# "rats" and "arts" are similar, but "star" is not similar to "tars", "rats", or "
# arts".

# NOTE
# The judging time limit has been increased for this question.
# All words in A have the same length and are anagrams of each other.
# 1 <= A.length <= 2000
# All words in A consist of lowercase letters only.
# 1 <= A[i].length <= 1000
# A.length * A[i].length <= 20000

# EXAMPLE
# Input: A = ["tars","rats","arts","star"]
# Output: 2

# Time:  O(n^2 * l) ~ O(n * l^4)
# Space: O(n) ~ O(n * l^3)

import collections
import itertools


class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)
        self.__size = 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)
        self.__size -= 1
        return True

    def size(self):
        return self.__size
```

```
class Solution(object):
    def numSimilarGroups(self, A):
        def isSimilar(a, b):
            diff = 0
            for x, y in itertools.izip(a, b):
                if x != y:
                    diff += 1
                    if diff > 2:
                        return False
            return diff == 2

        N, L = len(A), len(A[0])
        union_find = UnionFind(N)
        if N < L*L:
            for (i1, word1), (i2, word2) in \
                    itertools.combinations(enumerate(A), 2):
                if isSimilar(word1, word2):
                    union_find.union_set(i1, i2)
        else:
            buckets = collections.defaultdict(list)
            lookup = set()
            for i in xrange(len(A)):
                word = list(A[i])
                if A[i] not in lookup:
                    buckets[A[i]].append(i)
                    lookup.add(A[i])
                for j1, j2 in itertools.combinations(xrange(L), 2):
                    word[j1], word[j2] = word[j2], word[j1]
                    buckets["".join(word)].append(i)
                    word[j1], word[j2] = word[j2], word[j1]
            for word in A:  # Time:  O(n * l^4)
                for i1, i2 in itertools.combinations(buckets[word], 2):
                    union_find.union_set(i1, i2)
        return union_find.size()
```

jump-game.py

```
# DESC
# Example 2:
# Constraints:
# Each element in the array represents your maximum jump length at that position.
# Example 1:
# Given an array of non-negative integers, you are initially positioned at the fir
# st index of the array.
# Determine if you are able to reach the last index.

# NOTE
# 0 <= nums[i][j] <= 10^5
# 1 <= nums.length <= 3 * 10^4

# EXAMPLE
# Input: nums = [3,2,1,0,4]
# Output: false
# Explanation: You will always arrive at i
# ndex 3 no matter what. Its maximum jump length is 0, which makes it impossible t
# o reach the last index.
# 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.

# 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
```

brace-expansion-ii.py

```
# DESC
# Given an expression representing a set of words under the given grammar, return
# the sorted list of words that the expression represents.
# Example 1:
# Example 2:
# Grammar can best be understood through simple examples:
# Under a grammar given below, strings can represent a set of lowercase words.  Le
# t's use R(expr) to denote the set of words the expression represents.
# Formally, the 3 rules for our grammar:
# Constraints:

# NOTE
# R("w") = {"w"}
# For expressions e_1 and e_2, we have R(e_1 + e_2) = {a + b for (a, b) in R(e_1)
# × R(e_2)}, where + denotes concatenation, and × denotes the cartesian product.
# For expressions e_1, e_2, ... , e_k with k >= 2, we have R({e_1,e_2,...}) = R(e_
# 1)  R(e_2)  ...
# Single letters represent a singleton set containing that word.
#
#                 R("a") = {"a"
# }
#                 R("w") = {"w"}
# When we take a comma delimited list of 2 or more expressions, we take the union
# of possibilities.
#
#                 R("{a,b,c}") = {"a","b","c"}
#                 R("{{a,b},{b,c}}") = {"a","
# b","c"} (notice the final set only contains each word at most once)
# The given expression represents a set of words based on the grammar given in the
#  description.
# R("{a,b,c}") = {"a","b","c"}
# R("{a,b}{c,d}") = {"ac","ad","bc","bd"}
# For every lowercase letter x, we have R(x) = {x}
# expression[i] consists of '{', '}', ','or lowercase English letters.
# When we concatenate two expressions, we take the set of possible concatenations
# between two words where the first word comes from the first expression and the s
# econd word comes from the second expression.
#
#                 R("{a,b}{c,d}") = {"ac","ad","b
# c","bd"}
#                 R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg"
# , "acdfh", "acefg", "acefh"}
# R("a") = {"a"}
# 1 <= expression.length <= 60
# R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh",
# "acefg", "acefh"}
# R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains each word
#  at most once)

# EXAMPLE
# Input: "{{a,z},a{b,c},{ab,z}}"
# Output: ["a","ab","ac","z"]
# Explanation: Each dis
# tinct word is written only once in the final answer.
# Input: "{a,b}{c,{d,e}}"
# Output: ["ac","ad","ae","bc","bd","be"]

# Time:  O(p*l * log(p*l)), p is the production of all number of options
```

```
#                         , l is the length of a word
# Space: O(p*l)

import itertools


class Solution(object):
    def braceExpansionII(self, expression):
        """
        :type expression: str
        :rtype: List[str]
        """
        def form_words(options):
            words = map("".join, itertools.product(*options))
            words.sort()
            return words

        def generate_option(expr, i):
            option_set = set()
            while i[0] != len(expr) and expr[i[0]] != "}":
                i[0] += 1  # { or ,
                for option in generate_words(expr, i):
                    option_set.add(option)
            i[0] += 1  # }
            option = list(option_set)
            option.sort()
            return option

        def generate_words(expr, i):
            options = []
            while i[0] != len(expr) and expr[i[0]] not in ",}":
                tmp = []
                if expr[i[0]] not in "{,}":
                    tmp.append(expr[i[0]])
                    i[0] += 1  # a-z
                elif expr[i[0]] == "{":
                    tmp = generate_option(expr, i)
                options.append(tmp)
            return form_words(options)

        return generate_words(expression, [0])


class Solution2(object):
    def braceExpansionII(self, expression):
        """
        :type expression: str
        :rtype: List[str]
        """
        def form_words(options):
            words = []
            total = 1
            for opt in options:
                total *= len(opt)
            for i in xrange(total):
                tmp = []
                for opt in reversed(options):
                    i, c = divmod(i, len(opt))
                    tmp.append(opt[c])
                tmp.reverse()
```

```
            words.append("".join(tmp))
        words.sort()
        return words

    def generate_option(expr, i):
        option_set = set()
        while i[0] != len(expr) and expr[i[0]] != "}":
            i[0] += 1  # { or ,
            for option in generate_words(expr, i):
                option_set.add(option)
        i[0] += 1  # }
        option = list(option_set)
        option.sort()
        return option

    def generate_words(expr, i):
        options = []
        while i[0] != len(expr) and expr[i[0]] not in ",}":
            tmp = []
            if expr[i[0]] not in "{,}":
                tmp.append(expr[i[0]])
                i[0] += 1  # a-z
            elif expr[i[0]] == "{":
                tmp = generate_option(expr, i)
            options.append(tmp)
        return form_words(options)

    return generate_words(expression, [0])
```

reverse-only-letters.py

```
# DESC
# Example 1:
# Given a string S, return the "reversed" string where all characters that are not
#  a letter stay in the same place, and all letters reverse their positions.
# Example 3:
# Example 2:
# Note:

# NOTE
# S doesn't contain \ or "
# 33 <= S[i].ASCIIcode <= 122
# S.length <= 100

# EXAMPLE
# Input: "ab-cd"
# Output: "dc-ba"
# Input: "Test1ng-Leet=code-Q!"
# Output: "Qedo1ct-eeLg=ntse-T!"
# Input: "a-bC-dEf-ghIj"
# Output: "j-Ih-gfE-dCba"

# Given a string S, return the "reversed" string where all characters that are not a letter stay in the same p
#
#
#
# Example 1:
#
# Input: "ab-cd"
# Output: "dc-ba"
# Example 2:
#
# Input: "a-bC-dEf-ghIj"
# Output: "j-Ih-gfE-dCba"
# Example 3:
#
# Input: "Test1ng-Leet=code-Q!"
# Output: "Qedo1ct-eeLg=ntse-T!"
#
#
# Note:
#
# S.length <= 100
# 33 <= S[i].ASCIIcode <= 122
# S doesn't contain \ or "


# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reverseOnlyLetters(self, S):
        """
        :type S: str
        :rtype: str
        """
        def getNext(S):
            for i in reversed(xrange(len(S))):
                if S[i].isalpha():
```

```
            yield S[i]

result = []
letter = getNext(S)
for i in xrange(len(S)):
    if S[i].isalpha():
        result.append(letter.next())
    else:
        result.append(S[i])
return "".join(result)
```

max-consecutive-ones-ii.py

```python
# max-consecutive-ones-ii is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, prev, curr = 0, 0, 0
        for n in nums:
            if n == 0:
                result = max(result, prev+curr+1)
                prev, curr = curr, 0
            else:
                curr += 1
        return min(max(result, prev+curr+1), len(nums))
```

maximum-average-subtree.py

```
# maximum-average-subtree is not found.
# 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 maximumAverageSubtree(self, root):
        """
        :type root: TreeNode
        :rtype: float
        """
        def maximumAverageSubtreeHelper(root, result):
            if not root:
                return [0.0, 0]
            s1, n1 = maximumAverageSubtreeHelper(root.left, result)
            s2, n2 = maximumAverageSubtreeHelper(root.right, result)
            s = s1+s2+root.val
            n = n1+n2+1
            result[0] = max(result[0], s / n)
            return [s, n]

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

remove-k-digits.py

```
# DESC
# Example 3:
# Note:
# Example 1:
# Given a non-negative integer num represented as a string, remove k digits from t
# he number so that the new number is the smallest possible.
# Example 2:

# NOTE
# The given num does not contain any leading zero.
# The length of num is less than 10002 and will be  k.

# EXAMPLE
# Input: num = "1432219", k = 3
# Output: "1219"
# Explanation: Remove the three digit
# s 4, 3, and 2 to form the new number 1219 which is the smallest.
# Input: num = "10", k = 2
# Output: "0"
# Explanation: Remove all the digits from the
#  number and it is left with nothing which is 0.
# 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.

# 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'
```

last-stone-weight.py

```
# DESC
# Note:
# Each turn, we choose the two heaviest stones and smash them together.  Suppose t
# he stones have weights x and y with x <= y.  The result of this smash is:
# Example 1:
# We have a collection of stones, each stone has a positive integer weight.
# At the end, there is at most 1 stone left.  Return the weight of this stone (or
# 0 if there are no stones left.)

# NOTE
# 1 <= stones[i] <= 1000
# If x == y, both stones are totally destroyed;
# 1 <= stones.length <= 30
# If x != y, the stone of weight x is totally destroyed, and the stone of weight y
#  has new weight y-x.

# EXAMPLE
# Input: [2,7,4,1,8,1]
# Output: 1
# Explanation:
# We combine 7 and 8 to get 1 so the
# array converts to [2,4,1,1,1] then,
# we combine 2 and 4 to get 2 so the array con
# verts to [2,1,1,1] then,
# we combine 2 and 1 to get 1 so the array converts to [1
# ,1,1] then,
# we combine 1 and 1 to get 0 so the array converts to [1] then that's
#  the value of last stone.

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

import heapq


class Solution(object):
    def lastStoneWeight(self, stones):
        """
        :type stones: List[int]
        :rtype: int
        """
        max_heap = [-x for x in stones]
        heapq.heapify(max_heap)
        for i in xrange(len(stones)-1):
            x, y = -heapq.heappop(max_heap), -heapq.heappop(max_heap)
            heapq.heappush(max_heap, -abs(x-y))
        return -max_heap[0]
```

construct-quad-tree.py

```
# DESC
# Notice that you can assign the value of a node to True or False when isLeaf is F
# alse, and both are accepted in the answer.
# Constraints:
# The output represents the serialized format of a Quad-Tree using level order tra
# versal, where null signifies a path terminator where no node exists below.
# If you want to know more about the Quad-Tree, you can refer to the wiki.
# 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 2:
# A Quad-Tree is a tree data structure in which each internal node has exactly fou
# r children. Besides, each node has two attributes:
# Example 5:
# 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].
# Example 4:
# Given a n * n matrix grid of 0's and 1's only. We want to represent the grid wit
# h a Quad-Tree.
# We can construct a Quad-Tree from a two-dimensional area using the following steps:
# Example 3:
# Example 1:
# Quad-Tree format:
# Return the root of the Quad-Tree representing the grid.

# NOTE
# 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.
# n == grid.length == grid[i].length
# n == 2^x where 0 <= x <= 6
# 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.
# 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 f
# our children.

# EXAMPLE
# class Node {
#     public boolean val;
#     public boolean isLeaf;
#     public Node
# topLeft;
#     public Node topRight;
#     public Node bottomLeft;
#     public Node b
# ottomRight;
# }
# Input: grid = [[1,1],[1,1]]
# Output: [[1,1]]
# 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]]
# 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 int
# o four sub-grids.
# The topLeft, bottomLeft and bottomRight each has the same valu
# e.
# The topRight have different values so we divide it into 4 sub-grids where eac
# h has the same value.
# Explanation is shown in the photo below:
# Input: grid = [[0,1],[1,0]]
# 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 a
# nd 1 represents True in the photo representing the Quad-Tree.
# Input: grid = [[0]]
# Output: [[1,0]]

# 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


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))
```

missing-element-in-sorted-array.py

```python
# missing-element-in-sorted-arra is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def missingElement(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        def missing_count(nums, x):
            return (nums[x]-nums[0]+1)-(x-0+1)

        def check(nums, k, x):
            return k <= missing_count(nums, x)

        left, right = 0, len(nums)-1
        while left <= right:
            mid = left + (right-left)//2
            if check(nums, k, mid):
                right = mid-1
            else:
                left = mid+1
        assert(not check(nums, k, right))
        return nums[right] + (k-missing_count(nums, right))
```

flood-fill.py

```
# DESC
# Given a coordinate (sr, sc) representing the starting pixel (row and column) of
# the flood fill, and a pixel value newColor, "flood fill" the image.
# Example 1:
# Note:
# (sr, sc)
# An image is represented by a 2-D array of integers, each integer representing th
# e pixel value of the image (from 0 to 65535).
# To perform a "flood fill", consider the starting pixel, plus any pixels connecte
# d 4-directionally to the starting pixel of the same color as the starting pixel,
#  plus any pixels connected 4-directionally to those pixels (also with the same c
# olor as the starting pixel), and so on.  Replace the color of all of the aforeme
# ntioned pixels with the newColor.
# At the end, return the modified image.

# NOTE
# The length of image and image[0] will be in the range [1, 50].
# The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image
# [0].length.
# The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

# EXAMPLE
# Input:
# image = [[1,1,1],[1,1,0],[1,0,1]]
# sr = 1, sc = 1, newColor = 2
# Output: [
# [2,2,2],[2,2,0],[2,0,1]]
# Explanation:
# From the center of the image (with positi
# on (sr, sc) = (1, 1)), all pixels connected
# by a path of the same color as the
# starting pixel are colored with the new color.
# Note the bottom corner is not col
# ored 2, because it is not 4-directionally connected
# to the starting pixel.

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

class Solution(object):
    def floodFill(self, image, sr, sc, newColor):
        """
        :type image: List[List[int]]
        :type sr: int
        :type sc: int
        :type newColor: int
        :rtype: List[List[int]]
        """
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        def dfs(image, r, c, newColor, color):
            if not (0 <= r < len(image) and \
                    0 <= c < len(image[0]) and \
                    image[r][c] == color):
                return

            image[r][c] = newColor
            for d in directions:
```

```
        dfs(image, r+d[0], c+d[1], newColor, color)

color = image[sr][sc]
if color == newColor: return image
dfs(image, sr, sc, newColor, color)
return image
```

median-of-two-sorted-arrays.py

```
# DESC
# You may assume nums1 and nums2 cannot be both empty.
# Find the median of the two sorted arrays. The overall run time complexity should
#  be O(log (m+n)).
# There are two sorted arrays nums1 and nums2 of size m and n respectively.
# Example 2:
# Example 1:

# NOTE
#

# EXAMPLE
# nums1 = [1, 3]
# nums2 = [2]
#
# The median is 2.0
# nums1 = [1, 2]
# nums2 = [3, 4]
#
# The median is (2 + 3)/2 = 2.5

# Time:  O(log(min(m, n)))
# Space: O(1)

class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """
        len1, len2 = len(nums1), len(nums2)
        if (len1 + len2) % 2 == 1:
            return self.getKth(nums1, nums2, (len1 + len2)/2 + 1)
        else:
            return (self.getKth(nums1, nums2, (len1 + len2)/2) +
                    self.getKth(nums1, nums2, (len1 + len2)/2 + 1)) * 0.5

    def getKth(self, A, B, k):
        m, n = len(A), len(B)
        if m > n:
            return self.getKth(B, A, k)

        left, right = 0, m
        while left < right:
            mid = left + (right - left) / 2
            if 0 <= k - 1 - mid < n and A[mid] >= B[k - 1 - mid]:
                right = mid
            else:
                left = mid + 1

        Ai_minus_1 = A[left - 1] if left - 1 >= 0 else float("-inf")
        Bj = B[k - 1 - left] if k - 1 - left >= 0 else float("-inf")

        return max(Ai_minus_1, Bj)


# Time:  O(log(max(m, n)) * log(max_val - min_val))
```

```
# Space: O(1)
# Generic solution.
class Solution_Generic(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """
        len1, len2 = len(nums1), len(nums2)
        if (len1 + len2) % 2 == 1:
            return self.getKth([nums1, nums2], (len1 + len2)/2 + 1)
        else:
            return (self.getKth([nums1, nums2], (len1 + len2)/2) +
                    self.getKth([nums1, nums2], (len1 + len2)/2 + 1)) * 0.5

    def getKth(self, arrays, k):
        def binary_search(array, left, right, target, compare):
            while left <= right:
                mid = left + (right - left) / 2
                if compare(array, mid, target):
                    right = mid - 1
                else:
                    left = mid + 1
            return left

        def match(arrays, num, target):
            res = 0
            for array in arrays:
                if array:
                    res += binary_search(array, 0, len(array) - 1, num,
                                         lambda array, x, y: array[x] > y)
            return res >= target

        left, right = float("inf"), float("-inf")
        for array in arrays:
            if array:
                left = min(left, array[0])
                right = max(right, array[-1])

        return binary_search(arrays, left, right, k, match)

class Solution_3(object):
    def findMedianSortedArrays(self, A, B):

        if A is None and B is None:
            return -1.0
        lenA = len(A)
        lenB = len(B)
        lenn = lenA + lenB

        indexA,indexB,indexC = 0,0,0
        C = [False for i in xrange(lenn)]
        while indexA < lenA and indexB < lenB:
            if A[indexA] < B[indexB]:
                C[indexC] = A[indexA]
                indexC += 1
                indexA += 1
            else:
                C[indexC] = B[indexB]
```

```
            indexC += 1
            indexB += 1

    while indexA < lenA:
        C[indexC] = A[indexA]
        indexC += 1
        indexA += 1

    while indexB < lenB:
        C[indexC] = B[indexB]
        indexC += 1
        indexB += 1

    indexM1 = (lenn - 1) / 2
    indexM2 = lenn / 2

    if (lenn % 2 == 0):
        return (C[indexM1] + C[indexM2]) / 2.0
    else:
        return C[indexM2] / 1.0
```

smallest-rectangle-enclosing-black-pixels.py

```python
# smallest-rectangle-enclosing-black-pixels is not found.
# Time:  O(nlogn)
# Space: O(1)

import bisect
import itertools


class Solution(object):
    def minArea(self, image, x, y):
        """
        :type image: List[List[str]]
        :type x: int
        :type y: int
        :rtype: int
        """
        def binarySearch(left, right, find, image, has_one):
            while left <= right:  # O(logn) times
                mid = left + (right - left) / 2
                if find(image, has_one, mid):  # Time: O(n)
                    right = mid - 1
                else:
                    left = mid + 1
            return left


        searchColumns = lambda image, has_one, mid: any([int(row[mid]) for row in image]) == has_one
        left = binarySearch(0, y - 1, searchColumns, image, True)
        right = binarySearch(y + 1, len(image[0]) - 1, searchColumns, image, False)

        searchRows = lambda image, has_one, mid: any(itertools.imap(int, image[mid])) == has_one
        top = binarySearch(0, x - 1, searchRows, image, True)
        bottom = binarySearch(x + 1, len(image) - 1, searchRows, image, False)

        return (right - left) * (bottom - top)
```

design-excel-sum-formula.py

```
# design-excel-sum-formula is not found.
# Time:  set: O((r * c)^2)
#        get: O(1)
#        sum: O((r * c)^2)
# Space: O(r * c)

import collections


class Excel(object):

    def __init__(self, H, W):
        """
        :type H: int
        :type W: str
        """
        self.__exl = [[0 for _ in xrange(ord(W)-ord('A')+1)] \
                      for _ in xrange(H+1)]
        self.__fward = collections.defaultdict(lambda : collections.defaultdict(int))
        self.__bward = collections.defaultdict(set)


    def set(self, r, c, v):
        """
        :type r: int
        :type c: str
        :type v: int
        :rtype: void
        """
        self.__reset_dependency(r, c)
        self.__update_others(r, c, v)


    def get(self, r, c):
        """
        :type r: int
        :type c: str
        :rtype: int
        """
        return self.__exl[r][ord(c) - ord('A')]


    def sum(self, r, c, strs):
        """
        :type r: int
        :type c: str
        :type strs: List[str]
        :rtype: int
        """
        self.__reset_dependency(r, c)
        result = self.__calc_and_update_dependency(r, c, strs)
        self.__update_others(r, c, result)
        return result


    def __reset_dependency(self, r, c):
        key = (r, c)
        if key in self.__bward.keys():
```

```
        for k in self.__bward[key]:
            self.__fward[k].pop(key, None)
        self.__bward[key] = set()


def __calc_and_update_dependency(self, r, c, strs):
    result = 0
    for s in strs:
        s, e = s.split(':')[0], s.split(':')[1] if ':' in s else s
        left, right, top, bottom = ord(s[0])-ord('A'), ord(e[0])-ord('A'), int(s[1:]), int(e[1:])
        for i in xrange(top, bottom+1):
            for j in xrange(left, right+1):
                result += self.__exl[i][j]
                self.__fward[(i, chr(ord('A')+j))][(r, c)] += 1
                self.__bward[(r, c)].add((i, chr(ord('A')+j)))
    return result


def __update_others(self, r, c, v):
    prev = self.__exl[r][ord(c)-ord('A')]
    self.__exl[r][ord(c)-ord('A')] = v
    q = collections.deque()
    q.append(((r, c), v-prev))
    while q:
        key, diff = q.popleft()
        if key in self.__fward:
            for k, count in self.__fward[key].iteritems():
                q.append((k, diff*count))
                self.__exl[k[0]][ord(k[1])-ord('A')] += diff*count
```

count-largest-group.py

```
# count-largest-grou is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections


class Solution(object):
    def countLargestGroup(self, n):
        """
        :type n: int
        :rtype: int
        """
        count = collections.Counter()
        for x in xrange(1, n+1):
            count[sum(map(int, str(x)))] += 1
        max_count = max(count.itervalues())
        return sum(v == max_count for v in count.itervalues())
```

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

```
# DESC
# Follow up:
# Suppose an array sorted in ascending order is rotated at some pivot unknown to y
# ou beforehand.
# Example 2:
# (i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).
# [0,0,1,2,2,5,6]
# You are given a target value to search. If found in the array return true, other
# wise return false.
# Example 1:

# NOTE
# Would this affect the run-time complexity? How and why?
# This is a follow up problem to Search in Rotated Sorted Array, where nums may co
# ntain duplicates.

# EXAMPLE
# Input: nums = [2,5,6,0,0,1,2], target = 3
# Output: false
# Input: nums = [2,5,6,0,0,1,2], target = 0
# Output: true

# 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

```
# DESC
# Example 1:
# Given two words word1 and word2, find the minimum number of steps required to ma
# ke word1 and word2 the same, where in each step you can delete one character in
# either string.
# Note:

# NOTE
# Characters in given words can only be lower-case letters.
# The length of given words won't exceed 500.

# EXAMPLE
# Input: "sea", "eat"
# Output: 2
# Explanation: You need one step to make "sea" to "e
# a" and another step to make "eat" to "ea".

# 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)
```

max-points-on-a-line.py

```
# max-points-on-a-line is not found.
# Share
# Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.
#
# Example 1:
#
# Input: [[1,1],[2,2],[3,3]]
# Output: 3
# Explanation:
# ^
# |
# |        o
# |     o
# |  o
# +------------->
# 0  1  2  3  4
# Example 2:
#
# Input: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
# Output: 4
# Explanation:
# ^
# |
# |  o
# |     o        o
# |        o
# |  o        o
# +------------------->
# 0  1  2  3  4  5  6
# NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new me

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

import collections


# Definition for a point
class Point(object):
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b

class Solution(object):
    def maxPoints(self, points):
        """
        :type points: List[Point]
        :rtype: int
        """
        max_points = 0
        for i, start in enumerate(points):
            slope_count, same = collections.defaultdict(int), 1
            for j in xrange(i + 1, len(points)):
                end = points[j]
                if start.x == end.x and start.y == end.y:
                    same += 1
                else:
                    slope = float("inf")
```

```
            if start.x - end.x != 0:
                slope = (start.y - end.y) * 1.0 / (start.x - end.x)
            slope_count[slope] += 1

    current_max = same
    for slope in slope_count:
        current_max = max(current_max, slope_count[slope] + same)

    max_points = max(max_points, current_max)

return max_points
```

find-minimum-in-rotated-sorted-array-ii.py

```
# DESC
# The array may contain duplicates.
# Suppose an array sorted in ascending order is rotated at some pivot unknown to y
# ou beforehand.
# Note:
# (i.e.,  [0,1,2,4,5,6,7] might become  [4,5,6,7,0,1,2]).
# Example 1:
# Find the minimum element.
# [0,1,2,4,5,6,7]
# Example 2:

# NOTE
# This is a follow up problem to Find Minimum in Rotated Sorted Array.
# Would allow duplicates affect the run-time complexity? How and why?

# EXAMPLE
# Input: [2,2,2,0,1]
# Output: 0
# Input: [1,3,5]
# Output: 1

# Time:  O(logn) ~ O(n)
# Space: O(1)

class Solution(object):
    def findMin(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[right]:
                right -= 1
            elif nums[mid] < nums[right]:
                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]:
                left += 1
            elif nums[mid] < nums[left]:
                right = mid
```

```
        else:
            left = mid + 1

    return nums[left]
```

length-of-longest-fibonacci-subsequence.py

```
# DESC
# Example 1:
# Note:
# Example 2:
# (Recall that a subsequence is derived from another sequence A by deleting any nu
# mber of elements (including none) from A, without changing the order of the rema
# ining elements.  For example, [3, 5, 8] is a subsequence of [3, 4, 5, 6, 7, 8].)
# Given a strictly increasing array A of positive integers forming a sequence, fin
# d the length of the longest fibonacci-like subsequence of A.  If one does not ex
# ist, return 0.
# A sequence X_1, X_2, ..., X_n is fibonacci-like if:

# NOTE
# 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++.)
# n >= 3
# 3 <= A.length <= 1000
# X_i + X_{i+1} = X_{i+2} for all i + 2 <= n

# EXAMPLE
# Input: [1,3,7,11,12,14,18]
# Output: 3
# Explanation:
# The longest subsequence that i
# s fibonacci-like:
# [1,11,12], [3,11,14] or [7,11,18].
# Input: [1,2,3,4,5,6,7,8]
# Output: 5
# Explanation:
# The longest subsequence that is
# fibonacci-like: [1,2,3,5,8].

# 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
```

find-a-corresponding-node-of-a-binary-tree-in-a-clone-of-that-tree.py

```
# find-a-corresponding-node-of-a-binary-tree-in-a-clone-of-that-tree is not found.
# Time:  O(n)
# Space: O(h)

import itertools


# 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 getTargetCopy(self, original, cloned, target):
        """
        :type original: TreeNode
        :type cloned: TreeNode
        :type target: TreeNode
        :rtype: TreeNode
        """
        def preorder_gen(node):
            stk = [node]
            while stk:
                node = stk.pop()
                if not node:
                    continue
                yield node
                stk.append(node.right)
                stk.append(node.left)

        for node1, node2 in itertools.izip(preorder_gen(original),
                                           preorder_gen(cloned)):
            if node1 == target:
                return node2
```

word-pattern-ii.py

```
# word-pattern-ii is not found.
# Time:  O(n * C(n - 1, c - 1)), n is length of str, c is unique count of pattern,
#                                there are H(n - c, c - 1) = C(n - 1, c - 1) possible splits of string,
#                                and each one costs O(n) to check if it matches the word pattern.
# Space: O(n + c)

class Solution(object):
    def wordPatternMatch(self, pattern, str):
        """
        :type pattern: str
        :type str: str
        :rtype: bool
        """
        w2p, p2w = {}, {}
        return self.match(pattern, str, 0, 0, w2p, p2w)


    def match(self, pattern, str, i, j, w2p, p2w):
        is_match = False
        if i == len(pattern) and j == len(str):
            is_match = True
        elif i < len(pattern) and j < len(str):
            p = pattern[i]
            if p in p2w:
                w = p2w[p]
                if w == str[j:j+len(w)]:  # Match pattern.
                    is_match = self.match(pattern, str, i + 1, j + len(w), w2p, p2w)
                # Else return false.
            else:
                for k in xrange(j, len(str)):  # Try any possible word
                    w = str[j:k+1]
                    if w not in w2p:
                        # Build mapping. Space: O(n + c)
                        w2p[w], p2w[p] = p, w
                        is_match = self.match(pattern, str, i + 1, k + 1, w2p, p2w)
                        w2p.pop(w), p2w.pop(p)
                    if is_match:
                        break
        return is_match
```

logger-rate-limiter.py

```python
# logger-rate-limiter is not found.
# Time:  O(1), amortized
# Space: O(k), k is the max number of printed messages in last 10 seconds

import collections


class Logger(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__dq = collections.deque()
        self.__printed = set()

    def shouldPrintMessage(self, timestamp, message):
        """
        Returns true if the message should be printed in the given timestamp, otherwise returns false. The tim
        :type timestamp: int
        :type message: str
        :rtype: bool
        """
        while self.__dq and self.__dq[0][0] <= timestamp - 10:
            self.__printed.remove(self.__dq.popleft()[1])
        if message in self.__printed:
            return False
        self.__dq.append((timestamp, message))
        self.__printed.add(message)
        return True
```

day-of-the-week.py

```python
# day-of-the-week is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def dayOfTheWeek(self, day, month, year):
        """
        :type day: int
        :type month: int
        :type year: int
        :rtype: str
        """
        DAYS = ["Sunday", "Monday", "Tuesday", "Wednesday", \
                "Thursday", "Friday", "Saturday"]

        # Zeller Formula
        if month < 3:
            month += 12
            year -= 1
        c, y = divmod(year, 100)
        w = (c//4 - 2*c + y + y//4 + 13*(month+1)//5 + day - 1) % 7
        return DAYS[w]
```

bitwise-ors-of-subarrays.py

```python
# DESC
# A[i] | A[i+1] | ... | A[j]
# Note:
# Example 3:
# B = [A[i], A[i+1], ..., A[j]]
# Example 2:
# We have an array A of non-negative integers.
# Return the number of possible results.  (Results that occur more than once are o
# nly counted once in the final answer.)
# For every (contiguous) subarray B = [A[i], A[i+1], ..., A[j]] (with i <= j), we
# take the bitwise OR of all the elements in B, obtaining a result A[i] | A[i+1] |
#  ... | A[j].
# Example 1:

# NOTE
# 1 <= A.length <= 50000
# 0 <= A[i] <= 10^9

# EXAMPLE
# Input: [1,2,4]
# Output: 6
# Explanation:
# The possible results are 1, 2, 3, 4, 6, and 7.
# 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.
# Input: [0]
# Output: 1
# Explanation:
# There is only one possible result: 0.

# 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)
```

groups-of-special-equivalent-strings.py

```python
# DESC
# Return the number of groups of special-equivalent strings from A.
# Now, a group of special-equivalent strings from A is a non-empty subset of A suc
# h that:
# A move onto S consists of swapping any two even indexed characters of S, or any
# two odd indexed characters of S.
# For example, S = "zzxy" and T = "xyzz" are special-equivalent because we may mak
# e the moves "zzxy" -> "xzzy" -> "xyzz" that swap S[0] and S[2], then S[1] and S[
# 3].
# Example 1:
# Two strings S and T are special-equivalent if after any number of moves onto S,
# S == T.
# Example 2:
# Note:
# You are given an array A of strings.

# NOTE
# All A[i] have the same length.
# The group is the largest size possible (ie., there isn't a string S not in the g
# roup such that S is special equivalent to every string in the group)
# 1 <= A[i].length <= 20
# 1 <= A.length <= 1000
# Every pair of strings in the group are special equivalent, and;
# All A[i] consist of only lowercase letters.

# EXAMPLE
# Input: ["abc","acb","bac","bca","cab","cba"]
# Output: 3
# Input: ["abcd","cdab","cbad","xyzz","zzxy","zzyx"]
# Output: 3
# Explanation:
# One g
# roup is ["abcd", "cdab", "cbad"], since they are all pairwise special equivalent
# , and none of the other strings are all pairwise special equivalent to these.
#
# T
# he other two groups are ["xyzz", "zzxy"] and ["zzyx"].  Note that in particular,
#  "zzxy" is not special equivalent to "zzyx".

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

class Solution(object):
    def numSpecialEquivGroups(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        def count(word):
            result = [0]*52
            for i, letter in enumerate(word):
                result[ord(letter)-ord('a') + 26*(i%2)] += 1
            return tuple(result)

        return len({count(word) for word in A})
```

clone-binary-tree-with-random-pointer.py

```
# clone-binary-tree-with-random-pointer is not found.
# Time:  O(n)
# Space: O(h)

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


# Definition for NodeCopy.
class NodeCopy(object):
    def __init__(self, val=0, left=None, right=None, random=None):
        pass


class Solution(object):
    def copyRandomBinaryTree(self, root):
        """
        :type root: Node
        :rtype: NodeCopy
        """
        def iter_dfs(node, callback):
            result = None
            stk = [node]
            while stk:
                node = stk.pop()
                if not node:
                    continue
                left_node, copy = callback(node)
                if not result:
                    result = copy
                stk.append(node.right)
                stk.append(left_node)
            return result

        def merge(node):
            copy = NodeCopy(node.val)
            node.left, copy.left = copy, node.left
            return copy.left, copy

        def clone(node):
            copy = node.left
            node.left.random = node.random.left if node.random else None
            node.left.right = node.right.left if node.right else None
            return copy.left, copy

        def split(node):
            copy = node.left
            node.left, copy.left = copy.left, copy.left.left if copy.left else None
            return node.left, copy

        iter_dfs(root, merge)
        iter_dfs(root, clone)
        return iter_dfs(root, split)
```

```python
# Time:  O(n)
# Space: O(h)
class Solution_Recu(object):
    def copyRandomBinaryTree(self, root):
        """
        :type root: Node
        :rtype: NodeCopy
        """
        def dfs(node, callback):
            if not node:
                return None
            left_node, copy = callback(node)
            dfs(left_node, callback)
            dfs(node.right, callback)
            return copy

        def merge(node):
            copy = NodeCopy(node.val)
            node.left, copy.left = copy, node.left
            return copy.left, copy

        def clone(node):
            copy = node.left
            node.left.random = node.random.left if node.random else None
            node.left.right = node.right.left if node.right else None
            return copy.left, copy

        def split(node):
            copy = node.left
            node.left, copy.left = copy.left, copy.left.left if copy.left else None
            return node.left, copy

        dfs(root, merge)
        dfs(root, clone)
        return dfs(root, split)


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


class Solution2(object):
    def copyRandomBinaryTree(self, root):
        """
        :type root: Node
        :rtype: NodeCopy
        """
        lookup = collections.defaultdict(lambda: NodeCopy())
        lookup[None] = None
        stk = [root]
        while stk:
            node = stk.pop()
            if not node:
                continue
            lookup[node].val = node.val
            lookup[node].left = lookup[node.left]
            lookup[node].right = lookup[node.right]
```

```
            lookup[node].random = lookup[node.random]
            stk.append(node.right)
            stk.append(node.left)
        return lookup[root]


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


class Solution2_Recu(object):
    def copyRandomBinaryTree(self, root):
        """
        :type root: Node
        :rtype: NodeCopy
        """
        def dfs(node, lookup):
            if not node:
                return
            lookup[node].val = node.val
            lookup[node].left = lookup[node.left]
            lookup[node].right = lookup[node.right]
            lookup[node].random = lookup[node.random]
            dfs(node.left, lookup)
            dfs(node.right, lookup)

        lookup = collections.defaultdict(lambda: NodeCopy())
        lookup[None] = None
        dfs(root, lookup)
        return lookup[root]
```

shortest-distance-to-target-color.py

```
# shortest-distance-to-target-color is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def shortestDistanceColor(self, colors, queries):
        """
        :type colors: List[int]
        :type queries: List[List[int]]
        :rtype: List[int]
        """
        dp = [[-1 for _ in xrange(len(colors))] for _ in xrange(3)]
        dp[colors[0]-1][0] = 0
        for i in xrange(1, len(colors)):
            for color in xrange(3):
                dp[color][i] = dp[color][i-1]
            dp[colors[i]-1][i] = i

        dp[colors[len(colors)-1]-1][len(colors)-1] = len(colors)-1
        for i in reversed(xrange(len(colors)-1)):
            for color in xrange(3):
                if dp[color][i+1] == -1:
                    continue
                if dp[color][i] == -1 or \
                   abs(dp[color][i+1]-i) < abs(dp[color][i]-i):
                    dp[color][i] = dp[color][i+1]
            dp[colors[i]-1][i] = i

        return [abs(dp[color-1][i]-i) if dp[color-1][i] != -1 else -1 \
                    for i, color in queries]
```

can-make-arithmetic-progression-from-sequence.py

```python
# can-make-arithmetic-progression-from-sequence is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canMakeArithmeticProgression(self, arr):
        """
        :type arr: List[int]
        :rtype: bool
        """
        m = min(arr)
        d = (max(arr)-m)//(len(arr)-1)
        if not d:
            return True
        i = 0
        while i < len(arr):
            if arr[i] == m+i*d:
                i += 1
            else:
                j, r = divmod(arr[i]-m, d)
                if r or j >= len(arr) or arr[i] == arr[j]:
                    return False
                arr[i], arr[j] = arr[j], arr[i]
        return True
```

swap-adjacent-in-lr-string.py

```
# DESC
# Example:
# In a string composed of 'L', 'R', and 'X' characters, like "RXXLRXRXL", a move c
# onsists of either replacing one occurrence of "XL" with "LX", or replacing one o
# ccurrence of "RX" with "XR". Given the starting string start and the ending stri
# ng end, return True if and only if there exists a sequence of moves to transform
#  one string to the other.
# Constraints:

# NOTE
# 1 <= len(start) == len(end) <= 10000.
# Both start and end will only consist of characters in {'L', 'R', 'X'}.

# EXAMPLE
# Input: start = "RXXLRXRXL", end = "XRLXXRRLX"
# Output: True
# Explanation:
# We can t
# ransform start to end following these steps:
# RXXLRXRXL ->
# XRXLRXRXL ->
# XRLXRXRXL
#  ->
# XRLXXRRXL ->
# XRLXXRRLX

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canTransform(self, start, end):
        """
        :type start: str
        :type end: str
        :rtype: bool
        """
        N = len(start)
        i, j = 0, 0
        while i < N and j < N:
            while i < N and start[i] == 'X':
                i += 1
            while j < N and end[j] == 'X':
                j += 1
            if (i < N) != (j < N):
                return False
            elif i < N and j < N:
                if start[i] != end[j] or \
                   (start[i] == 'L' and i < j) or \
                   (start[i] == 'R' and i > j):
                    return False
                i += 1
                j += 1
        return True
```

maximum-equal-frequency.py

```python
# maximum-equal-frequenc is not found.
# Given an array nums of positive integers, return the longest possible length of an array prefix of nums, suc
#
# If after removing one element there are no remaining elements, it's still considered that every appeared num
#
#
#
# Example 1:
#
# Input: nums = [2,2,1,1,5,3,3,5]
# Output: 7
# Explanation: For the subarray [2,2,1,1,5,3,3] of length 7, if we remove nums[4]=5, we will get [2,2,1,1,3,3]
# Example 2:
#
# Input: nums = [1,1,1,2,2,2,3,3,3,4,4,4,5]
# Output: 13
# Example 3:
#
# Input: nums = [1,1,1,2,2,2]
# Output: 5
# Example 4:
#
# Input: nums = [10,2,8,9,3,8,1,5,2,3,7,6]
# Output: 8


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

import collections


class Solution(object):
    def maxEqualFreq(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        count = collections.Counter()
        freq = [0 for _ in xrange(len(nums)+1)]
        for i, n in enumerate(nums, 1):
            freq[count[n]] -= 1
            freq[count[n]+1] += 1
            count[n] += 1
            c = count[n]
            if freq[c]*c == i and i < len(nums):
                result = i+1
            remain = i-freq[c]*c
            if freq[remain] == 1 and remain in [1, c+1]:
                result = i
        return result

    def s2(self, nums):
        def maxEqualFreq(self, nums: List[int]) -> int:
            cnt, freq, maxF, res = collections.defaultdict(int), collections.defaultdict(int), 0, 0
            for i, num in enumerate(nums):
                cnt[num] += 1
```

```
        freq[cnt[num] - 1] -= 1
        freq[cnt[num]] += 1
        maxF = max(maxF, cnt[num])
        if maxF * freq[maxF] == i or (maxF - 1) * (freq[maxF - 1] + 1) == i or maxF == 1:
            res = i + 1
    return res```
```

##check-completeness-of-a-binary-tree.py
```python
# DESC
# Example 1:
# Example 2:
# Note:
# Definition of a complete binary tree from Wikipedia:
#
# In a complete binary tree
# every level, except possibly the last, is completely filled, and all nodes in th
# e last level are as far left as possible. It can have between 1 and 2h nodes inc
# lusive at the last level h.
# Given a binary tree, determine if it is a complete binary tree.

# NOTE
# The tree will have between 1 and 100 nodes.

# EXAMPLE
# Input: [1,2,3,4,5,6]
# Output: true
# Explanation: Every level before the last is fu
# ll (ie. levels with node-values {1} and {2, 3}), and all nodes in the last level
#  ({4, 5, 6}) are as far left as possible.
# 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
```

repeated-string-match.py

```python
# DESC
# For example, with A = "abcd" and B = "cdabcdab".
# Return 3, because by repeating A three times ("abcdabcdabcd"), B is a substring
# of it; and B is not a substring of A repeated two times ("abcdabcd").
# Note:
#
# The length of A and B will be between 1 and 10000.
# Given two strings A and B, find the minimum number of times A has to be repeated
#  such that B is a substring of it. If no such solution, return -1.

# NOTE
#

# EXAMPLE
#

# Time:  O(n + m)
# Space: O(1)

class Solution(object):
    def repeatedStringMatch(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: int
        """
        def check(index):
            return all(A[(i+index) % len(A)] == c
                       for i, c in enumerate(B))

        M, p = 10**9+7, 113
        p_inv = pow(p, M-2, M)
        q = (len(B)+len(A)-1) // len(A)

        b_hash, power = 0, 1
        for c in B:
            b_hash += power * ord(c)
            b_hash %= M
            power = (power*p) % M

        a_hash, power = 0, 1
        for i in xrange(len(B)):
            a_hash += power * ord(A[i%len(A)])
            a_hash %= M
            power = (power*p) % M

        if a_hash == b_hash and check(0): return q

        power = (power*p_inv) % M
        for i in xrange(len(B), (q+1)*len(A)):
            a_hash = (a_hash-ord(A[(i-len(B))%len(A)])) * p_inv
            a_hash += power * ord(A[i%len(A)])
            a_hash %= M
            if a_hash == b_hash and check(i-len(B)+1):
                return q if i < q*len(A) else q+1

        return -1
```

global-and-local-inversions.py

```
# DESC
# i < j
# Return true if and only if the number of global inversions is equal to the numbe
# r of local inversions.
# Example 1:
# Example 2:
# The number of local inversions is the number of i with 0 <= i < N and A[i] > A[i+1].
# The number of (global) inversions is the number of i < j with 0 <= i < j < N and
#  A[i] > A[j].
# We have some permutation A of [0, 1, ..., N - 1], where N is the length of A.
# Note:

# NOTE
# A will be a permutation of [0, 1, ..., A.length - 1].
# The time limit for this problem has been reduced.
# A will have length in range [1, 5000].

# EXAMPLE
# Input: A = [1,2,0]
# Output: false
# Explanation: There are 2 global inversions, and
#  1 local inversion.
# Input: A = [1,0,2]
# Output: true
# Explanation: There is 1 global inversion, and 1
# local inversion.

# We have some permutation A of [0, 1, ..., N - 1], where N is the length of A.
#
# The number of (global) inversions is the number of i < j with 0 <= i < j < N and A[i] > A[j].
#
# The number of local inversions is the number of i with 0 <= i < N and A[i] > A[i+1].
#
# Return true if and only if the number of global inversions is equal to the number of local inversions.
#
# Example 1:
#
# Input: A = [1,0,2]
# Output: true
# Explanation: There is 1 global inversion, and 1 local inversion.
# Example 2:
#
# Input: A = [1,2,0]
# Output: false
# Explanation: There are 2 global inversions, and 1 local inversion.
# Note:
#
# A will be a permutation of [0, 1, ..., A.length - 1].
# A will have length in range [1, 5000].
# The time limit for this problem has been reduced.

# Time:  O(n)
# Space: O(1)

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

```
        :rtype: bool
        """
        return all(abs(v-i) <= 1 for i,v in enumerate(A))
```

n-ary-tree-preorder-traversal.py

```
# DESC
# Constraints:
# Given an n-ary tree, return the preorder traversal of its nodes' values.
# Example 1:
# Example 2:
# Nary-Tree input serialization is represented in their level order traversal, eac
# h group of children is separated by the null value (See examples).
# Recursive solution is trivial, could you do it iteratively?
# Follow up:

# NOTE
# The height of the n-ary tree is less than or equal to 1000
# The total number of nodes is between [0, 10^4]

# EXAMPLE
# Input: root = [1,null,3,2,4,null,5,6]
# Output: [1,3,5,6,2,4]
# Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,1
# 2,null,13,null,null,14]
# Output: [1,2,3,6,7,11,14,4,8,12,5,9,13,10]

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

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


class Solution(object):
    def preorder(self, root):
        """
        :type root: Node
        :rtype: List[int]
        """
        if not root:
            return []
        result, stack = [], [root]
        while stack:
            node = stack.pop()
            result.append(node.val)
            for child in reversed(node.children):
                if child:
                    stack.append(child)
        return result


class Solution2(object):
    def preorder(self, root):
        """
        :type root: Node
        :rtype: List[int]
        """
        def dfs(root, result):
            result.append(root.val)
            for child in root.children:
                if child:
```

```
            dfs(child, result)

result = []
if root:
    dfs(root, result)
return result
```

increasing-subsequences.py

```
# DESC
# Given an integer array, your task is to find all the different possible increasi
# ng subsequences of the given array, and the length of an increasing subsequence
# should be at least 2.
# Constraints:
# Example:

# NOTE
# The given array may contain duplicates, and two equal integers should also be co
# nsidered as a special case of increasing sequence.
# The range of integer in the given array is [-100,100].
# The length of the given array will not exceed 15.

# 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]]

# 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
```

flipping-an-image.py

```
# DESC
# Notes:
# To flip an image horizontally means that each row of the image is reversed.  For
#  example, flipping [1, 1, 0] horizontally results in [0, 1, 1].
# Given a binary matrix A, we want to flip the image horizontally, then invert it,
#  and return the resulting image.
# To invert an image means that each 0 is replaced by 1, and each 1 is replaced by
#  0. For example, inverting [0, 1, 1] results in [1, 0, 0].
# Example 2:
# [1, 1, 0]
# Example 1:

# NOTE
# 1 <= A.length = A[0].length <= 20
# 0 <= A[i][j] <= 1

# EXAMPLE
# Input: [[1,1,0],[1,0,1],[0,0,0]]
# Output: [[1,0,0],[0,1,0],[1,1,1]]
# Explanation:
# First reverse each row: [[0,1,1],[1,0,1],[0,0,0]].
# Then, invert the image: [[1,0
# ,0],[0,1,0],[1,1,1]]
# Input: [[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]
# Output: [[1,1,0,0],[0,1,1,0],[0
# ,0,0,1],[1,0,1,0]]
# Explanation: First reverse each row: [[0,0,1,1],[1,0,0,1],[1,
# 1,1,0],[0,1,0,1]].
# Then invert the image: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,
# 0]]

# Time:  O(n^2)
# Space: O(1)


class Solution(object):
    def flipAndInvertImage(self, A):
        """
        :type A: List[List[int]]
        :rtype: List[List[int]]
        """
        for row in A:
            for i in xrange((len(row)+1) // 2):
                row[i], row[~i] = row[~i] ^ 1, row[i] ^ 1
        return A
```

binary-tree-level-order-traversal-ii.py

```
# DESC
# [3,9,20,null,null,15,7]
# return its bottom-up level order traversal as:
# Given a binary tree, return the bottom-up level order traversal of its nodes' va
# lues. (ie, from left to right, level by level from leaf to root).
# For example:
#
# Given binary tree [3,9,20,null,null,15,7],

# NOTE
#

# EXAMPLE
# [
#   [15,7],
#   [9,20],
#   [3]
# ]
# 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):
    def levelOrderBottom(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        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[::-1]
```

add-to-array-form-of-integer.py

```
# DESC
# Note
# Example 4:
# For a non-negative integer X, the array-form of X is an array of its digits in l
# eft to right order.  For example, if X = 1231, then the array form is [1,2,3,1].
# Given the array-form A of a non-negative integer X, return the array-form of the
#  integer X+K.
# Example 2:
# Example 1:
# Example 3:

# NOTE
# 0 <= A[i] <= 9
# If A.length > 1, then A[0] != 0
# 1 <= A.length <= 10000
# 0 <= K <= 10000

# EXAMPLE
# Input: A = [9,9,9,9,9,9,9,9,9,9], K = 1
# Output: [1,0,0,0,0,0,0,0,0,0,0]
# Explanat
# ion: 9999999999 + 1 = 10000000000
# Input: A = [2,7,4], K = 181
# Output: [4,5,5]
# Explanation: 274 + 181 = 455
# Input: A = [1,2,0,0], K = 34
# Output: [1,2,3,4]
# Explanation: 1200 + 34 = 1234
# Input: A = [2,1,5], K = 806
# Output: [1,0,2,1]
# Explanation: 215 + 806 = 1021

# Time:  O(n + logk)
# Space: O(1)

class Solution(object):
    def addToArrayForm(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: List[int]
        """
        A.reverse()
        carry, i = K, 0
        A[i] += carry
        carry, A[i] = divmod(A[i], 10)
        while carry:
            i += 1
            if i < len(A):
                A[i] += carry
            else:
                A.append(carry)
            carry, A[i] = divmod(A[i], 10)
        A.reverse()
        return A
```

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

```
# DESC
# Example 2:
# Note:
# Example 1:
# Given a binary string S (a string consisting only of '0' and '1's) and a positiv
# e integer N, return true if and only if for every integer X from 1 to N, the bin
# ary representation of X is a substring of S.

# NOTE
# 1 <= N <= 10^9
# 1 <= S.length <= 1000

# EXAMPLE
# Input: S = "0110", N = 3
# Output: true
# Input: S = "0110", N = 4
# Output: false

# 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

```
# DESC
# Note:
# Example 1:
# Example 3:
# Example 2:
# Given a string containing only three types of characters: '(', ')' and '*', writ
# e a function to check whether this string is valid. We define the validity of a
# string by these rules:

# NOTE
# An empty string is also valid.
# The string size will be in the range [1, 100].
# 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 parenthe
# sis '(' or an empty string.

# EXAMPLE
# Input: "()"
# Output: True
# Input: "(*))"
# Output: True
# Input: "(*)"
# Output: True

# 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
```

ambiguous-coordinates.py

```
# DESC
# Note:
# Our original representation never had extraneous zeroes, so we never started wit
# h numbers like "00", "0.0", "0.00", "1.0", "001", "00.01", or any other number t
# hat can be represented with less digits.  Also, a decimal point within a number
# never occurs without at least one digit occuring before it, so we never started
# with numbers like ".1".
# The final answer list can be returned in any order.  Also note that all coordina
# tes in the final answer have exactly one space between them (occurring after the
#  comma.)
# We had some 2-dimensional coordinates, like "(1, 3)" or "(2, 0.5)".  Then, we re
# moved all commas, decimal points, and spaces, and ended up with the string S.  R
# eturn a list of strings representing all possibilities for what our original coo
# rdinates could have been.

# NOTE
# 4 <= S.length <= 12.
# S[0] = "(", S[S.length - 1] = ")", and the other elements in S are digits.

# EXAMPLE
# Example 1:
# Input: "(123)"
# Output: ["(1, 23)", "(12, 3)", "(1.2, 3)", "(1, 2.3)"]
# Example 3:
# Input: "(0123)"
# Output: ["(0, 123)", "(0, 12.3)", "(0, 1.23)", "(0.1,
#  23)", "(0.1, 2.3)", "(0.12, 3)"]
# Example 2:
# Input: "(00011)"
# Output:  ["(0.001, 1)", "(0, 0.011)"]
# Explanation:
#
# 0.0, 00, 0001 or 00.01 are not allowed.
# Example 4:
# Input: "(100)"
# Output: [(10, 0)]
# Explanation:
# 1.0 is not allowed.

# Time:  O(n^4)
# Space: O(n)

import itertools


class Solution(object):
    def ambiguousCoordinates(self, S):
        """
        :type S: str
        :rtype: List[str]
        """
        def make(S, i, n):
            for d in xrange(1, n+1):
                left = S[i:i+d]
                right = S[i+d:i+n]
                if ((not left.startswith('0') or left == '0')
                        and (not right.endswith('0'))):
                    yield "".join([left, '.' if right else '', right])
```

```
        return ["({}, {})".format(*cand)
                for i in xrange(1, len(S)-2)
                for cand in itertools.product(make(S, 1, i),
                                              make(S, i+1, len(S)-2-i))]
```

total-hamming-distance.py

```
# DESC
# Note:
# The Hamming distance between two integers is the number of positions at which th
# e corresponding bits are different.
# Example:
# Now your job is to find the total Hamming distance between all pairs of the give
# n numbers.

# NOTE
# Elements of the given array are in the range of 0  to 10^9
# Length of the array will not exceed 10^4.

# 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) + HammingD
# istance(14, 2) = 2 + 2 + 2 = 6.

# 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
```

sparse-matrix-multiplication.py

```
# sparse-matrix-multiplication is not found.
# Time:  O(m * n * l), A is m x n matrix, B is n x l matrix
# Space: O(m * l)

class Solution(object):
    def multiply(self, A, B):
        """
        :type A: List[List[int]]
        :type B: List[List[int]]
        :rtype: List[List[int]]
        """
        m, n, l = len(A), len(A[0]), len(B[0])
        res = [[0 for _ in xrange(l)] for _ in xrange(m)]
        for i in xrange(m):
            for k in xrange(n):
                if A[i][k]:
                    for j in xrange(l):
                        res[i][j] += A[i][k] * B[k][j]
        return res
```

serialize-and-deserialize-binary-tree.py

```
# DESC
# Design an algorithm to serialize and deserialize a binary tree. There is no rest
# riction on how your serialization/deserialization algorithm should work. You jus
# t need to ensure that a binary tree can be serialized to a string and this strin
# g can be deserialized to the original tree structure.
# Example:
# Note: Do not use class member/global/static variables to store states. Your seri
# alize and deserialize algorithms should be stateless.
# Serialization is the process of converting a data structure or object into a seq
# uence of bits so that it can be stored in a file or memory buffer, or transmitte
# d across a network connection link to be reconstructed later in the same or anot
# her computer environment.
# Clarification: The above format is the same as how LeetCode serializes a binary
# tree. You do not necessarily need to follow this format, so please be creative a
# nd come up with different approaches yourself.

# NOTE
#

# EXAMPLE
# You may serialize the following tree:
#
#     1
#    / \
#   2   3
#      / \
#     4   5
#
#
# as "[1,2,3,null,null,4,5]"

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

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):
            if not node:
                vals.append('#')
                return
            vals.append(str(node.val))
            serializeHelper(node.left)
            serializeHelper(node.right)
        vals = []
        serializeHelper(root)
```

```
        return ' '.join(vals)


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

        :type data: str
        :rtype: TreeNode
        """
        def deserializeHelper():
            val = next(vals)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = deserializeHelper()
            node.right = deserializeHelper()
            return node
        def isplit(source, sep):
            sepsize = len(sep)
            start = 0
            while True:
                idx = source.find(sep, start)
                if idx == -1:
                    yield source[start:]
                    return
                yield source[start:idx]
                start = idx + sepsize
        vals = iter(isplit(data, ' '))
        return deserializeHelper()


# time: O(n)
# space: O(n)

class Codec2(object):

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

        :type root: TreeNode
        :rtype: str
        """
        def gen_preorder(node):
            if not node:
                yield '#'
            else:
                yield str(node.val)
                for n in gen_preorder(node.left):
                    yield n
                for n in gen_preorder(node.right):
                    yield n

        return ' '.join(gen_preorder(root))

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

        :type data: str
        :rtype: TreeNode
        """
```

```
def builder(chunk_iter):
    val = next(chunk_iter)
    if val == '#':
        return None
    node = TreeNode(int(val))
    node.left = builder(chunk_iter)
    node.right = builder(chunk_iter)
    return node

# https://stackoverflow.com/a/42373311/568901
chunk_iter = iter(data.split())
return builder(chunk_iter)
```

reverse-string.py

```python
# DESC
# You may assume all the characters consist of printable ascii characters.
# Do not allocate extra space for another array, you must do this by modifying the
#  input array in-place with O(1) extra memory.
# Write a function that reverses a string. The input string is given as an array o
# f characters char[].
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: ["h","e","l","l","o"]
# Output: ["o","l","l","e","h"]
# Input: ["H","a","n","n","a","h"]
# Output: ["h","a","n","n","a","H"]

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

class Solution(object):
    def reverseString(self, s):
        """
        :type s: str
        :rtype: str
        """
        string = list(s)
        i, j = 0, len(string) - 1
        while i < j:
            string[i], string[j] = string[j], string[i]
            i += 1
            j -= 1
        return "".join(string)


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def reverseString(self, s):
        """
        :type s: str
        :rtype: str
        """
        return s[::-1]
```

serialize-and-deserialize-bst.py

```
# DESC
# 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.
# Note: Do not use class member/global/static variables to store states. Your seri
# alize and deserialize algorithms should be stateless.
# The encoded string should be as compact as possible.
# Serialization is the process of converting a data structure or object into a seq
# uence of bits so that it can be stored in a file or memory buffer, or transmitte
# d across a network connection link to be reconstructed later in the same or anot
# her computer environment.

# NOTE
#

# EXAMPLE
#

# 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()
        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)
```

queries-on-a-permutation-with-key.py

```python
# queries-on-a-permutation-with-ke is not found.
# Time:  O(nlogn)
# Space: O(n)

class BIT(object):  # Fenwick Tree, 1-indexed
    def __init__(self, n):
        self.__bit = [0] * n

    def add(self, i, val):
        while i < len(self.__bit):
            self.__bit[i] += val
            i += (i & -i)

    def sum(self, i):
        result = 0
        while i > 0:
            result += self.__bit[i]
            i -= (i & -i)
        return result


class Solution(object):
    def processQueries(self, queries, m):
        """
        :type queries: List[int]
        :type m: int
        :rtype: List[int]
        """
        bit = BIT(2*m+1)
        lookup = {}
        for i in xrange(1, m+1):
            bit.add(m+i, 1)
            lookup[i] = m+i
        result, curr = [], m
        for q in queries:
            i = lookup.pop(q)
            result.append(bit.sum(i-1))
            bit.add(i, -1)
            lookup[q] = curr
            bit.add(curr, 1)
            curr -= 1
        return result
```

display-table-of-food-orders-in-a-restaurant.py

```python
# display-table-of-food-orders-in-a-restaurant is not found.
# Time:  O(n + tlogt + flogf)
# Space: O(n)

import collections


class Solution(object):
    def displayTable(self, orders):
        """
        :type orders: List[List[str]]
        :rtype: List[List[str]]
        """
        table_count = collections.defaultdict(collections.Counter)
        for _, table, food in orders:
            table_count[int(table)][food] += 1
        foods = sorted({food for _, _, food in orders})
        result = [["Table"]]
        result[0].extend(foods)
        for table in sorted(table_count):
            result.append([str(table)])
            result[-1].extend(str(table_count[table][food]) for food in foods)
        return result
```

minimum-moves-to-equal-array-elements.py

```
# DESC
# Example:
# Given a non-empty integer array of size n, find the minimum number of moves requ
# ired to make all array elements equal, where a move is incrementing n - 1 elemen
# ts by 1.

# NOTE
#

# EXAMPLE
# Input:
# [1,2,3]
#
# Output:
# 3
#
# Explanation:
# Only three moves are needed (remember ea
# ch move increments two elements):
#
# [1,2,3]  =>  [2,3,3]  =>  [3,4,3]  =>  [4,4,4
# ]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minMoves(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return sum(nums) - len(nums) * min(nums)
```

product-of-array-except-self.py

```
# DESC
# Example:
# Note: Please solve it without division and in O(n).
# 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].
# 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.)
# Constraint: It's guaranteed that the product of the elements of any prefix or su
# ffix of the array (including the whole array) fits in a 32 bit integer.

# NOTE
#

# EXAMPLE
# Input:  [1,2,3,4]
# Output: [24,12,8,6]

# 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
```

walking-robot-simulation.py

```
# DESC
# Some of the grid squares are obstacles.
# Return the square of the maximum Euclidean distance that the robot will be from
# the origin.
# Example 2:
# Note:
# The i-th obstacle is at grid point (obstacles[i][0], obstacles[i][1])
# If the robot would try to move onto them, the robot stays on the previous grid s
# quare instead (but still continues following the rest of the route.)
# A robot on an infinite grid starts at point (0, 0) and faces north.  The robot c
# an receive one of three possible types of commands:
# Example 1:

# NOTE
# The answer is guaranteed to be less than 2 ^ 31.
# 0 <= obstacles.length <= 10000
# 0 <= commands.length <= 10000
# 1 <= x <= 9: move forward x units
# -2: turn left 90 degrees
# -30000 <= obstacle[i][1] <= 30000
# -30000 <= obstacle[i][0] <= 30000
# -1: turn right 90 degrees

# EXAMPLE
# Input: commands = [4,-1,3], obstacles = []
# Output: 25
# Explanation: robot will go
#  to (3, 4)
# Input: commands = [4,-1,4,-2,4], obstacles = [[2,4]]
# Output: 65
# Explanation: rob
# ot will be stuck at (1, 4) before turning left and going to (1, 8)

# A robot on an infinite grid starts at point (0, 0) and faces north.  The robot can receive one of three poss
#
# -2: turn left 90 degrees
# -1: turn right 90 degrees
# 1 <= x <= 9: move forward x units
# Some of the grid squares are obstacles.
#
# The i-th obstacle is at grid point (obstacles[i][0], obstacles[i][1])
#
# If the robot would try to move onto them, the robot stays on the previous grid square instead (but still con
#
# Return the square of the maximum Euclidean distance that the robot will be from the origin.
#
#
#
# Example 1:
#
# Input: commands = [4,-1,3], obstacles = []
# Output: 25
# Explanation: robot will go to (3, 4)
# Example 2:
#
# Input: commands = [4,-1,4,-2,4], obstacles = [[2,4]]
# Output: 65
# Explanation: robot will be stuck at (1, 4) before turning left and going to (1, 8)
```

```
#
#
# Note:
#
# 0 <= commands.length <= 10000
# 0 <= obstacles.length <= 10000
# -30000 <= obstacle[i][0] <= 30000
# -30000 <= obstacle[i][1] <= 30000
# The answer is guaranteed to be less than 2 ^ 31.

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

class Solution(object):
    def robotSim(self, commands, obstacles):
        """
        :type commands: List[int]
        :type obstacles: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        x, y, i = 0, 0, 0
        lookup = set(map(tuple, obstacles))
        result = 0
        for cmd in commands:
            if cmd == -2:
                i = (i-1) % 4
            elif cmd == -1:
                i = (i+1) % 4
            else:
                for k in xrange(cmd):
                    if (x+directions[i][0], y+directions[i][1]) not in lookup:
                        x += directions[i][0]
                        y += directions[i][1]
                        result = max(result, x*x + y*y)
        return result
```

push-dominoes.py

```
# DESC
# After each second, each domino that is falling to the left pushes the adjacent d
# omino on the left.
# Example 2:
# When a vertical domino has dominoes falling on it from both sides, it stays stil
# l due to the balance of the forces.
# Similarly, the dominoes falling to the right push their adjacent dominoes standi
# ng on the right.
# Note:
# Return a string representing the final state.
# Example 1:
# For the purposes of this question, we will consider that a falling domino expend
# s no additional force to a falling or already fallen domino.
# In the beginning, we simultaneously push some of the dominoes either to the left
#  or to the right.
# Given a string "S" representing the initial state. S[i] = 'L', if the i-th domin
# o has been pushed to the left; S[i] = 'R', if the i-th domino has been pushed to
#  the right; S[i] = '.', if the i-th domino has not been pushed.
# There are N dominoes in a line, and we place each domino vertically upright.

# NOTE
# 0 <= N <= 10^5
# String dominoes contains only 'L', 'R' and '.'

# EXAMPLE
# Input: ".L.R...LR..L.."
# Output: "LL.RR.LLRRLL.."
# Input: "RR.L"
# Output: "RR.L"
# Explanation: The first domino expends no additional
#  force on the second domino.

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


class Solution(object):
    def pushDominoes(self, dominoes):
        """
        :type dominoes: str
        :rtype: str
        """
        force = [0]*len(dominoes)

        f = 0
        for i in xrange(len(dominoes)):
            if dominoes[i] == 'R':
                f = len(dominoes)
            elif dominoes[i] == 'L':
                f = 0
            else:
                f = max(f-1, 0)
            force[i] += f

        f = 0
        for i in reversed(xrange(len(dominoes))):
            if dominoes[i] == 'L':
                f = len(dominoes)
```

```
        elif dominoes[i] == 'R':
            f = 0
        else:
            f = max(f-1, 0)
        force[i] -= f

    return "".join('.' if f == 0 else 'R' if f > 0 else 'L'
                   for f in force)
```

index-pairs-of-a-string.py

```python
# index-pairs-of-a-string is not found.
# Time:  O(n + m + z), n is the total size of patterns
#                    , m is the total size of query string
#                    , z is the number of all matched strings
# Space: O(t), t is the total size of ac automata trie

import collections


class AhoNode(object):
    def __init__(self):
        self.children = collections.defaultdict(AhoNode)
        self.indices = []
        self.suffix = None
        self.output = None


class AhoTrie(object):

    def step(self, letter):
        while self.__node and letter not in self.__node.children:
            self.__node = self.__node.suffix
        self.__node = self.__node.children[letter] if self.__node else self.__root
        return self.__get_ac_node_outputs(self.__node)

    def __init__(self, patterns):
        self.__root = self.__create_ac_trie(patterns)
        self.__node = self.__create_ac_suffix_and_output_links(self.__root)

    def __create_ac_trie(self, patterns):  # Time:  O(n), Space: O(t)
        root = AhoNode()
        for i, pattern in enumerate(patterns):
            node = root
            for c in pattern:
                node = node.children[c]
            node.indices.append(i)
        return root

    def __create_ac_suffix_and_output_links(self, root):  # Time:  O(n), Space: O(t)
        queue = collections.deque()
        for node in root.children.itervalues():
            queue.append(node)
            node.suffix = root

        while queue:
            node = queue.popleft()
            for c, child in node.children.iteritems():
                queue.append(child)
                suffix = node.suffix
                while suffix and c not in suffix.children:
                    suffix = suffix.suffix
                child.suffix = suffix.children[c] if suffix else root
                child.output = child.suffix if child.suffix.indices else child.suffix.output

        return root

    def __get_ac_node_outputs(self, node):  # Time:  O(z)
        result = []
```

```
        for i in node.indices:
            result.append(i)
        output = node.output
        while output:
            for i in output.indices:
                result.append(i)
            output = output.output
        return result


class Solution(object):
    def indexPairs(self, text, words):
        """
        :type text: str
        :type words: List[str]
        :rtype: List[List[int]]
        """
        result = []
        reversed_words = [w[::-1] for w in words]
        trie = AhoTrie(reversed_words)
        for i in reversed(xrange(len(text))):
            for j in trie.step(text[i]):
                result.append([i, i+len(reversed_words[j])-1])
        result.reverse()
        return result
```

symmetric-tree.py

```
# DESC
# Given a binary tree, check whether it is a mirror of itself (ie, symmetric aroun
# d its center).
# [1,2,2,3,4,4,3]
# Follow up: Solve it both recursively and iteratively.
# But the following [1,2,2,null,3,null,3] is not:
# For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

# NOTE
#

# EXAMPLE
# 1
#    / \
#   2   2
#    \   \
#    3    3
# 1
#    / \
#   2   2
#  / \ / \
# 3  4 4  3

# Time:  O(n)
# Space: O(h), h is height of binary tree
# Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

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

# Iterative solution
class Solution(object):
    # @param root, a tree node
    # @return a boolean
    def isSymmetric(self, root):
        if root is None:
            return True
        stack = []
        stack.append(root.left)
        stack.append(root.right)

        while stack:
            p, q = stack.pop(), stack.pop()

            if p is None and q is None:
                continue

            if p is None or q is None or p.val != q.val:
                return False

            stack.append(p.left)
            stack.append(q.right)

            stack.append(p.right)
            stack.append(q.left)
```

```
        return True

# Recursive solution
class Solution2(object):
    # @param root, a tree node
    # @return a boolean
    def isSymmetric(self, root):
        if root is None:
            return True

        return self.isSymmetricRecu(root.left, root.right)

    def isSymmetricRecu(self, left, right):
        if left is None and right is None:
            return True
        if left is None or right is None or left.val != right.val:
            return False
        return self.isSymmetricRecu(left.left, right.right) and self.isSymmetricRecu(left.right, right.left)
```

prime-palindrome.py

```
# DESC
# Example 1:
# Recall that a number is prime if it's only divisors are 1 and itself, and it is
# greater than 1.
# Find the smallest prime palindrome greater than or equal to N.
# For example, 12321 is a palindrome.
# Example 3:
# Example 2:
# Note:
# For example, 2,3,5,7,11 and 13 are primes.
# Recall that a number is a palindrome if it reads the same from left to right as
# it does from right to left.

# NOTE
# 1 <= N <= 10^8
# The answer is guaranteed to exist and be less than 2 * 10^8.

# EXAMPLE
# Input: 6
# Output: 7
# Input: 13
# Output: 101
# Input: 8
# Output: 11

# Time:  O(n^(1/2) * (logn + n^(1/2)))
# Space: O(logn)


class Solution(object):
    def primePalindrome(self, N):
        """
        :type N: int
        :rtype: int
        """
        def is_prime(n):
            if n < 2 or n % 2 == 0:
                return n == 2
            return all(n % d for d in xrange(3, int(n**.5) + 1, 2))

        if 8 <= N <= 11:
            return 11
        for i in xrange(10**(len(str(N))//2), 10**5):
            j = int(str(i) + str(i)[-2::-1])
            if j >= N and is_prime(j):
                return j
```

fibonacci-number.py

```
# DESC
# Example 1:
# 0  N  30.
# Note:
# Example 3:
# Given N, calculate F(N).
# Example 2:
# The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonac
# ci sequence, such that each number is the sum of the two preceding ones, startin
# g from 0 and 1. That is,

# NOTE
#

# EXAMPLE
# Input: 3
# Output: 2
# Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.
# F(0) = 0,   F(1) = 1
# F(N) = F(N - 1) + F(N - 2), for N > 1.
# Input: 4
# Output: 3
# Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.
# Input: 2
# Output: 1
# Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def fib(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)) \
                     for col in ZB] for row in A]

        T = [[1, 1],
             [1, 0]]
        return matrix_mult([[1, 0]], matrix_expo(T, N))[0][1]  # [a1, a0] * T^N
```

```python
# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def fib(self, N):
        """
        :type N: int
        :rtype: int
        """
        prev, current = 0, 1
        for i in xrange(N):
            prev, current = current, prev + current,
        return prev
```

string-compression-ii.py

```
# string-compression-ii is not found.
# Run-length encoding is a string compression method that works by replacing consecutive identical characters
#
# Notice that in this problem, we are not adding '1' after single characters.
#
# Given a string s and an integer k. You need to delete at most k characters from s such that the run-length e
#
# Find the minimum length of the run-length encoded version of s after deleting at most k characters.
#
#
#
# Example 1:
#
# Input: s = "aaabcccd", k = 2
# Output: 4
# Explanation: Compressing s without deleting anything will give us "a3bc3d" of length 6. Deleting any of the
# Example 2:
#
# Input: s = "aabbaa", k = 2
# Output: 2
# Explanation: If we delete both 'b' characters, the resulting compressed string would be "a4" of length 2.
# Example 3:
#
# Input: s = "aaaaaaaaaaa", k = 0
# Output: 3
# Explanation: Since k is zero, we cannot delete anything. The compressed string is "a11" of length 3.
#
#
# Constraints:
#
# 1 <= s.length <= 100
# 0 <= k <= s.length
# s contains only lowercase English letters.


# Time:  O(n^2 * k)
# Space: O(n * k)

class Solution(object):
    def getLengthOfOptimalCompression(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        def length(cnt):
            l = 2 if cnt >= 2 else 1
            while cnt >= 10:
                l += 1
                cnt //= 10
            return l

        dp = [[len(s)]*(k+1) for _ in xrange(len(s)+1)]
        dp[0][0] = 0
        for i in xrange(1, len(s)+1):
            for j in xrange(k+1):
                if i-1 >= 0 and j-1 >= 0:
                    dp[i][j] = min(dp[i][j], dp[i-1][j-1])
```

```
        keep = delete = 0
        for m in xrange(i, len(s)+1):
            if s[i-1] == s[m-1]:
                keep += 1
            else:
                delete += 1
            if j+delete <= k:
                dp[m][j+delete] = min(dp[m][j+delete], dp[i-1][j]+length(keep));
return dp[len(s)][k]
```

queue-reconstruction-by-height.py

```
# DESC
# Note:
#
# The number of people is less than 1,100.
# Example
# Suppose you have a random list of people standing in a queue. Each person is des
# cribed by a pair of integers (h, k), where h is the height of the person and k i
# s 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
#

# 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
```

decode-ways-ii.py

```
# DESC
# Note:
# Also, since the answer may be very large, you should return the output mod 109 + 7.
# A message containing letters from A-Z is being encoded to numbers using the foll
# owing mapping way:
# Example 2:
# Beyond that, now the encoded string can also contain the character '*', which ca
# n be treated as one of the numbers from 1 to 9.
# Example 1:
# Given the encoded message containing digits and the character '*', return the to
# tal number of ways to decode it.

# NOTE
# The input string will only contain the character '*' and digits '0' - '9'.
# The length of the input string will fit in range [1, 105].

# EXAMPLE
# 'A' -> 1
# 'B' -> 2
# ...
# 'Z' -> 26
# Input: "1*"
# Output: 9 + 9 = 18
# Input: "*"
# Output: 9
# Explanation: The encoded message can be decoded to the stri
# ng: "A", "B", "C", "D", "E", "F", "G", "H", "I".

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numDecodings(self, s):
        """
        :type s: str
        :rtype: int
        """
        M, W = 1000000007, 3
        dp = [0] * W
        dp[0] = 1
        dp[1] = 9 if s[0] == '*' else dp[0] if s[0] != '0' else 0
        for i in xrange(1, len(s)):
            if s[i] == '*':
                dp[(i + 1) % W] = 9 * dp[i % W]
                if s[i - 1] == '1':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + 9 * dp[(i - 1) % W]) % M
                elif s[i - 1] == '2':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + 6 * dp[(i - 1) % W]) % M
                elif s[i - 1] == '*':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + 15 * dp[(i - 1) % W]) % M
            else:
                dp[(i + 1) % W] = dp[i % W] if s[i] != '0' else 0
                if s[i - 1] == '1':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + dp[(i - 1) % W]) % M
                elif s[i - 1] == '2' and s[i] <= '6':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + dp[(i - 1) % W]) % M
                elif s[i - 1] == '*':
                    dp[(i + 1) % W] = (dp[(i + 1) % W] + (2 if s[i] <= '6' else 1) * dp[(i - 1) % W]) % M
```

```
    return dp[len(s) % W]
```

number-of-subsequences-that-satisfy-the-given-sum-condition.py

```python
# number-of-subsequences-that-satisfy-the-given-sum-condition is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def numSubseq(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        MOD = 10**9 + 7
        nums.sort()
        result = 0
        left, right = 0, len(nums)-1
        while left <= right:
            if nums[left]+nums[right] > target:
                right -= 1
            else:
                result = (result+pow(2, right-left, MOD))%MOD
                left += 1
        return result
```

valid-boomerang.py

```
# DESC
# Example 2:
# A boomerang is a set of 3 points that are all distinct and not in a straight line.
# Given a list of three points in the plane, return whether these points are a boo
# merang.
# Example 1:
# Note:

# NOTE
# points[i].length == 2
# 0 <= points[i][j] <= 100
# points.length == 3

# EXAMPLE
# Input: [[1,1],[2,2],[3,3]]
# Output: false
# Input: [[1,1],[2,3],[3,2]]
# Output: true

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def isBoomerang(self, points):
        """
        :type points: List[List[int]]
        :rtype: bool
        """
        return (points[0][0] - points[1][0]) * (points[0][1] - points[2][1]) - \
               (points[0][0] - points[2][0]) * (points[0][1] - points[1][1]) != 0
```

k-inverse-pairs-array.py

```python
# k-inverse-pairs-arra is not found.
# Time:  O(n * k)
# Space: O(k)

class Solution(object):
    def kInversePairs(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        M = 1000000007
        dp = [[0]*(k+1) for _ in xrange(2)]
        dp[0][0] = 1
        for i in xrange(1, n+1):
            dp[i%2] = [0]*(k+1)
            dp[i%2][0] = 1
            for j in xrange(1, k+1):
                dp[i%2][j] = (dp[i%2][j-1] + dp[(i-1)%2][j]) % M
                if j-i >= 0:
                    dp[i%2][j] = (dp[i%2][j] - dp[(i-1)%2][j-i]) % M
        return dp[n%2][k]
```

convert-integer-to-the-sum-of-two-no-zero-integers.py

```python
# convert-integer-to-the-sum-of-two-no-zero-integers is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def getNoZeroIntegers(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        a, curr, base = 0, n, 1
        while curr:
            if curr % 10 == 0 or (curr % 10 == 1 and curr != 1):
                a += base
                curr -= 10  # carry
            a += base
            base *= 10
            curr //= 10
        return [a, n-a]


# Time:  O(nlogn)
# Space: O(logn)
class Solution2(object):
    def getNoZeroIntegers(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        return next([a, n-a] for a in xrange(1, n) if '0' not in '{}{}'.format(a, n-a))
```

open-the-lock.py

```
# DESC
# Given a target representing the value of the wheels that will unlock the lock, r
# eturn the minimum total number of turns required to open the lock, or -1 if it i
# s impossible.
# Example 3:
# You have a lock in front of you with 4 circular wheels.  Each wheel has 10 slots
# : '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.  The wheels can rotate freel
# y and wrap around: for example we can turn '9' to be '0', or '0' to be '9'.  Eac
# h move consists of turning one wheel one slot.
# Example 4:
# Example 1:
# Example 2:
# Note:
# You are given a list of deadends dead ends, meaning if the lock displays any of
# these codes, the wheels of the lock will stop turning and you will be unable to
# open it.
# The lock initially starts at '0000', a string representing the state of the 4 wheels.
# '0000'

# NOTE
# Every string in deadends and the string target will be a string of 4 digits from
#  the 10,000 possibilities '0000' to '9999'.
# The length of deadends will be in the range [1, 500].
# target will not be in the list deadends.

# EXAMPLE
# Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
# Output:
# 6
# Explanation:
# A sequence of valid moves would be "0000" -> "1000" -> "1100" ->
# "1200" -> "1201" -> "1202" -> "0202".
# Note that a sequence like "0000" -> "0001"
#  -> "0002" -> "0102" -> "0202" would be invalid,
# because the wheels of the lock
# become stuck after the display becomes the dead end "0102".
# Input: deadends = ["0000"], target = "8888"
# Output: -1
# Input: deadends = ["8888"], target = "0009"
# Output: 1
# Explanation:
# We can turn t
# he last wheel in reverse to move from "0000" -> "0009".
# Input: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], tar
# get = "8888"
# Output: -1
# Explanation:
# We can't reach the target without getting s
# tuck.

# Time:  O(k * n^k + d), n is the number of alphabets,
#                        k is the length of target,
#                        d is the size of deadends
# Space: O(k * n^k + d)

class Solution(object):
    def openLock(self, deadends, target):
        """
```

```
        :type deadends: List[str]
        :type target: str
        :rtype: int
        """
        dead = set(deadends)
        q = ["0000"]
        lookup = {"0000"}
        depth = 0
        while q:
            next_q = []
            for node in q:
                if node == target: return depth
                if node in dead: continue
                for i in xrange(4):
                    n = int(node[i])
                    for d in (-1, 1):
                        nn = (n+d) % 10
                        neighbor = node[:i] + str(nn) + node[i+1:]
                        if neighbor not in lookup:
                            lookup.add(neighbor)
                            next_q.append(neighbor)
            q = next_q
            depth += 1
        return -1
```

high-five.py

```python
# high-five is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections
import heapq


class Solution(object):
    def highFive(self, items):
        """
        :type items: List[List[int]]
        :rtype: List[List[int]]
        """
        min_heaps = collections.defaultdict(list)
        for i, val in items:
            heapq.heappush(min_heaps[i], val)
            if len(min_heaps[i]) > 5:
                heapq.heappop(min_heaps[i])
        return [[i, sum(min_heaps[i]) // len(min_heaps[i])] for i in sorted(min_heaps)]
```

building-h2o.py

```
# DESC
# Example 2:
# Write synchronization code for oxygen and hydrogen molecules that enforces these
#  constraints.
# 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 wa
# it until a complete molecule can be formed. Hydrogen and oxygen threads will be
# given releaseHydrogen and releaseOxygen methods respectively, which will allow t
# hem to pass the barrier. These threads should pass the barrier in groups of thre
# e, and they must be able to immediately bond with each other to form a water mol
# ecule. You must guarantee that all the threads from one molecule bond before any
#  other threads from the next molecule do.
# Constraints:
# We don't have to worry about matching the threads up explicitly; that is, the th
# reads do not necessarily know which other threads they are paired up with. The k
# ey is just that threads pass the barrier in complete sets; thus, if we examine t
# he sequence of threads that bond and divide them into groups of three, each grou
# p should contain one oxygen and two hydrogen threads.
# In other words:
# Example 1:

# NOTE
# If a hydrogen thread arrives at the barrier when no other threads are present, i
# t has to wait for an oxygen thread and another hydrogen thread.
# 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.
# If an oxygen thread arrives at the barrier when no hydrogen threads are present,
#  it has to wait for two hydrogen threads.

# EXAMPLE
# Input: "OOHHHH"
# Output: "HHOHHO"
# Explanation: "HOHHHO", "OHHHHO", "HHOHOH", "HOH
# HOH", "OHHHOH", "HHOOHH", "HOHOHH" and "OHHOHH" are also valid answers.
# Input: "HOH"
# Output: "HHO"
# Explanation: "HOH" and "OHH" are also valid answers.

# Time:  O(n)
# Space: O(1)

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
        """
        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()
```

maximum-depth-of-binary-tree.py

```
# DESC
# return its depth = 3.
# Given a binary tree, find its maximum depth.
# Given binary tree [3,9,20,null,null,15,7],
# Note: A leaf is a node with no children.
# Example:
# The maximum depth is the number of nodes along the longest path from the root no
# de down to the farthest leaf node.

# NOTE
#

# EXAMPLE
# 3
#    / \
#   9  20
#     /  \
#    15   7

# 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 maxDepth(self, root):
        if root is None:
            return 0
        else:
            return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

smallest-subtree-with-all-the-deepest-nodes.py

```
# DESC
# Note:
# The subtree of a node is that node, plus the set of all descendants of that node.
# Given a binary tree rooted at root, the depth of each node is the shortest dista
# nce to the root.
# A node is deepest if it has the largest depth possible among any node in the ent
# ire tree.
# Return the node with the largest depth such that it contains all the deepest nod
# es in its subtree.
# Example 1:

# NOTE
# The values of each node are unique.
# The number of nodes in the tree will be between 1 and 500.

# EXAMPLE
# Input: [3,5,1,6,2,0,8,null,null,7,4]
# Output: [2,7,4]
# Explanation:
#
#
#
# We return t
# he node with value 2, colored in yellow in the diagram.
# The nodes colored in blu
# e are the deepest nodes of the tree.
# The input "[3, 5, 1, 6, 2, 0, 8, null, null
# , 7, 4]" is a serialization of the given tree.
# The output "[2, 7, 4]" is a seria
# lization of the subtree rooted at the node with value 2.
# Both the input and outp
# ut have TreeNode type.

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

import collections


class Solution(object):
    def subtreeWithAllDeepest(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        Result = collections.namedtuple("Result", ("node", "depth"))

        def dfs(node):
            if not node:
                return Result(None, 0)
            left, right = dfs(node.left), dfs(node.right)
            if left.depth > right.depth:
                return Result(left.node, left.depth+1)
            if left.depth < right.depth:
                return Result(right.node, right.depth+1)
            return Result(node, left.depth+1)

        return dfs(root).node
```

coin-path.py

```python
# coin-path is not found.
# Time:  O(n * B)
# Space: O(n)

class Solution(object):
    def cheapestJump(self, A, B):
        """
        :type A: List[int]
        :type B: int
        :rtype: List[int]
        """
        result = []
        if not A or A[-1] == -1:
            return result
        n = len(A)
        dp, next_pos = [float("inf")] * n, [-1] * n
        dp[n-1] = A[n-1]
        for i in reversed(xrange(n-1)):
            if A[i] == -1:
                continue
            for j in xrange(i+1, min(i+B+1,n)):
                if A[i] + dp[j] < dp[i]:
                    dp[i] = A[i] + dp[j]
                    next_pos[i] = j
        if dp[0] == float("inf"):
            return result
        k = 0
        while k != -1:
            result.append(k+1)
            k = next_pos[k]
        return result
```

jump-game-iv.py

```python
# jump-game-iv is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def minJumps(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        groups = collections.defaultdict(list)
        for i, x in enumerate(arr):
            groups[x].append(i)
        q = collections.deque([(0, 0)])
        lookup = set([0])
        while q:
            pos, step = q.popleft()
            if pos == len(arr)-1:
                break
            neighbors = set(groups[arr[pos]] + [pos-1, pos+1])
            groups[arr[pos]] = []
            for p in neighbors:
                if p in lookup or not 0 <= p < len(arr):
                    continue
                lookup.add(p)
                q.append((p, step+1))
        return step
```

random-flip-matrix.py

```
# DESC
# Note:
# Explanation of Input Syntax:
# Example 2:
# The input is two lists: the subroutines called and their arguments. Solution's c
# onstructor has two arguments, n_rows and n_cols. flip and reset have no argument
# s. Arguments are always wrapped with a list, even if there aren't any.
# Example 1:
# You are given the number of rows n_rows and number of columns n_cols of a 2D bin
# ary matrix where all values are initially 0. Write a function flip which chooses
#  a 0 value uniformly at random, changes it to 1, and then returns the position [
# row.id, col.id] of that value. Also, write a function reset which sets all value
# s back to 0. Try to minimize the number of calls to system's Math.random() and o
# ptimize the time and space complexity.

# NOTE
# 1 <= n_rows, n_cols <= 10000
# the total number of calls to flip and reset will not exceed 1000.
# 0 <= row.id < n_rows and 0 <= col.id < n_cols
# flip will not be called when the matrix has no 0 values left.

# EXAMPLE
# Input:
# ["Solution","flip","flip","flip","flip"]
# [[2,3],[],[],[],[]]
# Output: [nu
# ll,[0,1],[1,2],[1,0],[1,1]]
# Input:
# ["Solution","flip","flip","reset","flip"]
# [[1,2],[],[],[],[]]
# Output: [n
# ull,[0,0],[0,1],null,[0,0]]

# Time:  ctor:  O(1)
#        flip:  O(1)
#        reset: O(min(f, r * c))
# Space: O(min(f, r * c))

import random


class Solution(object):

    def __init__(self, n_rows, n_cols):
        """
        :type n_rows: int
        :type n_cols: int
        """
        self.__n_rows = n_rows
        self.__n_cols = n_cols
        self.__n = n_rows*n_cols
        self.__lookup = {}


    def flip(self):
        """
        :rtype: List[int]
        """
```

```
        self.__n -= 1
        target = random.randint(0, self.__n)
        x = self.__lookup.get(target, target)
        self.__lookup[target] = self.__lookup.get(self.__n, self.__n)
        return divmod(x, self.__n_cols)


    def reset(self):
        """
        :rtype: void
        """
        self.__n = self.__n_rows*self.__n_cols
        self.__lookup = {}
```

lexicographically-smallest-equivalent-string.py

```python
# lexicographically-smallest-equivalent-string is not found.
# Time:  O(nlog*n) ~= O(n), n is the length of S
# 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[max(x_root, y_root)] = min(x_root, y_root)
        return True


class Solution(object):
    def smallestEquivalentString(self, A, B, S):
        """
        :type A: str
        :type B: str
        :type S: str
        :rtype: str
        """
        union_find = UnionFind(26)
        for i in xrange(len(A)):
            union_find.union_set(ord(A[i])-ord('a'), ord(B[i])-ord('a'))
        result = []
        for i in xrange(len(S)):
            parent = union_find.find_set(ord(S[i])-ord('a'))
            result.append(chr(parent+ord('a')))
        return "".join(result)
```

next-greater-element-ii.py

```
# DESC
# Note:
# The length of given array won't exceed 10000.
# Example 1:
# Given a circular array (the next element of the last element is the first elemen
# t of the array), print the Next Greater Number for every element. The Next Great
# er 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.

# NOTE
#

# EXAMPLE
# Input: [1,2,1]
# Output: [2,-1,2]
# Explanation: The first 1's next greater number i
# s 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.

# 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

```
# DESC
# Rotate the image by 90 degrees (clockwise).
# Example 1:
# 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 2:
# Note:
# You are given an n x n 2D matrix representing an image.

# NOTE
#

# EXAMPLE
# Given input matrix =
# [
#   [1,2,3],
#   [4,5,6],
#   [7,8,9]
# ],
#
# rotate the input mat
# rix in-place such that it becomes:
# [
#   [7,4,1],
#   [8,5,2],
#   [9,6,3]
# ]
# Given input matrix =
# [
#   [ 5, 1, 9,11],
#   [ 2, 4, 8,10],
#   [13, 3, 6, 7],
#   [15,
# 14,12,16]
# ],
#
# rotate the input matrix in-place such that it becomes:
# [
#   [15,13
# , 2, 5],
#   [14, 3, 4, 1],
#   [12, 6, 8, 9],
#   [16, 7,10,11]
# ]

# 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

# 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)]
```

minimum-time-visiting-all-points.py

```python
# minimum-time-visiting-all-points is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minTimeToVisitAllPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        return sum(max(abs(points[i+1][0] - points[i][0]),
                       abs(points[i+1][1] - points[i][1]))
                   for i in xrange(len(points)-1))
```

divisor-game.py

```
# DESC
# Example 1:
# Initially, there is a number N on the chalkboard.  On each player's turn, that p
# layer makes a move consisting of:
# Note:
# Alice and Bob take turns playing a game, with Alice starting first.
# Example 2:
# Also, if a player cannot make a move, they lose the game.
# Return True if and only if Alice wins the game, assuming both players play optimally.

# NOTE
# 1 <= N <= 1000
# Replacing the number N on the chalkboard with N - x.
# Choosing any x with 0 < x < N and N % x == 0.

# EXAMPLE
# Input: 2
# Output: true
# Explanation: Alice chooses 1, and Bob has no more moves.
# Input: 3
# Output: false
# Explanation: Alice chooses 1, Bob chooses 1, and Alice ha
# s no more moves.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def divisorGame(self, N):
        """
        :type N: int
        :rtype: bool
        """
        # 1. if we get an even, we can choose x = 1
        #    to make the opponent always get an odd
        # 2. if the opponent gets an odd, he can only choose x = 1 or other odds
        #    and we can still get an even
        # 3. at the end, the opponent can only choose x = 1 and we win
        # 4. in summary, we win if only if we get an even and
        #    keeps even until the opponent loses
        return N % 2 == 0


# Time:  O(n^3/2)
# Space: O(n)
# dp solution
class Solution2(object):
    def divisorGame(self, N):
        """
        :type N: int
        :rtype: bool
        """
        def memoization(N, dp):
            if N == 1:
                return False
            if N not in dp:
                result = False
                for i in xrange(1, N+1):
```

```
            if i*i > N:
                break
            if N % i == 0:
                if not memoization(N-i, dp):
                    result = True
                    break
        dp[N] = result
    return dp[N]

return memoization(N, {})
```

pascals-triangle-ii.py

```
# pascals-triangle-ii is not found.
# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    # @return a list of integers
    def getRow(self, rowIndex):
        result = [0] * (rowIndex + 1)
        for i in xrange(rowIndex + 1):
            old = result[0] = 1
            for j in xrange(1, i + 1):
                old, result[j] = result[j], old + result[j]
        return result

    def getRow2(self, rowIndex):
        """
        :type rowIndex: int
        :rtype: List[int]
        """
        row = [1]
        for _ in range(rowIndex):
            row = [x + y for x, y in zip([0] + row, row + [0])]
        return row

    def getRow3(self, rowIndex):
        """
        :type rowIndex: int
        :rtype: List[int]
        """
        if rowIndex == 0: return [1]
        res = [1, 1]

        def add(nums):
            res = nums[:1]
            for i, j in enumerate(nums):
                if i < len(nums) - 1:
                    res += [nums[i] + nums[i + 1]]
            res += nums[:1]
            return res

        while res[1] < rowIndex:
            res = add(res)
        return res


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    # @return a list of integers
    def getRow(self, rowIndex):
        result = [1]
        for i in range(1, rowIndex + 1):
            result = [1] + [result[j - 1] + result[j] for j in xrange(1, i)] + [1]
        return result
```

k-similar-strings.py

```
# DESC
# Example 3:
# Note:
# Strings A and B are K-similar (for some non-negative integer K) if we can swap t
# he positions of two letters in A exactly K times so that the resulting string eq
# uals B.
# Example 1:
# Example 4:
# Given two anagrams A and B, return the smallest K for which A and B are K-similar.
# Example 2:

# NOTE
# 1 <= A.length == B.length <= 20
# A and B contain only lowercase letters from the set {'a', 'b', 'c', 'd', 'e', 'f'}

# EXAMPLE
# Input: A = "ab", B = "ba"
# Output: 1
# Input: A = "aabc", B = "abca"
# Output: 2
# Input: A = "abc", B = "bca"
# Output: 2
# Input: A = "abac", B = "baca"
# Output: 2

# Time:  O(n * n!/(c_a!*...*c_z!), n is the length of A, B,
#                                  c_a...c_z is the count of each alphabet,
#                                  n = sum(c_a...c_z)
# Space: O(n * n!/(c_a!*...*c_z!)

import collections


class Solution(object):
    def kSimilarity(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: int
        """
        def neighbors(s, B):
            for i, c in enumerate(s):
                if c != B[i]:
                    break
            t = list(s)
            for j in xrange(i+1, len(s)):
                if t[j] == B[i]:
                    t[i], t[j] = t[j], t[i]
                    yield "".join(t)
                    t[j], t[i] = t[i], t[j]

        q = collections.deque([A])
        lookup = set()
        result = 0
        while q:
            for _ in xrange(len(q)):
                s = q.popleft()
                if s == B:
```

```
            return result
        for t in neighbors(s, B):
            if t not in lookup:
                lookup.add(t)
                q.append(t)
    result += 1
```

surrounded-regions.py

```
# DESC
# After running your function, the board should be:
# Explanation:
# Surrounded regions shouldn’t be on the border, which means that any 'O' on the b
# order 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 a
# re connected if they are adjacent cells connected horizontally or vertically.
# Example:
# A region is captured by flipping all 'O's into 'X's in that surrounded region.
# Given a 2D board containing 'X' and 'O' (the letter O), capture all regions surr
# ounded by 'X'.

# NOTE
#

# EXAMPLE
# X X X X
# X X X X
# X X X X
# X O X X
# X X X X
# X O O X
# X X O X
# X O X X

# 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'
                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'
```

leaf-similar-trees.py

```
# DESC
# Consider all the leaves of a binary tree.  From left to right order, the values
# of those leaves form a leaf value sequence.
# Constraints:
# Two binary trees are considered leaf-similar if their leaf value sequence is the
#  same.
# Return true if and only if the two given trees with head nodes root1 and root2 a
# re leaf-similar.
# For example, in the given tree above, the leaf value sequence is (6, 7, 4, 9, 8).

# NOTE
# Both of the given trees will have values between 0 and 200
# Both of the given trees will have between 1 and 200 nodes.

# EXAMPLE
#

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

import itertools


# 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 leafSimilar(self, root1, root2):
        """
        :type root1: TreeNode
        :type root2: TreeNode
        :rtype: bool
        """
        def dfs(node):
            if not node:
                return
            if not node.left and not node.right:
                yield node.val
            for i in dfs(node.left):
                yield i
            for i in dfs(node.right):
                yield i
        return all(a == b for a, b in
                   itertools.izip_longest(dfs(root1), dfs(root2)))
```

remove-sub-folders-from-the-filesystem.py

```python
# remove-sub-folders-from-the-filesystem is not found.
# Time:  O(n), n is the total sum of the lengths of folder names
# Space: O(t), t is the number of nodes in trie

import collections
import itertools


class Solution(object):
    def removeSubfolders(self, folder):
        """
        :type folder: List[str]
        :rtype: List[str]
        """
        def dfs(curr, path, result):
            if "_end" in curr:
                result.append("/" + "/".join(path))
                return
            for c in curr:
                if c == "_end":
                    continue
                path.append(c)
                dfs(curr[c], path, result)
                path.pop()

        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        for f in folder:
            f_list = f.split("/")
            reduce(dict.__getitem__,
                   itertools.islice(f_list, 1, len(f_list)),
                   trie).setdefault("_end")
        result = []
        dfs(trie, [], result)
        return result
```

range-sum-of-bst.py

```
# DESC
# The binary search tree is guaranteed to have unique values.
# Given the root node of a binary search tree, return the sum of values of all nod
# es with value between L and R (inclusive).
# Example 1:
# Note:
# Example 2:

# NOTE
# The final answer is guaranteed to be less than 2^31.
# The number of nodes in the tree is at most 10000.

# EXAMPLE
# Input: root = [10,5,15,3,7,13,18,1,null,6], L = 6, R = 10
# Output: 23
# Input: root = [10,5,15,3,7,null,18], L = 7, R = 15
# Output: 32

# 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 rangeSumBST(self, root, L, R):
        """
        :type root: TreeNode
        :type L: int
        :type R: int
        :rtype: int
        """
        result = 0
        s = [root]
        while s:
            node = s.pop()
            if node:
                if L <= node.val <= R:
                    result += node.val
                if L < node.val:
                    s.append(node.left)
                if node.val < R:
                    s.append(node.right)
        return result
```

power-of-three.py

```
# DESC
# Example 2:
# Example 3:
# Example 4:
# Follow up:
#
# Could you do it without using any loop / recursion?
# Example 1:
# Given an integer, write a function to determine if it is a power of three.

# NOTE
#

# EXAMPLE
# Input: 27
# Output: true
# Input: 45
# Output: false
# Input: 9
# Output: true
# Input: 0
# Output: false

# Time:  O(1)
# Space: O(1)

import math


class Solution(object):
    def __init__(self):
        self.__max_log3 = int(math.log(0x7fffffff) / math.log(3))
        self.__max_pow3 = 3 ** self.__max_log3

    def isPowerOfThree(self, n):
        """
        :type n: int
        :rtype: bool
        """
        return n > 0 and self.__max_pow3 % n == 0


class Solution2(object):
    def isPowerOfThree(self, n):
        return n > 0 and (math.log10(n)/math.log10(3)).is_integer()
```

binary-tree-pruning.py

```
# DESC
# We are given the head node root of a binary tree, where additionally every node'
# s value is either a 0 or a 1.
# Return the same tree where every subtree (of the given tree) not containing a 1
# has been removed.
# (Recall that the subtree of a node X is X, plus every node that is a descendant
# of X.)
# Note:

# NOTE
# The value of each node will only be 0 or 1.
# The binary tree will have at most 200 nodes.

# EXAMPLE
# Example 2:
# Input: [1,0,1,0,0,0,1]
# Output: [1,null,1,null,1]
# Example 1:
# Input: [1,null,0,0,1]
# Output: [1,null,0,null,1]
#
# Explanation:
# Only
# the red nodes satisfy the property "every subtree not containing a 1".
# The diagr
# am on the right represents the answer.
# Example 3:
# Input: [1,1,0,1,1,0,1,0]
# Output: [1,1,0,1,1,null,1]

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

class Solution(object):
    def pruneTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        if not root:
            return None
        root.left = self.pruneTree(root.left)
        root.right = self.pruneTree(root.right)
        if not root.left and not root.right and root.val == 0:
            return None
        return root
```

magical-string.py

```
# DESC
# You can see that the occurrence sequence above is the S itself.
# and the occurrences of '1's or '2's in each group are:
# Given an integer N as input, return the number of '1's in the first N number in
# the magical string S.
# 1   2          2    1   1    2     1    2     2    1    2    2 ......
# If we group the consecutive '1's and '2's in S, it will be:
# A magical string S consists of only '1' and '2' and obeys the following rules:
# The first few elements of string S is the following:
# S = "12211212212211211122......"
# Example 1:
# The string S is magical because concatenating the number of contiguous occurrenc
# es of characters '1' and '2' generates the string S itself.
# Note:
# N will not exceed 100,000.
# 1   22  11  2  1  22  1  22  11  2  11  22 ......

# NOTE
#

# EXAMPLE
# Input: 6
# Output: 3
# Explanation: The first 6 elements of magical string S is "122
# 11" 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

```
# DESC
# Example 2:
# Given an array nums sorted in ascending order, return true if and only if you ca
# n split it into 1 or more subsequences such that each subsequence consists of co
# nsecutive integers and has length at least 3.
# Example 1:
# Constraints:
# Example 3:

# NOTE
# 1 <= nums.length <= 10000

# EXAMPLE
# Input: [1,2,3,3,4,5]
# Output: True
# Explanation:
# You can split them into two conse
# cutive subsequences :
# 1, 2, 3
# 3, 4, 5
# Input: [1,2,3,3,4,4,5,5]
# Output: True
# Explanation:
# You can split them into two c
# onsecutive subsequences :
# 1, 2, 3, 4, 5
# 3, 4, 5
# Input: [1,2,3,4,4,5]
# Output: False

# 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
# beautiful-arra is not found.
# 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

```
# DESC
# The path does not need to start or end at the root or a leaf, but it must go dow
# nwards
# (traveling only from parent nodes to child nodes).
# Find the number of paths that sum to a given value.
# Example:
# You are given a binary tree in which each node contains an integer value.
# The tree has no more than 1,000 nodes and the values are in the range -1,000,000
#  to 1,000,000.

# NOTE
#

# 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)
# 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)
```

find-the-index-of-the-large-integer.py

```python
# find-the-index-of-the-large-integer is not found.
# Time:  O(logn)
# Space: O(1)

class ArrayReader(object):
   def compareSub(self, l, r, x, y):
       pass

   def length(self):
       pass

class Solution(object):
    def getIndex(self, reader):
        """
        :type reader: ArrayReader
        :rtype: integer
        """
        left, right = 0, reader.length()-1
        while left < right:
            mid = left + (right-left)//2
            if reader.compareSub(left, mid, mid if (right-left+1)%2 else mid+1, right) >= 0:
                right = mid
            else:
                left = mid+1
        return left
```

synonymous-sentences.py

```
# synonymous-sentences is not found.
# Time:  O(p*l * log(p*l)), p is the production of all number of synonyms
#                         , l is the length of a word
# Space: O(p*l)

import collections
import itertools


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:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        return True


class Solution(object):
    def generateSentences(self, synonyms, text):
        """
        :type synonyms: List[List[str]]
        :type text: str
        :rtype: List[str]
        """
        def assign_id(x, lookup, inv_lookup):
            if x in lookup:
                return
            lookup[x] = len(lookup)
            inv_lookup[lookup[x]] = x

        lookup, inv_lookup = {}, {}
        for u, v in synonyms:
            assign_id(u, lookup, inv_lookup), assign_id(v, lookup, inv_lookup)
        union_find = UnionFind(len(lookup))
        for u, v in synonyms:
            union_find.union_set(lookup[u], lookup[v])
        groups = collections.defaultdict(list)
        for i in xrange(len(union_find.set)):
            groups[union_find.find_set(i)].append(i)
        result = []
        for w in text.split(' '):
            if w not in lookup:
                result.append([w])
                continue
            result.append(sorted(map(lambda x: inv_lookup[x],
                                     groups[union_find.find_set(lookup[w])])))
        return [" ".join(sentense) for sentense in itertools.product(*result)]
```

maximum-profit-in-job-scheduling.py

```python
# maximum-profit-in-job-scheduling is not found.
# Time:  O(nlogn)
# Space: O(n)

import itertools
import bisect


class Solution(object):
    def jobScheduling(self, startTime, endTime, profit):
        """
        :type startTime: List[int]
        :type endTime: List[int]
        :type profit: List[int]
        :rtype: int
        """
        jobs = sorted(itertools.izip(endTime, startTime, profit))
        dp = [(0, 0)]
        for e, s, p in jobs:
            i = bisect.bisect_right(dp, (s+1, 0))-1
            if dp[i][1]+p > dp[-1][1]:
                dp.append((e, dp[i][1]+p))
        return dp[-1][1]


# Time:  O(nlogn)
# Space: O(n)
import heapq
class Solution(object):
    def jobScheduling(self, startTime, endTime, profit):
        """
        :type startTime: List[int]
        :type endTime: List[int]
        :type profit: List[int]
        :rtype: int
        """
        min_heap = zip(startTime, endTime, profit)
        heapq.heapify(min_heap)
        result = 0
        while min_heap:
            s, e, p = heapq.heappop(min_heap)
            if s < e:
                heapq.heappush(min_heap, (e, s, result+p))
            else:
                result = max(result, p)
        return result
```

count-negative-numbers-in-a-sorted-matrix.py

```python
# count-negative-numbers-in-a-sorted-matrix is not found.
# Time:  O(m + n)
# Space: O(1)

class Solution(object):
    def countNegatives(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        result, c = 0, len(grid[0])-1
        for row in grid:
            while c >= 0 and row[c] < 0:
                c -= 1
            result += len(grid[0])-1-c
        return result
```

design-underground-system.py

```
# DESC
# 2. checkOut(int id, string stationName, int t)
# Example 2:
# Example 1:
# Implement the class UndergroundSystem that supports three methods:
# Constraints:
# 1. checkIn(int id, string stationName, int t)
# checkIn(int id, string stationName, int t)
# 3. getAverageTime(string startStation, string endStation)
# 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.

# NOTE
# There will be at most 20000 operations.
# 1 <= id, t <= 10^6
# A customer can only be checked into one place at a time.
# 1 <= stationName.length <= 10
# The average time is computed from all the previous traveling from startStation t
# o endStation that happened directly.
# Returns the average time to travel between the startStation and the endStation.
# Answers within 10^-5 of the actual value will be accepted as correct.
# All strings consist of uppercase, lowercase English letters and digits.
# A customer with id card equal to id, gets in the station stationName at time t.
# A customer with id card equal to id, gets out from the station stationName at time t.
# Call to getAverageTime is always valid.

# EXAMPLE
# Input
# ["UndergroundSystem","checkIn","checkIn","checkIn","checkOut","checkOut","
# checkOut","getAverageTime","getAverageTime","checkIn","getAverageTime","checkOut
# ","getAverageTime"]
# [[],[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,14.00000,11.000
# 00,null,11.00000,null,12.00000]
#
# Explanation
# UndergroundSystem undergroundSystem
#  = new UndergroundSystem();
# undergroundSystem.checkIn(45, "Leyton", 3);
# undergro
# undSystem.checkIn(32, "Paradise", 8);
# undergroundSystem.checkIn(27, "Leyton", 10
# );
# undergroundSystem.checkOut(45, "Waterloo", 15);
# undergroundSystem.checkOut(27
# , "Waterloo", 20);
# undergroundSystem.checkOut(32, "Cambridge", 22);
# undergroundS
# ystem.getAverageTime("Paradise", "Cambridge");       // return 14.00000. There w
# as only one travel from "Paradise" (at time 8) to "Cambridge" (at time 22)
# under
# groundSystem.getAverageTime("Leyton", "Waterloo");          // return 11.00000.
```

```
# There were two travels from "Leyton" to "Waterloo", a customer with id=45 from t
# ime=3 to time=15 and a customer with id=27 from time=10 to time=20. So the avera
# ge time is ( (15-3) + (20-10) ) / 2 = 11.00000
# undergroundSystem.checkIn(10, "Le
# yton", 24);
# undergroundSystem.getAverageTime("Leyton", "Waterloo");          //
# return 11.00000
# undergroundSystem.checkOut(10, "Waterloo", 38);
# undergroundSyste
# m.getAverageTime("Leyton", "Waterloo");          // return 12.00000
# Input
# ["UndergroundSystem","checkIn","checkOut","getAverageTime","checkIn","chec
# kOut","getAverageTime","checkIn","checkOut","getAverageTime"]
# [[],[10,"Leyton",3
# ],[10,"Paradise",8],["Leyton","Paradise"],[5,"Leyton",10],[5,"Paradise",16],["Le
# yton","Paradise"],[2,"Leyton",21],[2,"Paradise",30],["Leyton","Paradise"]]
#
# Outp
# ut
# [null,null,null,5.00000,null,null,5.50000,null,null,6.66667]
#
# Explanation
# Und
# ergroundSystem undergroundSystem = new UndergroundSystem();
# undergroundSystem.ch
# eckIn(10, "Leyton", 3);
# undergroundSystem.checkOut(10, "Paradise", 8);
# undergrou
# ndSystem.getAverageTime("Leyton", "Paradise"); // return 5.00000
# undergroundSyst
# em.checkIn(5, "Leyton", 10);
# undergroundSystem.checkOut(5, "Paradise", 16);
# unde
# rgroundSystem.getAverageTime("Leyton", "Paradise"); // return 5.50000
# undergroun
# dSystem.checkIn(2, "Leyton", 21);
# undergroundSystem.checkOut(2, "Paradise", 30);
#
# undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 6.66667

# 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])


    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
```

number-complement.py

```python
# DESC
# Example 2:
# Given a positive integer num, output its complement number. The complement strat
# egy is to flip the bits of its binary representation.
# Constraints:
# Example 1:

# NOTE
# You could assume no leading zero bit in the integer's binary representation.
# This question is the same as 1009: https://leetcode.com/problems/complement-of-b
# ase-10-integer/
# num >= 1
# The given integer num is guaranteed to fit within the range of a 32-bit signed i
# nteger.

# EXAMPLE
# Input: num = 1
# Output: 0
# Explanation: The binary representation of 1 is 1 (no le
# ading zero bits), and its complement is 0. So you need to output 0.
# Input: num = 5
# Output: 2
# Explanation: The binary representation of 5 is 101 (no
# leading zero bits), and its complement is 010. So you need to output 2.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def findComplement(self, num):
        """
        :type num: int
        :rtype: int
        """
        return 2 ** (len(bin(num)) - 2) - 1 - num


class Solution2(object):
    def findComplement(self, num):
        i = 1
        while i <= num:
            i <<= 1
        return (i - 1) ^ num


class Solution3(object):
    def findComplement(self, num):
        bits = '{0:b}'.format(num)
        complement_bits = ''.join('1' if bit == '0' else '0' for bit in bits)
        return int(complement_bits, 2)
```

add-digits.py

```python
# DESC
# Follow up:
#
# Could you do it without any loop/recursion in O(1) runtime?
# Example:
# Given a non-negative integer num, repeatedly add all its digits until the result
#  has only one digit.

# NOTE
#

# EXAMPLE
# Input: 38
# Output: 2
# Explanation: The process is like: 3 + 8 = 11, 1 + 1 = 2.
#
#             Since 2 has only one digit, return it.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    """
    :type num: int
    :rtype: int
    """
    def addDigits(self, num):
        return (num - 1) % 9 + 1 if num > 0 else 0
```

delete-columns-to-make-sorted.py

```
# DESC
# Example 3:
# Suppose we chose a set of deletion indices D such that after deletions, each rem
# aining column in A is in non-decreasing sorted order.
# Now, we may choose any set of deletion indices, and for each string, we delete a
# ll the characters in those indices.
# Constraints:
# Return the minimum possible value of D.length.
# We are given an array A of N lowercase letter strings, all of the same length.
# For example, if we have an array A = ["abcdef","uvwxyz"] and deletion indices {0
# , 2, 3}, then the final array after deletions is ["bef", "vyz"], and the remaini
# ng columns of A are ["b","v"], ["e","y"], and ["f","z"].  (Formally, the c-th co
# lumn is [A[0][c], A[1][c], ..., A[A.length-1][c]]).
# Example 1:
# Example 2:

# NOTE
# 1 <= A[i].length <= 1000
# 1 <= A.length <= 100

# EXAMPLE
# Input: A = ["cba","daf","ghi"]
# Output: 1
# Explanation:
# After choosing D = {1}, e
# ach column ["c","d","g"] and ["a","f","i"] are in non-decreasing sorted order.
# I
# f we chose D = {}, then a column ["b","a","h"] would not be in non-decreasing so
# rted order.
# Input: A = ["a","b"]
# Output: 0
# Explanation: D = {}
# Input: A = ["zyx","wvu","tsr"]
# Output: 3
# Explanation: D = {0, 1, 2}

# Time:  O(n * l)
# Space: O(1)

class Solution(object):
    def minDeletionSize(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        result = 0
        for c in xrange(len(A[0])):
            for r in xrange(1, len(A)):
                if A[r-1][c] > A[r][c]:
                    result += 1
                    break
        return result


# Time:  O(n * l)
# Space: O(n)
import itertools
```

```
class Solution2(object):
    def minDeletionSize(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        result = 0
        for col in itertools.izip(*A):
            if any(col[i] > col[i+1] for i in xrange(len(col)-1)):
                result += 1
        return result
```

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

```
# find-all-duplicates-in-an-arra is not found.
# 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):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        return [elem for elem, count in Counter(nums).items() if count == 2]
```

two-sum-bsts.py

```
# two-sum-bsts is not found.
# Time:  O(n)
# Space: O(n)

# 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 twoSumBSTs(self, root1, root2, target):
        """
        :type root1: TreeNode
        :type root2: TreeNode
        :type target: int
        :rtype: bool
        """
        def inorder_gen(root, asc=True):
            result, stack = [], [(root, False)]
            while stack:
                root, is_visited = stack.pop()
                if root is None:
                    continue
                if is_visited:
                    yield root.val
                else:
                    if asc:
                        stack.append((root.right, False))
                        stack.append((root, True))
                        stack.append((root.left, False))
                    else:
                        stack.append((root.left, False))
                        stack.append((root, True))
                        stack.append((root.right, False))

        left_gen, right_gen = inorder_gen(root1, True), inorder_gen(root2, False)
        left, right = next(left_gen), next(right_gen)
        while left is not None and right is not None:
            if left + right < target:
                left = next(left_gen)
            elif left + right > target:
                right = next(right_gen)
            else:
                return True
        return False
```

number-of-islands-ii.py

```python
# number-of-islands-ii is not found.
# Time:  O(klog*k) ~= O(k), k is the length of the positions
# Space: O(k)

class Solution(object):
    def numIslands2(self, m, n, positions):
        """
        :type m: int
        :type n: int
        :type positions: List[List[int]]
        :rtype: List[int]
        """
        def node_id(node, n):
            return node[0] * n + node[1]

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

        def union_set(x, y):
            x_root, y_root = find_set(x), find_set(y)
            set[min(x_root, y_root)] = max(x_root, y_root)

        numbers = []
        number = 0
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
        set = {}
        for position in positions:
            node = (position[0], position[1])
            set[node_id(node, n)] = node_id(node, n)
            number += 1

            for d in directions:
                neighbor = (position[0] + d[0], position[1] + d[1])
                if 0 <= neighbor[0] < m and 0 <= neighbor[1] < n and \
                   node_id(neighbor, n) in set:
                   if find_set(node_id(node, n)) != find_set(node_id(neighbor, n)):
                       # Merge different islands, amortised time: O(log*k) ~= O(1)
                       union_set(node_id(node, n), node_id(neighbor, n))
                       number -= 1
            numbers.append(number)

        return numbers
```

count-number-of-nice-subarrays.py

```python
# count-number-of-nice-subarrays is not found.
# Time:  O(n)
# Space: O(k)

class Solution(object):
    def numberOfSubarrays(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        def atMost(nums, k):
            result, left, count = 0, 0, 0
            for right in xrange(len(nums)):
                count += nums[right]%2
                while count > k:
                    count -= nums[left]%2
                    left += 1
                result += right-left+1
            return result

        return atMost(nums, k) - atMost(nums, k-1)


# Time:  O(n)
# Space: O(k)
import collections


class Solution2(object):
    def numberOfSubarrays(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        result = 0
        dq = collections.deque([-1])
        for i in xrange(len(nums)):
            if nums[i]%2:
                dq.append(i)
            if len(dq) > k+1:
                dq.popleft()
            if len(dq) == k+1:
                result += dq[1]-dq[0]
        return result
```

zigzag-conversion.py

```
# DESC
# Example 2:
# Write the code that will take a string and make this conversion given a number o
# f rows:
# Example 1:
# And then read line by line: "PAHNAPLSIIGYIR"
# "PAHNAPLSIIGYIR"
# 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)

# NOTE
#

# EXAMPLE
# string convert(string s, int numRows);
# Input: s = "PAYPALISHIRING", numRows = 4
# Output: "PINALSIGYAHRPI"
# Explanation:
#
#
# P     I    N
# A   L S  I G
# Y A   H R
# P     I
# Input: s = "PAYPALISHIRING", numRows = 3
# Output: "PAHNAPLSIIGYIR"
# P   A   H   N
# A P L S I I G
# Y   I   R

# 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
```

longest-substring-with-at-most-two-distinct-characters.py

```python
# longest-substring-with-at-most-two-distinct-characters is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param s, a string
    # @return an integer
    def lengthOfLongestSubstringTwoDistinct(self, s):
        longest, start, distinct_count, visited = 0, 0, 0, [0 for _ in xrange(256)]
        for i, char in enumerate(s):
            if visited[ord(char)] == 0:
                distinct_count += 1
            visited[ord(char)] += 1
            while distinct_count > 2:
                visited[ord(s[start])] -= 1
                if visited[ord(s[start])] == 0:
                    distinct_count -= 1
                start += 1
            longest = max(longest, i - start + 1)
        return longest


# Time:  O(n)
# Space: O(1)
from collections import Counter


class Solution2(object):
    def lengthOfLongestSubstringTwoDistinct(self, s):
        """
        :type s: str
        :rtype: int
        """
        counter = Counter()
        left, max_length = 0, 0
        for right, char in enumerate(s):
            counter[char] += 1
            while len(counter) > 2:
                counter[s[left]] -= 1
                if counter[s[left]] == 0:
                    del counter[s[left]]
                left += 1
            max_length = max(max_length, right-left+1)
        return max_length
```

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

```python
# number-of-dice-rolls-with-target-sum is not found.
# Time:  O(d * f * t)
# Space: O(t)

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

```
# DESC
# Note:
# Example 3:
# Example 1:
# Since the answer may be large, output the answer modulo 10^9 + 7.
# A chess knight can move as indicated in the chess diagram below:
# This time, we place our chess knight on any numbered key of a phone pad (indicat
# ed above), and the knight makes N-1 hops.  Each hop must be from one key to anot
# her numbered key.
# How many distinct numbers can you dial in this manner?
# Each time it lands on a key (including the initial placement of the knight), it
# presses the number of that key, pressing N digits total.
# Example 2:

# NOTE
# 1 <= N <= 5000

# EXAMPLE
# Input: 2
# Output: 20
# Input: 3
# Output: 46
# Input: 1
# Output: 10

# 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
```

pseudo-palindromic-paths-in-a-binary-tree.py

```python
# pseudo-palindromic-paths-in-a-binary-tree is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution(object):
    def pseudoPalindromicPaths (self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result = 0
        stk = [(root, 0)]
        while stk:
            node, count = stk.pop()
            if not node:
                continue
            count ^= 1 << (node.val-1)
            result += int(node.left == node.right and count&(count-1) == 0)
            stk.append((node.right, count))
            stk.append((node.left, count))
        return result


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def pseudoPalindromicPaths (self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, count):
            if not root:
                return 0
            count ^= 1 << (node.val-1)
            return int(node.left == node.right and count&(count-1) == 0) + \
                   dfs(node.left, count) + dfs(node.right, count)
        return dfs(root, 0)
```

shortest-unsorted-continuous-subarray.py

```
# shortest-unsorted-continuous-subarra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findUnsortedSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        n = len(nums)
        left, right = -1, -2
        min_from_right, max_from_left = nums[-1], nums[0]
        for i in xrange(1, n):
            max_from_left = max(max_from_left, nums[i])
            min_from_right = min(min_from_right, nums[n-1-i])
            if nums[i] < max_from_left: right = i
            if nums[n-1-i] > min_from_right: left = n-1-i


# Time:  O(nlogn)
# Space: O(n)
class Solution2(object):
    def findUnsortedSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        a = sorted(nums) #sort the list
        left, right = 0, len(nums) -1 #define left and right pointer
        while (nums[left] == a[left] or nums[right] == a[right]):
            if right - left <= 1:
                return 0
            if nums[left] == a[left]:
                left += 1
            if nums[right] == a[right]:
                right -= 1
        return right - left + 1
```

dungeon-game.py

```
# DESC
# Write a function to determine the knight's minimum initial health so that he is
# able to rescue the princess.
# Note:
# The knight has an initial health point represented by a positive integer. If at
# any point his health point drops to 0 or below, he dies immediately.
# Some of the rooms are guarded by demons, so the knight loses health (negative in
# tegers) upon entering these rooms; other rooms are either empty (0's) or contain
#  magic orbs that increase the knight's health (positive integers).
# For example, given the dungeon below, the initial health of the knight must be a
# t least 7 if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.
# The demons had captured the princess (P) and imprisoned her in the bottom-right
# corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid.
# Our valiant knight (K) was initially positioned in the top-left room and must fi
# ght his way through the dungeon to rescue the princess.
# In order to reach the princess as quickly as possible, the knight decides to mov
# e only rightward or downward in each step.

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

# EXAMPLE
#

# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    # @param dungeon, a list of lists of integers
    # @return a integer
    def calculateMinimumHP(self, dungeon):
        DP = [float("inf") for _ in dungeon[0]]
        DP[-1] = 1

        for i in reversed(xrange(len(dungeon))):
            DP[-1] = max(DP[-1] - dungeon[i][-1], 1)
            for j in reversed(xrange(len(dungeon[i]) - 1)):
                min_HP_on_exit = min(DP[j], DP[j + 1])
                DP[j] = max(min_HP_on_exit - dungeon[i][j], 1)

        return DP[0]

# Time:  O(m * n logk), where k is the possible maximum sum of loses
# Space: O(m + n)
class Solution2(object):
    # @param dungeon, a list of lists of integers
    # @return a integer
    def calculateMinimumHP(self, dungeon):
        maximum_loses = 0
        for rooms in dungeon:
            for room in rooms:
                if room < 0:
                    maximum_loses += abs(room)

        return self.binarySearch(dungeon, maximum_loses)
```

```
def binarySearch(self, dungeon, maximum_loses):
    start, end = 1, maximum_loses + 1
    result = 0
    while start < end:
        mid = start + (end - start) / 2
        if self.DP(dungeon, mid):
            end = mid
        else:
            start = mid + 1
    return start

def DP(self, dungeon, HP):
    remain_HP = [0 for _ in dungeon[0]]
    remain_HP[0] = HP + dungeon[0][0]
    for j in xrange(1, len(remain_HP)):
        if remain_HP[j - 1] > 0:
            remain_HP[j] = max(remain_HP[j - 1] + dungeon[0][j], 0)

    for i in xrange(1, len(dungeon)):
        if remain_HP[0] > 0:
            remain_HP[0] = max(remain_HP[0] + dungeon[i][0], 0)
        else:
            remain_HP[0] = 0

        for j in xrange(1, len(remain_HP)):
            remain = 0
            if remain_HP[j - 1] > 0:
                remain = max(remain_HP[j - 1] + dungeon[i][j], remain)
            if remain_HP[j] > 0:
                remain = max(remain_HP[j] + dungeon[i][j], remain)
            remain_HP[j] = remain

    return remain_HP[-1] > 0
```

knight-probability-in-chessboard.py

```
# DESC
# On an NxN chessboard, a knight starts at the r-th row and c-th column and attemp
# ts to make exactly K moves. The rows and columns are 0 indexed, so the top-left
# square is (0, 0), and the bottom-right square is (N-1, N-1).
# Each time the knight is to move, it chooses one of eight possible moves uniforml
# y at random (even if the piece would go off the chessboard) and moves there.
# The knight continues moving until it has made exactly K moves or has moved off t
# he chessboard. Return the probability that the knight remains on the board after
#  it has stopped moving.
# A chess knight has 8 possible moves it can make, as illustrated below. Each move
#  is two squares in a cardinal direction, then one square in an orthogonal direct
# ion.
# Note:
# Example:

# NOTE
# K will be between 0 and 100.
# The knight always initially starts on the board.
# N will be between 1 and 25.

# EXAMPLE
# Input: 3, 2, 0, 0
# Output: 0.0625
# Explanation: There are two moves (to (1,2), (2,
# 1)) that will keep the knight on the board.
# From each of those positions, there
# are also two moves that will keep the knight on the board.
# The total probability
#  the knight stays on the board is 0.0625.

# Time:  O(k * n^2)
# Space: O(n^2)

class Solution(object):
    def knightProbability(self, N, K, r, c):
        """
        :type N: int
        :type K: int
        :type r: int
        :type c: int
        :rtype: float
        """
        directions = \
            [[ 1, 2], [ 1, -2], [ 2, 1], [ 2, -1], \
             [-1, 2], [-1, -2], [-2, 1], [-2, -1]]
        dp = [[[1 for _ in xrange(N)] for _ in xrange(N)] for _ in xrange(2)]
        for step in xrange(1, K+1):
            for i in xrange(N):
                for j in xrange(N):
                    dp[step%2][i][j] = 0
                    for direction in directions:
                        rr, cc = i+direction[0], j+direction[1]
                        if 0 <= cc < N and 0 <= rr < N:
                            dp[step%2][i][j] += 0.125 * dp[(step-1)%2][rr][cc]

        return dp[K%2][r][c]
```

bulb-switcher-iii.py

```
# bulb-switcher-iii is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numTimesAllBlue(self, light):
        """
        :type light: List[int]
        :rtype: int
        """
        result, right = 0, 0
        for i, num in enumerate(light, 1):
            right = max(right, num)
            result += (right == i)
        return result
```

find-numbers-with-even-number-of-digits.py

```python
# find-numbers-with-even-number-of-digits is not found.
# Time:  O(nlog(logm)), n the length of nums, m is the max value of nums
# Space: O(logm)

import bisect


class Solution(object):
    def __init__(self):
        M = 10**5
        self.__lookup = [0]
        i = 10
        while i < M:
            self.__lookup.append(i)
            i *= 10
        self.__lookup.append(i)

    def findNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def digit_count(n):
            return bisect.bisect_right(self.__lookup, n)

        return sum(digit_count(n) % 2 == 0 for n in nums)


# Time:  O(nlogm), n the length of nums, m is the max value of nums
# Space: O(logm)
class Solution2(object):
    def findNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def digit_count(n):
            result = 0
            while n:
                n //= 10
                result += 1
            return result

        return sum(digit_count(n) % 2 == 0 for n in nums)


# Time:  O(nlogm), n the length of nums, m is the max value of nums
# Space: O(logm)
class Solution3(object):
    def findNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return sum(len(str(n)) % 2 == 0 for n in nums)
```

find-words-that-can-be-formed-by-characters.py

```
# DESC
# Example 2:
# Note:
# Return the sum of lengths of all good strings in words.
# chars
# A string is good if it can be formed by characters from chars (each character ca
# n only be used once).
# You are given an array of strings words and a string chars.
# Example 1:

# NOTE
# 1 <= words[i].length, chars.length <= 100
# All strings contain lowercase English letters only.
# 1 <= words.length <= 1000

# EXAMPLE
# Input: words = ["hello","world","leetcode"], chars = "welldonehoneyr"
# Output: 10
#
# Explanation:
# The strings that can be formed are "hello" and "world" so the ans
# wer is 5 + 5 = 10.
# Input: words = ["cat","bt","hat","tree"], chars = "atach"
# Output: 6
# Explanation:
#
# The strings that can be formed are "cat" and "hat" so the answer is 3 + 3 = 6.

# Time:  O(m * n), m is the length of chars, n is the number of words
# Space: O(1)

import collections


class Solution(object):
    def countCharacters(self, words, chars):
        """
        :type words: List[str]
        :type chars: str
        :rtype: int
        """
        def check(word, chars, count):
            if len(word) > len(chars):
                return False
            curr_count = collections.Counter()
            for c in word:
                curr_count[c] += 1
                if c not in count or count[c] < curr_count[c]:
                    return False
            return True

        count = collections.Counter(chars)
        return sum(len(word) for word in words if check(word, chars, count))
```

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

```
# DESC
# 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.
# Example 1:
# (A node A is an ancestor of B if either: any child of A is equal to B, or any ch
# ild of A is an ancestor of B.)
# Note:

# NOTE
# Each node will have value between 0 and 100000.
# The number of nodes in the tree is between 2 and 5000.

# EXAMPLE
# Input: [8,3,10,1,6,null,14,null,null,4,7,13]
# Output: 7
# Explanation:
# We have var
# ious 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 va
# lue of 7 is obtained by |8 - 1| = 7.

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

```python
# 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"))
```

tweet-counts-per-frequency.py

```
# tweet-counts-per-frequenc is not found.
# Time:  add:   O(logn),
#        query: O(c), c is the total count of matching records
# Space: O(n)

import collections
import random


# Template:
# https://github.com/kamyu104/LeetCode-Solutions/blob/master/Python/design-skiplist.py
class SkipNode(object):
    def __init__(self, level=0, val=None):
        self.val = val
        self.nexts = [None]*level
        self.prevs = [None]*level

class SkipList(object):
    P_NUMERATOR, P_DENOMINATOR = 1, 2  # P = 1/4 in redis implementation
    MAX_LEVEL = 32  # enough for 2^32 elements

    def __init__(self, end=float("inf"), can_duplicated=False):
        random.seed(0)
        self.__head = SkipNode()
        self.__len = 0
        self.__can_duplicated = can_duplicated
        self.add(end)

    def lower_bound(self, target):
        return self.__lower_bound(target, self.__find_prev_nodes(target))

    def find(self, target):
        return self.__find(target, self.__find_prev_nodes(target))

    def add(self, val):
        if not self.__can_duplicated and self.find(val):
            return False
        node = SkipNode(self.__random_level(), val)
        if len(self.__head.nexts) < len(node.nexts):
            self.__head.nexts.extend([None]*(len(node.nexts)-len(self.__head.nexts)))
        prevs = self.__find_prev_nodes(val)
        for i in xrange(len(node.nexts)):
            node.nexts[i] = prevs[i].nexts[i]
            if prevs[i].nexts[i]:
                prevs[i].nexts[i].prevs[i] = node
            prevs[i].nexts[i] = node
            node.prevs[i] = prevs[i]
        self.__len += 1
        return True

    def remove(self, val):
        prevs = self.__find_prev_nodes(val)
        curr = self.__find(val, prevs)
        if not curr:
            return False
        self.__len -= 1
        for i in reversed(xrange(len(curr.nexts))):
            prevs[i].nexts[i] = curr.nexts[i]
```

```
            if curr.nexts[i]:
                curr.nexts[i].prevs[i] = prevs[i]
            if not self.__head.nexts[i]:
                self.__head.nexts.pop()
        return True

    def __lower_bound(self, val, prevs):
        if prevs:
            candidate = prevs[0].nexts[0]
            if candidate:
                return candidate
        return None

    def __find(self, val, prevs):
        candidate = self.__lower_bound(val, prevs)
        if candidate and candidate.val == val:
            return candidate
        return None

    def __find_prev_nodes(self, val):
        prevs = [None]*len(self.__head.nexts)
        curr = self.__head
        for i in reversed(xrange(len(self.__head.nexts))):
            while curr.nexts[i] and curr.nexts[i].val < val:
                curr = curr.nexts[i]
            prevs[i] = curr
        return prevs

    def __random_level(self):
        level = 1
        while random.randint(1, SkipList.P_DENOMINATOR) <= SkipList.P_NUMERATOR and \
              level < SkipList.MAX_LEVEL:
            level += 1
        return level

    def __len__(self):
        return self.__len-1  # excluding end node

    def __str__(self):
        result = []
        for i in reversed(xrange(len(self.__head.nexts))):
            result.append([])
            curr = self.__head.nexts[i]
            while curr:
                result[-1].append(str(curr.val))
                curr = curr.nexts[i]
        return "\n".join(map(lambda x: "->".join(x), result))


class TweetCounts(object):

    def __init__(self):
        self.__records = collections.defaultdict(lambda: SkipList(can_duplicated=True))
        self.__lookup = {"minute":60, "hour":3600, "day":86400}

    def recordTweet(self, tweetName, time):
        """
        :type tweetName: str
        :type time: int
        :rtype: None
```

```
        """
        self.__records[tweetName].add(time)

    def getTweetCountsPerFrequency(self, freq, tweetName, startTime, endTime):
        """
        :type freq: str
        :type tweetName: str
        :type startTime: int
        :type endTime: int
        :rtype: List[int]
        """
        delta = self.__lookup[freq]
        result = [0]*((endTime- startTime)//delta+1)
        it = self.__records[tweetName].lower_bound(startTime)
        while it is not None and it.val <= endTime:
            result[(it.val-startTime)//delta] += 1
            it = it.nexts[0]
        return result


# Time:  add:   O(n),
#        query: O(rlogn), r is the size of result
# Space: O(n)
import bisect
class TweetCounts2(object):

    def __init__(self):
        self.__records = collections.defaultdict(list)
        self.__lookup = {"minute":60, "hour":3600, "day":86400}

    def recordTweet(self, tweetName, time):
        """
        :type tweetName: str
        :type time: int
        :rtype: None
        """
        bisect.insort(self.__records[tweetName], time)

    def getTweetCountsPerFrequency(self, freq, tweetName, startTime, endTime):
        """
        :type freq: str
        :type tweetName: str
        :type startTime: int
        :type endTime: int
        :rtype: List[int]
        """
        delta = self.__lookup[freq]
        i = startTime
        result = []
        while i <= endTime:
            j = min(i+delta, endTime+1)
            result.append(bisect.bisect_left(self.__records[tweetName], j) - \
                          bisect.bisect_left(self.__records[tweetName], i))
            i += delta
        return result


# Time:  add:   O(1),
#        query: O(n)
# Space: O(n)
```

```
class TweetCounts3(object):

    def __init__(self):
        self.__records = collections.defaultdict(list)
        self.__lookup = {"minute":60, "hour":3600, "day":86400}

    def recordTweet(self, tweetName, time):
        """
        :type tweetName: str
        :type time: int
        :rtype: None
        """
        self.__records[tweetName].append(time)

    def getTweetCountsPerFrequency(self, freq, tweetName, startTime, endTime):
        """
        :type freq: str
        :type tweetName: str
        :type startTime: int
        :type endTime: int
        :rtype: List[int]
        """
        delta = self.__lookup[freq]
        result = [0]*((endTime- startTime)//delta+1)
        for t in self.__records[tweetName]:
            if startTime <= t <= endTime:
                result[(t-startTime)//delta] += 1
        return result
```

maximum-product-of-word-lengths.py

```
# DESC
# Example 2:
# Example 1:
# Constraints:
# Example 3:
# Given a string array words, find the maximum value of length(word[i]) * length(w
# ord[j]) where the two words do not share common letters. You may assume that eac
# h word will contain only lower case letters. If no such two words exist, return
# 0.

# NOTE
# 0 <= words[i].length <= 10^3
# words[i] consists only of lowercase English letters.
# 0 <= words.length <= 10^3

# EXAMPLE
# Input: ["a","aa","aaa","aaaa"]
# Output: 0
# Explanation: No such pair of words.
# Input: ["a","ab","abc","d","cd","bcd","abcd"]
# Output: 4
# Explanation: The two wo
# rds can be "ab", "cd".
# Input: ["abcw","baz","foo","bar","xtfn","abcdef"]
# Output: 16
# Explanation: The t
# wo words can be "abcw", "xtfn".

# 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):
            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
# DESC
# Given a sorted linked list, delete all nodes that have duplicate numbers, leavin
# g only distinct numbers from the original list.
# Example 2:
# Example 1:
# Return the linked list sorted as well.

# NOTE
#

# EXAMPLE
# Input: 1->2->3->3->4->4->5
# Output: 1->2->5
# 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
```

lucky-numbers-in-a-matrix.py

```
# lucky-numbers-in-a-matrix is not found.
# Time:  O(m * n)
# Space: O(m + n)

import itertools


class Solution(object):
    def luckyNumbers (self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        rows = map(min, matrix)
        cols = map(max, itertools.izip(*matrix))
        return [cell for i, row in enumerate(matrix)
                     for j, cell in enumerate(row) if rows[i] == cols[j]]


# Time:  O(m * n)
# Space: O(m + n)
import itertools


class Solution2(object):
    def luckyNumbers (self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        return list(set(map(min, matrix)) &
                    set(map(max, itertools.izip(*matrix))))
```

numbers-with-repeated-digits.py

```
# DESC
# Given a positive integer N, return the number of positive integers less than or
# equal to N that have at least 1 repeated digit.
# Note:
# Example 3:
# Example 1:
# Example 2:

# NOTE
# 1 <= N <= 10^9

# EXAMPLE
# Input: 100
# Output: 10
# Explanation: The positive numbers (<= 100) with atleast 1
# repeated digit are 11, 22, 33, 44, 55, 66, 77, 88, 99, and 100.
# Input: 20
# Output: 1
# Explanation: The only positive number (<= 20) with at least
# 1 repeated digit is 11.
# Input: 1000
# Output: 262

# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def numDupDigitsAtMostN(self, N):
        """
        :type N: int
        :rtype: int
        """
        def P(m, n):
            result = 1
            while n > 0:
                result *= m-n+1
                n -= 1
            return result

        digits = map(int, str(N+1))
        result = 0

        # Given 321
        #
        # 1. count numbers without repeated digits:
        # - X
        # - XX
        for i in xrange(1, len(digits)):
            result += P(9, 1)*P(9, i-1)

        # 2. count numbers without repeated digits:
        # - 1XX ~ 3XX
        # - 30X ~ 32X
        # - 320 ~ 321
        prefix_set = set()
        for i, x in enumerate(digits):
            for y in xrange(1 if i == 0 else 0, x):
                if y in prefix_set:
```

```
            continue
        result += P(9-i, len(digits)-i-1)
    if x in prefix_set:
        break
    prefix_set.add(x)
return N-result
```

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

```python
# longest-arithmetic-subsequence-of-given-difference is not found.
# 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
```

maximum-number-of-vowels-in-a-substring-of-given-length.py

```python
# maximum-number-of-vowels-in-a-substring-of-given-length is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxVowels(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        VOWELS = set("aeiou")
        result = curr = 0
        for i, c in enumerate(s):
            curr += c in VOWELS
            if i >= k:
                curr -= s[i-k] in VOWELS
            result = max(result, curr)
        return result
```

sum-of-two-integers.py

```
# DESC
# Example 2:
# Example 1:
# Calculate the sum of two integers a and b, but you are not allowed to use the op
# erator + and -.

# NOTE
#

# EXAMPLE
# Input: a = -2, b = 3
# Output: 1
# Input: a = 1, b = 2
# Output: 3

# 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)
        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
```

xor-operation-in-an-array.py

```
# xor-operation-in-an-arra is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def xorOperation(self, n, start):
        """
        :type n: int
        :type start: int
        :rtype: int
        """
        def xorNums(n, start):
            def xorNumsBeginEven(n, start):
                assert(start%2 == 0)
                # 2*i ^ (2*i+1) = 1
                return ((n//2)%2)^((start+n-1) if n%2 else 0)

            return start^xorNumsBeginEven(n-1, start+1) if start%2 else xorNumsBeginEven(n, start)

        return int(n%2 and start%2) + 2*xorNums(n, start//2)


# Time:  O(n)
# Space: O(1)
import operator


class Solution2(object):
    def xorOperation(self, n, start):
        """
        :type n: int
        :type start: int
        :rtype: int
        """
        return reduce(operator.xor, (i for i in xrange(start, start+2*n, 2)))
```

partition-equal-subset-sum.py

```
# DESC
# Note:
# 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 i
# s equal.
# Example 1:
# Example 2:

# NOTE
# Each of the array element will not exceed 100.
# The array size will not exceed 200.

# EXAMPLE
# Input: [1, 2, 3, 5]
#
# Output: false
#
# Explanation: The array cannot be partitioned
#  into equal sum subsets.
# Input: [1, 5, 11, 5]
#
# Output: true
#
# Explanation: The array can be partitioned as
#  [1, 5, 5] and [11].

# 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

```
# DESC
# Note:
# Example 3:
# Example 1:
# On an infinite number line, the position of the i-th stone is given by stones[i]
# .  Call a stone an endpoint stone if it has the smallest or largest position.
# In particular, if the stones are at say, stones = [1,2,5], you cannot move the e
# ndpoint stone at position 5, since moving it to any position (such as 0, or 3) w
# ill still keep that stone as an endpoint stone.
# The game ends when you cannot make any more moves, ie. the stones are in consecu
# tive positions.
# When the game ends, what is the minimum and maximum number of moves that you cou
# ld have made?  Return the answer as an length 2 array: answer = [minimum_moves,
# maximum_moves]
# Each turn, you pick up an endpoint stone and move it to an unoccupied position s
# o that it is no longer an endpoint stone.
# Example 2:

# NOTE
# stones[i] have distinct values.
# 1 <= stones[i] <= 10^9
# 3 <= stones.length <= 10^4

# EXAMPLE
# Input: [7,4,9]
# Output: [1,2]
# Explanation:
# We can move 4 -> 8 for one move to fi
# nish the game.
# Or, we can move 9 -> 5, 4 -> 6 for two moves to finish the game.
# 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 canno
# t move 10 -> 2 to finish the game, because that would be an illegal move.
# 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

```
# DESC
# Example 1:
# Constraints:
# We have a list of flight bookings.  The i-th booking bookings[i] = [i, j, k] mea
# ns that we booked k seats from flights labeled i to j inclusive.
# There are n flights, and they are labeled from 1 to n.
# Return an array answer of length n, representing the number of seats booked on e
# ach flight in order of their label.

# NOTE
# 1 <= bookings[i][2] <= 10000
# 1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
# 1 <= bookings.length <= 20000

# EXAMPLE
# Input: bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
# Output: [10,55,45,25,25]

# 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
```

text-justification.py

```
# DESC
# Example 3:
# Example 1:
# Given an array of words and a width maxWidth, format the text such that each lin
# e has exactly maxWidth characters and is fully (left and right) justified.
# Example 2:
# Extra spaces between words should be distributed as evenly as possible. If the n
# umber of spaces on a line do not divide evenly between words, the empty slots on
#  the left will be assigned more spaces than the slots on the right.
# You should pack your words in a greedy approach; that is, pack as many words as
# you can in each line. Pad extra spaces ' ' when necessary so that each line has
# exactly maxWidth characters.
# Note:
# For the last line of text, it should be left justified and no extra space is ins
# erted between words.

# NOTE
# The input array words contains at least one word.
# A word is defined as a character sequence consisting of non-space characters only.
# Each word's length is guaranteed to be greater than 0 and not exceed maxWidth.

# EXAMPLE
# Input:
# words = ["This", "is", "an", "example", "of", "text", "justification."]
# m
# axWidth = 16
# Output:
# [
#    "This    is    an",
#    "example  of text",
#    "justifi
# cation.  "
# ]
# Input:
# words = ["What","must","be","acknowledgment","shall","be"]
# maxWidth = 16
#
# Output:
# [
#   "What   must   be",
#   "acknowledgment  ",
#   "shall be        "
# ]
# Exp
# lanation: Note that the last line is "shall be    " instead of "shall     be",
#
#             because the last line must be left-justified instead of fully-justif
# ied.
#              Note that the second line is also left-justified becase it con
# tains only one word.
# Input:
# words = ["Science","is","what","we","understand","well","enough","to","ex
# plain",
#          "to","a","computer.","Art","is","everything","else","we","do"]
#
# maxWidth = 20
# Output:
# [
```

```
#   "Science  is  what we",
#   "understand      well",
#   "e
# nough to explain to",
#   "a  computer.  Art is",
#   "everything  else  we",
#   "do
#                 "
# ]

# Time:  O(n)
# Space: O(k), k is maxWidth.

class Solution(object):
    def fullJustify(self, words, maxWidth):
        """
        :type words: List[str]
        :type maxWidth: int
        :rtype: List[str]
        """
        def addSpaces(i, spaceCnt, maxWidth, is_last):
            if i < spaceCnt:
                # For the last line of text, it should be left justified,
                # and no extra space is inserted between words.
                return 1 if is_last else (maxWidth // spaceCnt) + int(i < maxWidth % spaceCnt)
            return 0

        def connect(words, maxWidth, begin, end, length, is_last):
            s = []  # The extra space O(k) is spent here.
            n = end - begin
            for i in xrange(n):
                s += words[begin + i],
                s += ' ' * addSpaces(i, n - 1, maxWidth - length, is_last),
            # For only one word in a line.
            line = "".join(s)
            if len(line) < maxWidth:
                line += ' ' * (maxWidth - len(line))
            return line

        res = []
        begin, length = 0, 0
        for i in xrange(len(words)):
            if length + len(words[i]) + (i - begin) > maxWidth:
                res += connect(words, maxWidth, begin, i, length, False),
                begin, length = i, 0
            length += len(words[i])

        # Last line.
        res += connect(words, maxWidth, begin, len(words), length, True),
        return res
```

pairs-of-songs-with-total-durations-divisible-by-60.py

```
# DESC
# Example 1:
# Return the number of pairs of songs for which their total duration in seconds is
#  divisible by 60.  Formally, we want the number of indices i, j such that i < j
# with (time[i] + time[j]) % 60 == 0.
# In a list of songs, the i-th song has a duration of time[i] seconds.
# Example 2:
# Note:

# NOTE
# 1 <= time.length <= 60000
# 1 <= time[i] <= 500

# EXAMPLE
# Input: [60,60,60]
# Output: 3
# Explanation: All three pairs have a total duration o
# f 120, which is divisible by 60.
# Input: [30,20,150,100,40]
# Output: 3
# Explanation: Three pairs have a total durati
# on divisible by 60:
# (time[0] = 30, time[2] = 150): total duration 180
# (time[1] =
#  20, time[3] = 100): total duration 120
# (time[1] = 20, time[4] = 40): total dura
# tion 60

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def numPairsDivisibleBy60(self, time):
        """
        :type time: List[int]
        :rtype: int
        """
        result = 0
        count = collections.Counter()
        for t in time:
            result += count[-t%60]
            count[t%60] += 1
        return result
```

jewels-and-stones.py

```
# DESC
# Example 2:
# You're given strings J representing the types of stones that are jewels, and S r
# epresenting the stones you have.  Each character in S is a type of stone you hav
# e.  You want to know how many of the stones you have are also jewels.
# Example 1:
# The letters in J are guaranteed distinct, and all characters in J and S are lett
# ers. Letters are case sensitive, so "a" is considered a different type of stone
# from "A".
# Note:

# NOTE
# S and J will consist of letters and have length at most 50.
# The characters in J are distinct.

# EXAMPLE
# Input: J = "aA", S = "aAAbbbb"
# Output: 3
# Input: J = "z", S = "ZZ"
# Output: 0

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

class Solution(object):
    def numJewelsInStones(self, J, S):
        """
        :type J: str
        :type S: str
        :rtype: int
        """
        lookup = set(J)
        return sum(s in lookup for s in S)
```

output-contest-matches.py

```python
# output-contest-matches is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def findContestMatch(self, n):
        """
        :type n: int
        :rtype: str
        """
        matches = map(str, range(1, n+1))
        while len(matches)/2:
            matches = ["({},{})".format(matches[i], matches[-i-1]) for i in xrange(len(matches)/2)]
        return matches[0]
```

k-empty-slots.py

```python
# k-empty-slots is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def kEmptySlots(self, flowers, k):
        """
        :type flowers: List[int]
        :type k: int
        :rtype: int
        """
        days = [0] * len(flowers)
        for i in xrange(len(flowers)):
            days[flowers[i]-1] = i
        result = float("inf")
        i, left, right = 0, 0, k+1
        while right < len(days):
            if days[i] < days[left] or days[i] <= days[right]:
                if i == right:
                    result = min(result, max(days[left], days[right]))
                left, right = i, k+1+i
            i += 1
        return -1 if result == float("inf") else result+1
```

image-overlap.py

```
# image-overla is not found.
# Time:  O(n^4)
# Space: O(n^2)

class Solution(object):
    def largestOverlap(self, A, B):
        """
        :type A: List[List[int]]
        :type B: List[List[int]]
        :rtype: int
        """
        count = [0] * (2*len(A)-1)**2
        for i, row in enumerate(A):
            for j, v in enumerate(row):
                if not v:
                    continue
                for i2, row2 in enumerate(B):
                    for j2, v2 in enumerate(row2):
                        if not v2:
                            continue
                        count[(len(A)-1+i-i2)*(2*len(A)-1) +
                              len(A)-1+j-j2] += 1
        return max(count)
```

decompress-run-length-encoded-list.py

```
# DESC
# Return the decompressed list.
# [freq, val] = [nums[2*i], nums[2*i+1]]
# Constraints:
# Example 1:
# Consider each adjacent pair of elements [freq, val] = [nums[2*i], nums[2*i+1]] (
# with i >= 0).  For each such pair, there are freq elements with value val concat
# enated in a sublist. Concatenate all the sublists from left to right to generate
#  the decompressed list.
# We are given a list nums of integers representing a list compressed with run-len
# gth encoding.
# Example 2:

# NOTE
# 2 <= nums.length <= 100
# 1 <= nums[i] <= 100
# nums.length % 2 == 0

# EXAMPLE
# Input: nums = [1,1,2,3]
# Output: [1,3,3]
# Input: nums = [1,2,3,4]
# Output: [2,4,4,4]
# Explanation: The first pair [1,2] mean
# s we have freq = 1 and val = 2 so we generate the array [2].
# The second pair [3,
# 4] means we have freq = 3 and val = 4 so we generate [4,4,4].
# At the end the con
# catenation [2] + [4,4,4] is [2,4,4,4].

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def decompressRLElist(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        return [nums[i+1] for i in xrange(0, len(nums), 2) for _ in xrange(nums[i])]
```

print-binary-tree.py

```
# DESC
# Print a binary tree in an m*n 2D string array following these rules:
# Example 2:
# Example 1:
# Example 3:
# Note:
# The height of binary tree is in the range of [1, 10].

# NOTE
# Print the subtrees following the same rules.
# The column number n should always be an odd number.
# The row number m should be equal to the height of the given binary tree.
# 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 p
# art). You should print the left subtree in the left-bottom part and print the ri
# ght 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 n
# ot, you don't need to print anything for the none subtree but still need to leav
# e 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 "".

# EXAMPLE
# Input:
#      1
#     / \
#    2   3
#     \
#      4
# Output:
# [["", "", "", "1", "", "", "
# "],
#  ["", "2", "", "", "", "3", ""],
#  ["", "", "4", "", "", "", ""]]
# Input:
#      1
#     /
#    2
# Output:
# [["", "1", ""],
#  ["2", "", ""]]
# Input:
#       1
#      / \
#     2   5
#    /
#   3
#  /
# 4
# Output:
#
# [["",  "",  "", "",
#   "", "", "", "1", "",  "",  "",  "",  "", "", ""]
#  ["",  "",  "", "2", "", "",
# "", "",  "",  "",  "",  "5", "", "", ""]
#  ["",  "3", "", "",  "", "", "", "",  "
# ",  "",  "",  "",  "", "", ""]
```

```
#  ["4", "",  "", "",  "", "", "", "",  "",  "",  "
# ",  "",  "", "", ""]]

# 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
```

projection-area-of-3d-shapes.py

```
# DESC
# On a N * N grid, we place some 1 * 1 * 1 cubes that are axis-aligned with the x,
#  y, and z axes.
# Note:
# A projection is like a shadow, that maps our 3 dimensional figure to a 2 dimensi
# onal plane.
# Example 3:
# Example 5:
# Each value v = grid[i][j] represents a tower of v cubes placed on top of grid ce
# ll (i, j).
# Example 2:
# v = grid[i][j]
# Now we view the projection of these cubes onto the xy, yz, and zx planes.
# Example 1:
# Return the total area of all three projections.
# Here, we are viewing the "shadow" when looking at the cubes from the top, the fr
# ont, and the side.
# Example 4:

# NOTE
# 0 <= grid[i][j] <= 50
# 1 <= grid.length = grid[0].length <= 50

# EXAMPLE
# Input: [[1,0],[0,2]]
# Output: 8
# Input: [[2,2,2],[2,1,2],[2,2,2]]
# Output: 21
# Input: [[1,1,1],[1,0,1],[1,1,1]]
# Output: 14
# Input: [[1,2],[3,4]]
# Output: 17
# Explanation:
# Here are the three projections ("s
# hadows") of the shape made with each axis-aligned plane.
# Input: [[2]]
# Output: 5

# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def projectionArea(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        result = 0
        for i in xrange(len(grid)):
            max_row, max_col = 0, 0
            for j in xrange(len(grid)):
                if grid[i][j]:
                    result += 1
                max_row = max(max_row, grid[i][j])
                max_col = max(max_col, grid[j][i])
            result += max_row + max_col
        return result
```

subtree-of-another-tree.py

```
# DESC
# Example 2:
#
# Given tree s:
# Example 1:
#
# Given tree s:
# Given two non-empty binary trees s and t, check whether tree t has exactly the s
# ame structure and node values with a subtree of s. A subtree of s is a tree cons
# ists of a node in s and all of this node's descendants. The tree s could also be
#  considered as a subtree of itself.

# NOTE
#

# EXAMPLE
# 4
#   / \
#  1   2
# 4
#   / \
#  1   2
# 3
#    / \
#   4   5
#  / \
#  1   2
# 3
#    / \
#   4   5
#  / \
#  1   2
#    /
#   0

# Time:  O(m * n), m is the number of nodes of s, n is the number of nodes of t
# Space: O(h), h is the height of s

class Solution(object):
    def isSubtree(self, s, t):
        """
        :type s: TreeNode
        :type t: TreeNode
        :rtype: bool
        """
        def isSame(x, y):
            if not x and not y:
                return True
            if not x or not y:
                return False
            return x.val == y.val and \
                   isSame(x.left, y.left) and \
                   isSame(x.right, y.right)

        def preOrderTraverse(s, t):
            return s != None and \
                   (isSame(s, t) or \
                    preOrderTraverse(s.left, t) or \
```

```
            preOrderTraverse(s.right, t))

    return preOrderTraverse(s, t)
```

find-leaves-of-binary-tree.py

```
# find-leaves-of-binary-tree is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def findLeaves(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        def findLeavesHelper(node, result):
            if not node:
                return -1
            level = 1 + max(findLeavesHelper(node.left, result), \
                            findLeavesHelper(node.right, result))
            if len(result) < level + 1:
                result.append([])
            result[level].append(node.val)
            return level

        result = []
        findLeavesHelper(root, result)
        return result
```

verifying-an-alien-dictionary.py

```
# verifying-an-alien-dictionar is not found.
# Time:  O(n * l), l is the average length of words
# Space: O(1)

class Solution(object):
    def isAlienSorted(self, words, order):
        """
        :type words: List[str]
        :type order: str
        :rtype: bool
        """
        lookup = {c: i for i, c in enumerate(order)}
        for i in xrange(len(words)-1):
            word1 = words[i]
            word2 = words[i+1]
            for k in xrange(min(len(word1), len(word2))):
                if word1[k] != word2[k]:
                    if lookup[word1[k]] > lookup[word2[k]]:
                        return False
                    break
            else:
                if len(word1) > len(word2):
                    return False
        return True
```

count-number-of-teams.py

```
# count-number-of-teams is not found.
# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def numTeams(self, rating):
        """
        :type rating: List[int]
        :rtype: int
        """
        result = 0
        for i in xrange(1, len(rating)-1):
            less, greater = [0]*2, [0]*2
            for j in xrange(len(rating)):
                if rating[i] > rating[j]:
                    less[i < j] += 1
                if rating[i] < rating[j]:
                    greater[i < j] += 1
            result += less[0]*greater[1] + greater[0]*less[1]
        return result
```

paint-house.py

```
# paint-house is not found.
# Time:  O(n)
# Space: O(1)

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

        min_cost = [costs[0], [0, 0, 0]]

        n = len(costs)
        for i in xrange(1, n):
            min_cost[i % 2][0] = costs[i][0] + \
                                 min(min_cost[(i - 1) % 2][1], min_cost[(i - 1) % 2][2])
            min_cost[i % 2][1] = costs[i][1] + \
                                 min(min_cost[(i - 1) % 2][0], min_cost[(i - 1) % 2][2])
            min_cost[i % 2][2] = costs[i][2] + \
                                 min(min_cost[(i - 1) % 2][0], min_cost[(i - 1) % 2][1])

        return min(min_cost[(n - 1) % 2])

# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def minCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        if not costs:
            return 0

        n = len(costs)
        for i in xrange(1, n):
            costs[i][0] += min(costs[i - 1][1], costs[i - 1][2])
            costs[i][1] += min(costs[i - 1][0], costs[i - 1][2])
            costs[i][2] += min(costs[i - 1][0], costs[i - 1][1])

        return min(costs[n - 1])
```

divide-array-into-increasing-sequences.py

```python
# divide-array-into-increasing-sequences is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canDivideIntoSubsequences(self, nums, K):
        """
        :type nums: List[int]
        :type K: int
        :rtype: bool
        """
        curr, max_count = 1, 1
        for i in xrange(1, len(nums)):
            curr = 1 if nums[i-1] < nums[i] else curr+1
            max_count = max(max_count, curr)
        return K*max_count <= len(nums)
```

count-different-palindromic-subsequences.py

```
# DESC
# A sequence is palindromic if it is equal to the sequence reversed.
# Example 1:
# A subsequence of a string S is obtained by deleting 0 or more characters from S.
# Two sequences A_1, A_2, ... and B_1, B_2, ... are different if there is some i f
# or which A_i != B_i.
# Note:
# Example 2:
# Given a string S, find the number of different non-empty palindromic subsequence
# s in S, and return that number modulo 10^9 + 7.

# NOTE
# The length of S will be in the range [1, 1000].
# Each character S[i] will be in the set {'a', 'b', 'c', 'd'}.

# EXAMPLE
# Input:
# S = 'bccb'
# Output: 6
# Explanation:
# The 6 different non-empty palindromic
#  subsequences are 'b', 'c', 'bb', 'cc', 'bcb', 'bccb'.
# Note that 'bcb' is counte
# d only once, even though it occurs twice.
# Input:
# S = 'abcdabcdabcdabcdabcdabcdabcddcbadcbadcbadcbadcbadcbadcbadcba'
# 0
# utput: 104860361
# Explanation:
# There are 3104860382 different non-empty palindro
# mic subsequences, which is 104860361 modulo 10^9 + 7.

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

class Solution(object):
    def countPalindromicSubsequences(self, S):
        """
        :type S: str
        :rtype: int
        """
        def dp(i, j, prv, nxt, lookup):
            if lookup[i][j] is not None:
                return lookup[i][j]
            result = 1
            if i <= j:
                for x in xrange(4):
                    i0 = nxt[i][x]
                    j0 = prv[j][x]
                    if i <= i0 <= j:
                        result = (result + 1) % P
                    if None < i0 < j0:
                        result = (result + dp(i0+1, j0-1, prv, nxt, lookup)) % P
            result %= P
            lookup[i][j] = result
            return result

        prv = [None] * len(S)
```

```
nxt = [None] * len(S)

last = [None] * 4
for i in xrange(len(S)):
    last[ord(S[i])-ord('a')] = i
    prv[i] = tuple(last)

last = [None] * 4
for i in reversed(xrange(len(S))):
    last[ord(S[i])-ord('a')] = i
    nxt[i] = tuple(last)

P = 10**9 + 7
lookup = [[None] * len(S) for _ in xrange(len(S))]
return dp(0, len(S)-1, prv, nxt, lookup) - 1
```

making-a-large-island.py

```
# DESC
# Notes:
# Example 2:
# Example 3:
# In a 2D grid of 0s and 1s, we change at most one 0 to a 1.
# Example 1:
# After, what is the size of the largest island? (An island is a 4-directionally c
# onnected group of 1s).

# NOTE
# 0 <= grid[i][j] <= 1.
# 1 <= grid.length = grid[0].length <= 50.

# EXAMPLE
# Input: [[1, 1], [1, 1]]
# Output: 4
# Explanation: Can't change any 0 to 1, only one
#  island with area = 4.
# Input: [[1, 1], [1, 0]]
# Output: 4
# Explanation: Change the 0 to 1 and make the is
# land bigger, only one island with area = 4.
# Input: [[1, 0], [0, 1]]
# Output: 3
# Explanation: Change one 0 to 1 and connect two
#  1s, then we get an island with area = 3.

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


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

        def dfs(r, c, index, grid):
            if not (0 <= r < len(grid) and
                    0 <= c < len(grid[0]) and
                    grid[r][c] == 1):
                return 0
            result = 1
            grid[r][c] = index
            for d in directions:
                result += dfs(r+d[0], c+d[1], index, grid)
            return result

        area = {}
        index = 2
        for r in xrange(len(grid)):
            for c in xrange(len(grid[r])):
                if grid[r][c] == 1:
                    area[index] = dfs(r, c, index, grid)
                    index += 1
```

```
    result = max(area.values() or [0])
    for r in xrange(len(grid)):
        for c in xrange(len(grid[r])):
            if grid[r][c] == 0:
                seen = set()
                for d in directions:
                    nr, nc = r+d[0], c+d[1]
                    if not (0 <= nr < len(grid) and
                            0 <= nc < len(grid[0]) and
                            grid[nr][nc] > 1):
                        continue
                    seen.add(grid[nr][nc])
                result = max(result, 1 + sum(area[i] for i in seen))
    return result
```

least-operators-to-express-number.py

```
# DESC
# Given a single positive integer x, we will write an expression of the form x (op
# 1) x (op2) x (op3) x ... where each operator op1, op2, etc. is either addition,
# subtraction, multiplication, or division (+, -, *, or /).  For example, with x =
#  3, we might write 3 * 3 / 3 + 3 - 3 which is a value of 3.
# Example 1:
# We would like to write an expression with the least number of operators such tha
# t the expression equals the given target.  Return the least number of operators
# used.
# Note:
# Example 2:
# Example 3:
# When writing such an expression, we adhere to the following conventions:

# NOTE
# There are no parentheses placed anywhere.
# The division operator (/) returns rational numbers.
# It's not allowed to use the unary negation operator (-).  For example, "x - x" i
# s a valid expression as it only uses subtraction, but "-x + x" is not because it
#  uses negation.
# 1 <= target <= 2 * 10^8
# We use the usual order of operations: multiplication and division happens before
#  addition and subtraction.
# 2 <= x <= 100

# EXAMPLE
# Input: x = 100, target = 100000000
# Output: 3
# Explanation: 100 * 100 * 100 * 100.
#   The expression contains 3 operations.
# Input: x = 3, target = 19
# Output: 5
# Explanation: 3 * 3 + 3 * 3 + 3 / 3.  The exp
# ression contains 5 operations.
# Input: x = 5, target = 501
# Output: 8
# Explanation: 5 * 5 * 5 * 5 - 5 * 5 * 5 + 5
# / 5.  The expression contains 8 operations.

# Time:  O(logn/logx) = O(1)
# Space: O(logn) = O(1)

class Solution(object):
    def leastOpsExpressTarget(self, x, target):
        """
        :type x: int
        :type target: int
        :rtype: int
        """
        pos, neg, k = 0, 0, 0
        while target:
            target, r = divmod(target, x)
            if k:
                pos, neg = min(r*k + pos, (r+1)*k + neg), \
                           min((x-r)*k + pos, (x-r-1)*k + neg)
            else:
                pos, neg = r*2, (x-r)*2
            k += 1
```

```
    return min(pos, k+neg) - 1
```

linked-list-random-node.py

```
# DESC
# 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?
# Given a singly linked list, return a random node's value from the linked list. E
# ach node must have the same probability of being chosen.
# Example:

# NOTE
#

# 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 So
# lution(head);
#
# // getRandom() should return either 1, 2, or 3 randomly. Each ele
# ment 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

```
# swap-for-longest-repeated-character-substring is not found.
# Time:  O(n)
# Space: O(1)

import collections


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

diameter-of-n-ary-tree.py

```
# diameter-of-n-ary-tree is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a Node.
class Node(object):
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []


class Solution(object):
    def diameter(self, root):
        """
        :type root: 'Node'
        :rtype: int
        """
        def iter_dfs(root):
            result = [0]*2
            stk = [(1, (root, result))]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node, ret = params
                    for child in reversed(node.children):
                        ret2 = [0]*2
                        stk.append((2, (ret2, ret)))
                        stk.append((1, (child, ret2)))
                else:
                    ret2, ret = params
                    ret[0] = max(ret[0], ret2[0], ret[1]+ret2[1]+1)
                    ret[1] = max(ret[1], ret2[1]+1)
            return result

        return iter_dfs(root)[0]


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def diameter(self, root):
        """
        :type root: 'Node'
        :rtype: int
        """
        def dfs(node):
            max_dia, max_depth = 0, 0
            for child in node.children:
                child_max_dia, child_max_depth = dfs(child)
                max_dia = max(max_dia, child_max_dia, max_depth+child_max_depth+1)
                max_depth = max(max_depth, child_max_depth+1)
            return max_dia, max_depth

        return dfs(root)[0]
```

minimum-number-of-days-to-make-m-bouquets.py

```python
# minimum-number-of-days-to-make-m-bouquets is not found.
# Time:  O(nlogd), d is the max day of bloomDay
# Space: O(1)

class Solution(object):
    def minDays(self, bloomDay, m, k):
        """
        :type bloomDay: List[int]
        :type m: int
        :type k: int
        :rtype: int
        """
        def check(bloomDay, m, k, x):
            result = count = 0
            for d in bloomDay:
                count = count+1 if d <= x else 0
                if count == k:
                    count = 0
                    result += 1
                    if result == m:
                        break
            return result >= m

        if m*k > len(bloomDay):
            return -1
        left, right = 1, max(bloomDay)
        while left <= right:
            mid = left + (right-left)//2
            if check(bloomDay, m, k, mid):
                right = mid-1
            else:
                left = mid+1
        return left
```

super-palindromes.py

```
# DESC
# Note:
# Now, given two positive integers L and R (represented as strings), return the nu
# mber of superpalindromes in the inclusive range [L, R].
# Example 1:
# Let's say a positive integer is a superpalindrome if it is a palindrome, and it
# is also the square of a palindrome.

# NOTE
# L and R are strings representing integers in the range [1, 10^18).
# 1 <= len(R) <= 18
# 1 <= len(L) <= 18
# int(L) <= int(R)

# EXAMPLE
# Input: L = "4", R = "1000"
# Output: 4
# Explanation: 4, 9, 121, and 484 are superpa
# lindromes.
# Note that 676 is not a superpalindrome: 26 * 26 = 676, but 26 is not
# a palindrome.

# Time:  O(n^0.25 * logn)
# Space: O(logn)

class Solution(object):
    def superpalindromesInRange(self, L, R):
        """
        :type L: str
        :type R: str
        :rtype: int
        """
        def is_palindrome(k):
            return str(k) == str(k)[::-1]

        K = int((10**((len(R)+1)*0.25)))
        l, r = int(L), int(R)

        result = 0

        # count odd length
        for k in xrange(K):
            s = str(k)
            t = s + s[-2::-1]
            v = int(t)**2
            if v > r:
                break
            if v >= l and is_palindrome(v):
                result += 1

        # count even length
        for k in xrange(K):
            s = str(k)
            t = s + s[::-1]
            v = int(t)**2
            if v > r:
                break
            if v >= l and is_palindrome(v):
```

```
            result += 1

    return result
```

print-words-vertically.py

```python
# print-words-verticall is not found.
# Time:  O(n)
# Space: O(n)

import itertools


class Solution(object):
    def printVertically(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        return ["".join(c).rstrip() for c in itertools.izip_longest(*s.split(), fillvalue=' ')]
```

minimum-difference-between-largest-and-smallest-value-in-three-moves.py

```python
# minimum-difference-between-largest-and-smallest-value-in-three-moves is not found.
# Time:  O(n + klogk)
# Space: O(k)

import random


class Solution(object):
    def minDifference(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def nth_element(nums, left, n, right, compare=lambda a, b: a < b):
            def partition_around_pivot(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

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

        k = 4
        if len(nums) <= k:
            return 0
        nth_element(nums, 0, k, len(nums)-1)
        nums[:k] = sorted(nums[:k])
        nth_element(nums, k, max(k, len(nums)-k), len(nums)-1)
        nums[-k:] = sorted(nums[-k:])
        return min(nums[-k+i]-nums[i] for i in xrange(k))
```

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

```
# DESC
# Formally, return the largest V for which V = (A[i] + A[i+1] + ... + A[i+L-1]) +
# (A[j] + A[j+1] + ... + A[j+M-1]) and either:
# Example 3:
# Note:
# Given an array A of non-negative integers, return the maximum sum of elements in
#  two non-overlapping (contiguous) subarrays, which have lengths L and M.  (For c
# larification, the L-length subarray could occur before or after the M-length sub
# array.)
# Example 1:
# Example 2:

# NOTE
# 0 <= A[i] <= 1000
# 0 <= j < j + M - 1 < i < i + L - 1 < A.length.
# L + M <= A.length <= 1000
# 0 <= i < i + L - 1 < j < j + M - 1 < A.length, or
# L >= 1
# M >= 1

# EXAMPLE
# Input: A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3
# Output: 31
# Explanation: One choic
# e of subarrays is [5,6,0,9] with length 4, and [3,8] with length 3.
# Input: A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2
# Output: 20
# Explanation: One choice
# of subarrays is [9] with length 1, and [6,5] with length 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.

# 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
```

rotate-array.py

```
# rotate-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    """
    :type nums: List[int]
    :type k: int
    :rtype: void Do not return anything, modify nums in-place instead.
    """
    def rotate(self, nums, k):
        def reverse(nums, start, end):
            while start < end:
                nums[start], nums[end - 1] = nums[end - 1], nums[start]
                start += 1
                end -= 1

        k %= len(nums)
        reverse(nums, 0, len(nums))
        reverse(nums, 0, k)
        reverse(nums, k, len(nums))


# Time:  O(n)
# Space: O(1)
from fractions import gcd


class Solution2(object):
    """
    :type nums: List[int]
    :type k: int
    :rtype: void Do not return anything, modify nums in-place instead.
    """

    def rotate(self, nums, k):
        def apply_cycle_permutation(k, offset, cycle_len, nums):
            tmp = nums[offset]
            for i in xrange(1, cycle_len):
                nums[(offset + i * k) % len(nums)], tmp = tmp, nums[(offset + i * k) % len(nums)]
            nums[offset] = tmp

        k %= len(nums)
        num_cycles = gcd(len(nums), k)
        cycle_len = len(nums) / num_cycles
        for i in xrange(num_cycles):
            apply_cycle_permutation(k, i, cycle_len, nums)


# Time:  O(n)
# Space: O(1)
class Solution3(object):
    """
    :type nums: List[int]
    :type k: int
    :rtype: void Do not return anything, modify nums in-place instead.
    """
```

```
    def rotate(self, nums, k):
        count = 0
        start = 0
        while count < len(nums):
            curr = start
            prev = nums[curr]
            while True:
                idx = (curr + k) % len(nums)
                nums[idx], prev = prev, nums[idx]
                curr = idx
                count += 1
                if start == curr:
                    break
            start += 1


# Time:  O(n)
# Space: O(n)
class Solution4(object):
    """
    :type nums: List[int]
    :type k: int
    :rtype: void Do not return anything, modify nums in-place instead.
    """
    def rotate(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        nums[:] = nums[len(nums) - k:] + nums[:len(nums) - k]


# Time:  O(k * n)
# Space: O(1)
class Solution5(object):
    """
    :type nums: List[int]
    :type k: int
    :rtype: void Do not return anything, modify nums in-place instead.
    """
    def rotate(self, nums, k):
        while k > 0:
            nums.insert(0, nums.pop())
            k -= 1
```

binary-tree-level-order-traversal.py

```python
# DESC
# [3,9,20,null,null,15,7]
# Given a binary tree, return the level order traversal of its nodes' values. (ie,
#  from left to right, level by level).
# return its level order traversal as:
# For example:
#
# Given binary tree [3,9,20,null,null,15,7],

# NOTE
#

# EXAMPLE
# [
#   [3],
#   [9,20],
#   [15,7]
# ]
# 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

```
# DESC
# node.left
# Constraints:
# Example 1:
# Return the root node of a binary search tree that matches the given preorder tra
# versal.
# (Recall that a binary search tree is a binary tree where for every node, any des
# cendant of node.left has a value < node.val, and any descendant of node.right ha
# s 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.

# NOTE
# 1 <= preorder.length <= 100
# 1 <= preorder[i] <= 10^8
# The values of preorder are distinct.

# EXAMPLE
# Input: [8,5,1,7,10,12]
# Output: [8,5,10,1,7,null,12]

# 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])
```

number-of-sub-arrays-of-size-k-and-average-greater-than-or-equal-to-threshold.py

```
# number-of-sub-arrays-of-size-k-and-average-greater-than-or-equal-to-threshold is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def numOfSubarrays(self, arr, k, threshold):
        """
        :type arr: List[int]
        :type k: int
        :type threshold: int
        :rtype: int
        """
        result, curr = 0, sum(itertools.islice(arr, 0, k-1))
        for i in xrange(k-1, len(arr)):
            curr += arr[i]-(arr[i-k] if i-k >= 0 else 0)
            result += int(curr >= threshold*k)
        return result


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def numOfSubarrays(self, arr, k, threshold):
        """
        :type arr: List[int]
        :type k: int
        :type threshold: int
        :rtype: int
        """
        accu = [0]
        for x in arr:
            accu.append(accu[-1]+x)
        result = 0
        for i in xrange(len(accu)-k):
            if accu[i+k]-accu[i] >= threshold*k:
                result += 1
        return result
```

group-the-people-given-the-group-size-they-belong-to.py

```python
# group-the-people-given-the-group-size-they-belong-to is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def groupThePeople(self, groupSizes):
        """
        :type groupSizes: List[int]
        :rtype: List[List[int]]
        """
        groups, result = collections.defaultdict(list), []
        for i, size in enumerate(groupSizes):
            groups[size].append(i)
            if len(groups[size]) == size:
                result.append(groups.pop(size))
        return result
```

sum-of-left-leaves.py

```
# DESC
# Example:
# Find the sum of all left leaves in a given binary tree.

# NOTE
#

# EXAMPLE
# 3
#    / \
#   9  20
#     /  \
#    15   7
#
# There are two left leaves in the binary tre
# e, with values 9 and 15 respectively. Return 24.

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

class Solution(object):
    def sumOfLeftLeaves(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def sumOfLeftLeavesHelper(root, is_left):
            if not root:
                return 0
            if not root.left and not root.right:
                return root.val if is_left else 0
            return sumOfLeftLeavesHelper(root.left, True) + \
                   sumOfLeftLeavesHelper(root.right, False)

        return sumOfLeftLeavesHelper(root, False)
```

design-file-system.py

```python
# design-file-system is not found.
# Time:  create: O(n)
#        get:    O(n)
# Space: O(n)

class FileSystem(object):

    def __init__(self):
        self.__lookup = {"": -1}

    def create(self, path, value):
        """
        :type path: str
        :type value: int
        :rtype: bool
        """
        if path[:path.rfind('/')] not in self.__lookup:
            return False
        self.__lookup[path] = value
        return True

    def get(self, path):
        """
        :type path: str
        :rtype: int
        """
        if path not in self.__lookup:
            return -1
        return self.__lookup[path]
```

add-strings.py

```
# DESC
# Given two non-negative integers num1 and num2 represented as string, return the
# sum of num1 and num2.
# Note:

# NOTE
# Both num1 and num2 does not contain any leading zero.
# The length of both num1 and num2 is < 5100.
# Both num1 and num2 contains only digits 0-9.
# You must not use any built-in BigInteger library or convert the inputs to intege
# r directly.

# EXAMPLE
#

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def addStrings(self, num1, num2):
        """
        :type num1: str
        :type num2: str
        :rtype: str
        """
        result = []
        i, j, carry = len(num1) - 1, len(num2) - 1, 0

        while i >= 0 or j >= 0 or carry:
            if i >= 0:
                carry += ord(num1[i]) - ord('0')
                i -= 1
            if j >= 0:
                carry += ord(num2[j]) - ord('0')
                j -= 1
            result.append(str(carry % 10))
            carry /= 10
        result.reverse()

        return "".join(result)

    def addStrings2(self, num1, num2):
        """
        :type num1: str
        :type num2: str
        :rtype: str
        """
        length = max(len(num1), len(num2))
        num1 = num1.zfill(length)[::-1]
        num2 = num2.zfill(length)[::-1]
        res, plus = '', 0
        for index, num in enumerate(num1):
            tmp = str(int(num) + int(num2[index]) + plus)
            res += tmp[-1]
            if int(tmp) > 9:
                plus = 1
            else:
                plus = 0
```

```
if plus:
    res += '1'
return res[::-1]
```

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

```
# DESC
# Example 2:
# Internally you can think of this:
# Do not allocate extra space for another array, you must do this by modifying the
#  input array in-place with O(1) extra memory.
# Given a sorted array nums, remove the duplicates in-place such that duplicates a
# ppeared at most twice and return the new length.
# Clarification:
# Example 1:
# Note that the input array is passed in by reference, which means modification to
#  the input array will be known to the caller as well.
# Confused why the returned value is an integer but your answer is an array?

# NOTE
#

# EXAMPLE
# Given nums = [0,0,1,1,1,1,2,3,3],
#
# Your function should return length = 7, with
# the first seven elements of nums being modified to 0, 0, 1, 1, 2, 3 and 3 respec
# tively.
#
# It doesn't matter what values are set beyond the returned length.
# Given nums = [1,1,1,2,2,3],
#
# Your function should return length = 5, with the fi
# rst five elements of nums being 1, 1, 2, 2 and 3 respectively.
#
# It doesn't matte
# r what you leave beyond the returned length.
# // nums is passed in by reference. (i.e., without making a copy)
# int len = remov
# eDuplicates(nums);
#
# // any modification to nums in your function would be known
# by the caller.
# // using the length returned by your function, it prints the firs
# t len elements.
# for (int i = 0; i < len; i++) {
#     print(nums[i]);
# }

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param a list of integers
    # @return an integer
    def removeDuplicates(self, A):
        if not A:
            return 0

        last, i, same = 0, 1, False
        while i < len(A):
            if A[last] != A[i] or not same:
                same = A[last] == A[i]
                last += 1
```

```
            A[last] = A[i]
        i += 1

    return last + 1
```

reverse-bits.py

```
# DESC
# Follow up:
# Reverse bits of a given 32 bits unsigned integer.
# Example 1:
# Example 2:
# If this function is called many times, how would you optimize it?
# Note:

# NOTE
# In Java, the compiler represents the signed integers using 2's complement notati
# on. Therefore, in Example 2 above the input represents the signed integer -3 and
#  the output represents the signed integer -1073741825.
# Note that in some languages such as Java, there is no unsigned integer type. In
# this case, both input and output will be given as signed integer type and should
#  not affect your implementation, as the internal binary representation of the in
# teger is the same whether it is signed or unsigned.

# EXAMPLE
# Input: 11111111111111111111111111111101
# Output: 10111111111111111111111111111111
#
# Explanation: The input binary string 11111111111111111111111111111101 represent
# s the unsigned integer 4294967293, so return 3221225471 which its binary represe
# ntation is 10111111111111111111111111111111.
# Input: 00000010100101000001111010011100
# Output: 00111001011110000010100101000000
#
# Explanation: The input binary string 00000010100101000001111010011100 represent
# s the unsigned integer 43261596, so return 964176192 which its binary representa
# tion is 00111001011110000010100101000000.

# Time : O(logn) = O(32)
# Space: O(1)

class Solution(object):
    # @param n, an integer
    # @return an integer
    def reverseBits(self, n):
        result = 0
        for i in xrange(32):
            result <<= 1
            result |= n & 1
            n >>= 1
        return result

    def reverseBits2(self, n):
        string = bin(n)
        if '-' in string:
            string = string[:3] + string[3:].zfill(32)[::-1]
        else:
            string = string[:2] + string[2:].zfill(32)[::-1]
        return int(string, 2)
```

maximum-number-of-ones.py

```python
# maximum-number-of-ones is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def maximumNumberOfOnes(self, width, height, sideLength, maxOnes):
        """
        :type width: int
        :type height: int
        :type sideLength: int
        :type maxOnes: int
        :rtype: int
        """
        if width < height:
            width, height = height, width

        # 1. split matrix by SxS tiles
        # 2. split each SxS tile into four parts
        #    (r, c), (r, S-c), (S-r, c), (S-r, S-c)
        # 3. for each count of tile part in matrix is
        #    (R+1)*(C+1), (R+1)*C, R*(C+1), R*C (already in descending order)
        # 4. fill one into matrix by tile part of which count is in descending order
        #    until number of ones in a tile comes to maxOnes
        #
        # ps. area of a tile and its count in matrix are as follows:
        #
        #  |<---- c ---->|<-- S-c -->|
        #  ^             |           |
        #  |             |           |
        #  r (R+1)*(C+1) |  (R+1)*C  |
        #  |             |           |
        #  v             |           |
        #  ----------------------------
        #  ^             |           |
        #  |             |           |
        #  S-r  R*(C+1)  |   R*C     |
        #  |             |           |
        #  v             |           |
        #  ----------------------------
        #

        R, r = divmod(height, sideLength)
        C, c = divmod(width, sideLength)
        assert(R <= C)
        area_counts = [(r*c, (R+1)*(C+1)), \
                       (r*(sideLength-c), (R+1)*C), \
                       ((sideLength-r)*c, R*(C+1)), \
                       ((sideLength-r)*(sideLength-c), R*C)]
        result = 0
        for area, count in area_counts:
            area = min(maxOnes, area)
            result += count*area
            maxOnes -= area
            if not maxOnes:
                break
        return result
```

split-concatenated-strings.py

```python
# split-concatenated-strings is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def splitLoopedString(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        tmp = []
        for s in strs:
            tmp += max(s, s[::-1])
        s = "".join(tmp)

        result, st = "a", 0
        for i in xrange(len(strs)):
            body = "".join([s[st + len(strs[i]):], s[0:st]])
            for p in strs[i], strs[i][::-1]:
                for j in xrange(len(strs[i])):
                    if p[j] >= result[0]:
                        result = max(result, "".join([p[j:], body, p[:j]]))
            st += len(strs[i])
        return result
```

average-of-levels-in-binary-tree.py

```
# DESC
# Note:
# Example 1:

# NOTE
# The range of node's value is in the range of 32-bit signed integer.

# EXAMPLE
# Input:
#     3
#    / \
#   9  20
#     /  \
#    15   7
# Output: [3, 14.5, 11]
# Explanation
# :
# The average value of nodes on level 0 is 3,  on level 1 is 14.5, and on level
# 2 is 11. Hence return [3, 14.5, 11].

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


class Solution(object):
    def averageOfLevels(self, root):
        """
        :type root: TreeNode
        :rtype: List[float]
        """
        result = []
        q = [root]
        while q:
            total, count = 0, 0
            next_q = []
            for n in q:
                total += n.val
                count += 1
                if n.left:
                    next_q.append(n.left)
                if n.right:
                    next_q.append(n.right)
            q = next_q
            result.append(float(total) / count)
        return result
```

generalized-abbreviation.py

```
# generalized-abbreviation is not found.
# Time:  O(n * 2^n)
# Space: O(n)

class Solution(object):
    def generateAbbreviations(self, word):
        """
        :type word: str
        :rtype: List[str]
        """
        def generateAbbreviationsHelper(word, i, cur, res):
            if i == len(word):
                res.append("".join(cur))
                return
            cur.append(word[i])
            generateAbbreviationsHelper(word, i + 1, cur, res)
            cur.pop()
            if not cur or not cur[-1][-1].isdigit():
                for l in xrange(1, len(word) - i + 1):
                    cur.append(str(l))
                    generateAbbreviationsHelper(word, i + l, cur, res)
                    cur.pop()

        res, cur = [], []
        generateAbbreviationsHelper(word, 0, cur, res)
        return res
```

poor-pigs.py

```
# DESC
# General case:
# Note:
# If there are n buckets and a pig drinking poison will die within m minutes, how
# many pigs (x) you need to figure out the poisonous bucket within p minutes? Ther
# e is exactly one bucket with poison.
# There are 1000 buckets, one and only one of them is poisonous, while the rest ar
# e filled with water. They all look identical. If a pig drinks the poison it will
#  die within 15 minutes. What is the minimum amount of pigs you need to figure ou
# t which bucket is poisonous within one hour?
# Answer this question, and write an algorithm for the general case.

# NOTE
# A pig can be allowed to drink simultaneously on as many buckets as one would lik
# e, and the feeding takes no time.
# After a pig has instantly finished drinking buckets, there has to be a cool down
#  time of m minutes. During this time, only observation is allowed and no feeding
# s at all.
# Any given bucket can be sampled an infinite number of times (by an unlimited num
# ber of pigs).

# EXAMPLE
#

# Time:  O(1)
# Space: O(1)

import math


class Solution(object):
    def poorPigs(self, buckets, minutesToDie, minutesToTest):
        """
        :type buckets: int
        :type minutesToDie: int
        :type minutesToTest: int
        :rtype: int
        """
        return int(math.ceil(math.log(buckets) / math.log(minutesToTest / minutesToDie + 1)))
```

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

```
# DESC
# Example:
# An arrow can be shot up exactly vertically from different points along the x-axi
# s. A balloon with xstart and xend bursts by an arrow shot at x if xstart  x  x
# end. There is no limit to the number of arrows that can be shot. An arrow once s
# hot keeps travelling up infinitely. The problem is to find the minimum number of
#  arrows that must be shot to burst all balloons.
# There are a number of spherical balloons spread in two-dimensional space. For ea
# ch balloon, provided input is the start and end coordinates of the horizontal di
# ameter. Since it's horizontal, y-coordinates don't matter and hence the x-coordi
# nates of start and end of the diameter suffice. Start is always smaller than end
# . There will be at most 104 balloons.

# NOTE
#

# EXAMPLE
# Input:
# [[10,16], [2,8], [1,6], [7,12]]
#
# Output:
# 2
#
# Explanation:
# One way is to sh
# oot one arrow for example at x = 6 (bursting the balloons [2,8] and [1,6]) and a
# nother 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
```

palindrome-pairs.py

```
# DESC
# Example 2:
# Example 1:
# Given a list of unique words, find all pairs of distinct indices (i, j) in the g
# iven list, so that the concatenation of the two words, i.e. words[i] + words[j]
# is a palindrome.

# NOTE
#

# EXAMPLE
# Input: ["abcd","dcba","lls","s","sssll"]
# Output: [[0,1],[1,0],[3,2],[2,4]]
# Expl
# anation: The palindromes are ["dcbaabcd","abcddcba","slls","llssssll"]
# Input: ["bat","tab","cat"]
# Output: [[0,1],[1,0]]
# Explanation: The palindromes a
# re ["battab","tabbat"]

# Time:  O(n * k^2), n is the number of the words, k is the max length of the words.
# Space: O(n * k)

import collections


class Solution(object):
    def palindromePairs(self, words):
        """
        :type words: List[str]
        :rtype: List[List[int]]
        """
        res = []
        lookup = {}
        for i, word in enumerate(words):
            lookup[word] = i

        for i in xrange(len(words)):
            for j in xrange(len(words[i]) + 1):
                prefix = words[i][j:]
                suffix = words[i][:j]
                if prefix == prefix[::-1] and \
                   suffix[::-1] in lookup and lookup[suffix[::-1]] != i:
                    res.append([i, lookup[suffix[::-1]]])
                if j > 0 and suffix == suffix[::-1] and \
                   prefix[::-1] in lookup and lookup[prefix[::-1]] != i:
                    res.append([lookup[prefix[::-1]], i])
        return res

# Time:  O(n * k^2), n is the number of the words, k is the max length of the words.
# Space: O(n * k^2)
# Manacher solution.
class Solution_TLE(object):
    def palindromePairs(self, words):
        """
        :type words: List[str]
        :rtype: List[List[int]]
        """
```

```
        def manacher(s, P):
            def preProcess(s):
                if not s:
                    return ['^', '$']
                T = ['^']
                for c in s:
                    T +=  ["#", c]
                T += ['#', '$']
                return T

            T = preProcess(s)
            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]

        prefix, suffix = collections.defaultdict(list), collections.defaultdict(list)
        for i, word in enumerate(words):
            P = [0] * (2 * len(word) + 3)
            manacher(word, P)
            for j in xrange(len(P)):
                if j - P[j] == 1:
                    prefix[word[(j + P[j]) / 2:]].append(i)
                if j + P[j] == len(P) - 2:
                    suffix[word[:(j - P[j]) / 2]].append(i)
        res = []
        for i, word in enumerate(words):
            for j in prefix[word[::-1]]:
                if j != i:
                    res.append([i, j])
            for j in suffix[word[::-1]]:
                if len(word) != len(words[j]):
                    res.append([j, i])
        return res


# Time:  O(n * k^2), n is the number of the words, k is the max length of the words.
# Space: O(n * k)
# Trie solution.
class TrieNode(object):
    def __init__(self):
        self.word_idx = -1
        self.leaves = {}

    def insert(self, word, i):
        cur = self
        for c in word:
            if not c in cur.leaves:
                cur.leaves[c] = TrieNode()
            cur = cur.leaves[c]
        cur.word_idx = i

    def find(self, s, idx, res):
```

```
        cur = self
        for i in reversed(xrange(len(s))):
            if s[i] in cur.leaves:
                cur = cur.leaves[s[i]]
                if cur.word_idx not in (-1, idx) and \
                   self.is_palindrome(s, i - 1):
                    res.append([cur.word_idx, idx])
            else:
                break

    def is_palindrome(self, s, j):
        i = 0
        while i <= j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True

class Solution_MLE(object):
    def palindromePairs(self, words):
        """
        :type words: List[str]
        :rtype: List[List[int]]
        """
        res = []
        trie = TrieNode()
        for i in xrange(len(words)):
            trie.insert(words[i], i)

        for i in xrange(len(words)):
            trie.find(words[i], i, res)

        return res
```

minimum-cost-to-connect-sticks.py

```python
# minimum-cost-to-connect-sticks is not found.
# Time:  O(nlogn)
# Space: O(n)

import heapq


class Solution(object):
    def connectSticks(self, sticks):
        """
        :type sticks: List[int]
        :rtype: int
        """
        heapq.heapify(sticks)
        result = 0
        while len(sticks) > 1:
            x, y = heapq.heappop(sticks), heapq.heappop(sticks)
            result += x+y
            heapq.heappush(sticks, x+y)
        return result
```

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

```
# DESC
# Example:
# Given target = 20, return false.
# Given target = 5, return true.
# Write an efficient algorithm that searches for a value in an m x n matrix. This
# matrix has the following properties:
# Consider the following matrix:

# NOTE
# 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
# [
#   [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]
# ]

# 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
```

count-complete-tree-nodes.py

```
# DESC
# Definition of a complete binary tree from Wikipedia:
#
# In a complete binary tree
# every level, except possibly the last, is completely filled, and all nodes in th
# e last level are as far left as possible. It can have between 1 and 2h nodes inc
# lusive at the last level h.
# Note:
# Given a complete binary tree, count the number of nodes.
# Example:

# NOTE
#

# EXAMPLE
# Input:
#     1
#    / \
#   2   3
#  / \  /
# 4  5 6
#
# Output: 6

# Time:  O(h * logn) = O((logn)^2)
# Space: O(1)

class Solution(object):
    # @param {TreeNode} root
    # @return {integer}
    def countNodes(self, root):
        if root is None:
            return 0

        node, level = root, 0
        while node.left is not None:
            node = node.left
            level += 1

        # Binary search.
        left, right = 2 ** level, 2 ** (level + 1)
        while left < right:
            mid = left + (right - left) / 2
            if not self.exist(root, mid):
                right = mid
            else:
                left = mid + 1

        return left - 1

    # Check if the nth node exist.
    def exist(self, root, n):
        k = 1
        while k <= n:
            k <<= 1
        k >>= 2

        node = root
```

```
while k > 0:
    if (n & k) == 0:
        node = node.left
    else:
        node = node.right
    k >>= 1
return node is not None
```

mini-parser.py

```
# DESC
# Given a nested list of integers represented as a string, implement a parser to d
# eserialize it.
# Example 2:
# Example 1:
# Each element is either an integer, or a list -- whose elements may also be integ
# ers or other lists.
# Note: You may assume that the string is well-formed:

# NOTE
# String contains only digits 0-9, [, - ,, ].
# String does not contain white spaces.
# String is non-empty.

# EXAMPLE
# Given s = "324",
#
# You should return a NestedInteger object which contains a sing
# le integer 324.
# Given s = "[123,[456,[789]]]",
#
# Return a NestedInteger object containing a neste
# d list with 2 elements:
#
# 1. An integer containing value 123.
# 2. A nested list co
# ntaining 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):
        """
        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]
```

rearrange-string-k-distance-apart.py

```
# rearrange-string-k-distance-apart is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def rearrangeString(self, str, k):
        """
        :type str: str
        :type k: int
        :rtype: str
        """
        cnts = [0] * 26
        for c in str:
            cnts[ord(c) - ord('a')] += 1

        sorted_cnts = []
        for i in xrange(26):
            sorted_cnts.append((cnts[i], chr(i + ord('a'))))
        sorted_cnts.sort(reverse=True)

        max_cnt = sorted_cnts[0][0]
        blocks = [[] for _ in xrange(max_cnt)]
        i = 0
        for cnt in sorted_cnts:
            for _ in xrange(cnt[0]):
                blocks[i].append(cnt[1])
                i = (i + 1) % max(cnt[0], max_cnt - 1)

        for i in xrange(max_cnt-1):
            if len(blocks[i]) < k:
                return ""

        return "".join(map(lambda x : "".join(x), blocks))


# Time:  O(nlogc), c is the count of unique characters.
# Space: O(c)
from collections import Counter
from heapq import heappush, heappop
class Solution2(object):
    def rearrangeString(self, s, k):
        """
        :type str: str
        :type k: int
        :rtype: str
        """
        if k <= 1:
            return s

        cnts = Counter(s)
        heap = []
        for c, cnt in cnts.iteritems():
            heappush(heap, [-cnt, c])

        result = []
        while heap:
            used_cnt_chars = []
            for _ in xrange(min(k, len(s) - len(result))):
```

```
            if not heap:
                return ""
            cnt_char = heappop(heap)
            result.append(cnt_char[1])
            cnt_char[0] += 1
            if cnt_char[0] < 0:
                used_cnt_chars.append(cnt_char)
        for cnt_char in used_cnt_chars:
            heappush(heap, cnt_char)

    return "".join(result)
```

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

```
# DESC
# Example 4:
# Example 1:
# Example 3:
# Given two numbers, hour and minutes. Return the smaller angle (in degrees) forme
# d between the hour and the minute hand.
# Example 5:
# Constraints:
# Example 2:

# NOTE
# 1 <= hour <= 12
# Answers within 10^-5 of the actual value will be accepted as correct.
# 0 <= minutes <= 59

# EXAMPLE
# Input: hour = 12, minutes = 30
# Output: 165
# Input: hour = 3, minutes = 15
# Output: 7.5
# Input: hour = 4, minutes = 50
# Output: 155
# Input: hour = 12, minutes = 0
# Output: 0
# Input: hour = 3, minutes = 30
# Output: 75

# 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
```

shortest-word-distance.py

```python
# shortest-word-distance is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {string[]} words
    # @param {string} word1
    # @param {string} word2
    # @return {integer}
    def shortestDistance(self, words, word1, word2):
        dist = float("inf")
        i, index1, index2 = 0, None, None
        while i < len(words):
            if words[i] == word1:
                index1 = i
            elif words[i] == word2:
                index2 = i

            if index1 is not None and index2 is not None:
                dist = min(dist, abs(index1 - index2))
            i += 1

        return dist
```

sum-of-distances-in-tree.py

```
# DESC
# The ith edge connects nodes edges[i][0] and edges[i][1] together.
# Return a list ans, where ans[i] is the sum of the distances between node i and a
# ll other nodes.
# Example 1:
# An undirected, connected tree with N nodes labelled 0...N-1 and N-1 edges are given.
# Note: 1 <= N <= 10000

# NOTE
#

# EXAMPLE
# Input: N = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]
# Output: [8,12,6,10,10,10]
#
# Explanation:
# Here is a diagram of the given tree:
#   0
#  / \
# 1   2
#    /|\
#   3 4 5
#
# We can see that dist(0,1) + dist(0,2) + dist(0,3) + dist(0,4) + dist(0,5)
# equal
# s 1 + 1 + 2 + 2 + 2 = 8.  Hence, answer[0] = 8, and so on.

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

import collections


class Solution(object):
    def sumOfDistancesInTree(self, N, edges):
        """
        :type N: int
        :type edges: List[List[int]]
        :rtype: List[int]
        """
        def dfs(graph, node, parent, count, result):
            for nei in graph[node]:
                if nei != parent:
                    dfs(graph, nei, node, count, result)
                    count[node] += count[nei]
                    result[node] += result[nei]+count[nei]

        def dfs2(graph, node, parent, count, result):
            for nei in graph[node]:
                if nei != parent:
                    result[nei] = result[node]-count[nei] + \
                                  len(count)-count[nei]
                    dfs2(graph, nei, node, count, result)

        graph = collections.defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)
```

```
count = [1] * N
result = [0] * N

dfs(graph, 0, None, count, result)
dfs2(graph, 0, None, count, result)
return result
```

number-of-operations-to-make-network-connected.py

```python
# number-of-operations-to-make-network-connected is not found.
# Time:  O(|E| + |V|)
# Space: O(|V|)

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:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        self.count -= 1
        return True


class Solution(object):
    def makeConnected(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: int
        """
        if len(connections) < n-1:
            return -1
        union_find = UnionFind(n)
        for i, j in connections:
            union_find.union_set(i, j)
        return union_find.count - 1


# Time:  O(|E| + |V|)
# Space: O(|V|)
import collections


class Solution2(object):
    def makeConnected(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: int
        """
        def dfs(i, lookup):
            if i in lookup:
                return 0
            lookup.add(i)
            if i in G:
                for j in G[i]:
                    dfs(j, lookup)
            return 1
```

```
if len(connections) < n-1:
    return -1
G = collections.defaultdict(list)
for i, j in connections:
    G[i].append(j)
    G[j].append(i)
lookup = set()
return sum(dfs(i, lookup) for i in xrange(n)) - 1
```

out-of-boundary-paths.py

```
# DESC
# Example 2:
# Example 1:
# Note:
# There is an m by n grid with a ball. Given the start coordinate (i,j) of the bal
# l, you can move the ball to adjacent cell or cross the grid boundary in four dir
# ections (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 ve
# ry large, return it after mod 109 + 7.

# 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].

# EXAMPLE
# Input: m = 1, n = 3, N = 3, i = 0, j = 1
# Output: 12
# Explanation:
# Input: m = 2, n = 2, N = 2, i = 0, j = 0
# Output: 6
# Explanation:

# 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]
```

maximum-size-subarray-sum-equals-k.py

```python
# maximum-size-subarray-sum-equals-k is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def maxSubArrayLen(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        sums = {}
        cur_sum, max_len = 0, 0
        for i in xrange(len(nums)):
            cur_sum += nums[i]
            if cur_sum == k:
                max_len = i + 1
            elif cur_sum - k in sums:
                max_len = max(max_len, i - sums[cur_sum - k])
            if cur_sum not in sums:
                sums[cur_sum] = i  # Only keep the smallest index.
        return max_len
```

minimum-cost-for-tickets.py

```
# DESC
# Example 1:
# In a country popular for train travel, you have planned some train travelling on
# e year in advance.  The days of the year that you will travel is given as an arr
# ay days.  Each day is an integer from 1 to 365.
# The passes allow that many days of consecutive travel.  For example, if we get a
#  7-day pass on day 2, then we can travel for 7 days: day 2, 3, 4, 5, 6, 7, and 8
# .
# Note:
# Return the minimum number of dollars you need to travel every day in the given l
# ist of days.
# Example 2:
# Train tickets are sold in 3 different ways:

# NOTE
# 1 <= costs[i] <= 1000
# days is in strictly increasing order.
# a 1-day pass is sold for costs[0] dollars;
# a 7-day pass is sold for costs[1] dollars;
# a 30-day pass is sold for costs[2] dollars.
# 1 <= days.length <= 365
# 1 <= days[i] <= 365
# costs.length == 3

# EXAMPLE
# Input: days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
# Output: 17
# Explanat
# ion:
# For example, here is one way to buy passes that lets you travel your trave
# l plan:
# On day 1, you bought a 30-day pass for costs[2] = $15 which covered days
#  1, 2, ..., 30.
# On day 31, you bought a 1-day pass for costs[0] = $2 which cover
# ed day 31.
# In total you spent $17 and covered all the days of your travel.
# Input: days = [1,4,6,7,8,20], costs = [2,7,15]
# Output: 11
# Explanation:
# For exam
# ple, here is one way to buy passes that lets you travel your travel plan:
# On day
#  1, you bought a 1-day pass for costs[0] = $2, which covered day 1.
# On day 3, yo
# u bought a 7-day pass for costs[1] = $7, which covered days 3, 4, ..., 9.
# On day
#  20, you bought a 1-day pass for costs[0] = $2, which covered day 20.
# In total y
# ou spent $11 and covered all the days of your travel.

# Time:  O(n)
# space: O(1)

class Solution(object):
    def mincostTickets(self, days, costs):
        """
        :type days: List[int]
        :type costs: List[int]
```

```
    :rtype: int
    """
    durations = [1, 7, 30]
    W = durations[-1]
    dp = [float("inf") for i in xrange(W)]
    dp[0] = 0
    last_buy_days = [0, 0, 0]
    for i in xrange(1,len(days)+1):
        dp[i%W] = float("inf")
        for j in xrange(len(durations)):
            while i-1 < len(days) and \
                  days[i-1] > days[last_buy_days[j]]+durations[j]-1:
                last_buy_days[j] += 1  # Time: O(n)
            dp[i%W] = min(dp[i%W], dp[last_buy_days[j]%W]+costs[j])
    return dp[len(days)%W]
```

battleships-in-a-board.py

```
# DESC
# Follow up:
# Could you do it in one-pass, using only O(1) extra memory and without
#  modifying the value of the board?
# Invalid Example:
# Example:

# NOTE
# 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 - the
# re are no adjacent battleships.
# You receive a valid board, made of only battleships or empty slots.

# EXAMPLE
# X..X
# ...X
# ...X
# ...X
# XXXX
# ...X

# 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
```

merge-sorted-array.py

```
# merge-sorted-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A  a list of integers
    # @param m  an integer, length of A
    # @param B  a list of integers
    # @param n  an integer, length of B
    # @return nothing
    def merge(self, A, m, B, n):
        last, i, j = m + n - 1, m - 1, n - 1

        while i >= 0 and j >= 0:
            if A[i] > B[j]:
                A[last] = A[i]
                last, i = last - 1, i - 1
            else:
                A[last] = B[j]
                last, j = last - 1, j - 1

        while j >= 0:
                A[last] = B[j]
                last, j = last - 1, j - 1
```

subarray-sum-equals-k.py

```python
# DESC
# Given an array of integers and an integer k, you need to find the total number o
# f continuous subarrays whose sum equals to k.
# Constraints:
# Example 1:

# NOTE
# 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].

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

# 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
```

design-browser-history.py

```python
# design-browser-histor is not found.
# Time:  ctor  : O(1)
#        visit : O(n)
#        back  : O(1)
#        foward: O(1)
# Space: O(n)

class BrowserHistory(object):

    def __init__(self, homepage):
        """
        :type homepage: str
        """
        self.__history = [homepage]
        self.__curr = 0

    def visit(self, url):
        """
        :type url: str
        :rtype: None
        """
        while len(self.__history) > self.__curr+1:
            self.__history.pop()
        self.__history.append(url)
        self.__curr += 1

    def back(self, steps):
        """
        :type steps: int
        :rtype: str
        """
        self.__curr = max(self.__curr-steps, 0)
        return self.__history[self.__curr]

    def forward(self, steps):
        """
        :type steps: int
        :rtype: str
        """
        self.__curr = min(self.__curr+steps, len(self.__history)-1)
        return self.__history[self.__curr]
```

group-shifted-strings.py

```
# group-shifted-strings is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections


class Solution(object):
    # @param {string[]} strings
    # @return {string[][]}
    def groupStrings(self, strings):
        groups = collections.defaultdict(list)
        for s in strings:  # Grouping.
            groups[self.hashStr(s)].append(s)

        result = []
        for key, val in groups.iteritems():
            result.append(sorted(val))

        return result

    def hashStr(self, s):
        base = ord(s[0])
        hashcode = ""
        for i in xrange(len(s)):
            if ord(s[i]) - base >= 0:
                hashcode += unichr(ord('a') + ord(s[i]) - base)
            else:
                hashcode += unichr(ord('a') + ord(s[i]) - base + 26)
        return hashcode
```

frog-jump.py

```
# frog-jum is not found.
# Time:  O(n^2)
# Space: O(n^2)

class Solution(object):
    def canCross(self, stones):
        """
        :type stones: List[int]
        :rtype: bool
        """
        if stones[1] != 1:
            return False

        last_jump_units = {s: set() for s in stones}
        last_jump_units[1].add(1)
        for s in stones[:-1]:
            for j in last_jump_units[s]:
                for k in (j-1, j, j+1):
                    if k > 0 and s+k in last_jump_units:
                        last_jump_units[s+k].add(k)
        return bool(last_jump_units[stones[-1]])
```

kth-smallest-element-in-a-bst.py

```
# DESC
# Example 1:
# Follow up:
#
# What if the BST is modified (insert/delete operations) often and you
#  need to find the kth smallest frequently? How would you optimize the kthSmalles
# t routine?
# Constraints:
# Given a binary search tree, write a function kthSmallest to find the kth smalles
# t element in it.
# Example 2:

# NOTE
# The number of elements of the BST is between 1 to 10^4.
# You may assume k is always valid, 1  k  BST's total elements.

# EXAMPLE
# Input: root = [5,3,6,2,4,null,null,1], k = 3
#        5
#       / \
#      3   6
#     /
#  \
#    2   4
#   /
#  1
# Output: 3
# Input: root = [3,1,4,null,2], k = 1
#    3
#   / \
#  1   4
#   \
#    2
# Output: 1

# Time:  O(max(h, k))
# Space: O(h)

class Solution(object):
    # @param {TreeNode} root
    # @param {integer} k
    # @return {integer}
    def kthSmallest(self, root, k):
        s, cur, rank = [], root, 0

        while s or cur:
            if cur:
                s.append(cur)
                cur = cur.left
            else:
                cur = s.pop()
                rank += 1
                if rank == k:
                    return cur.val
                cur = cur.right

        return float("-inf")
```

```
# time: O(max(h, k))
# space: O(h)

from itertools import islice


class Solution2(object):
    def kthSmallest(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: int
        """
        def gen_inorder(root):
            if root:
                for n in gen_inorder(root.left):
                    yield n

                yield root.val

                for n in gen_inorder(root.right):
                    yield n

        return next(islice(gen_inorder(root), k-1, k))
```

best-time-to-buy-and-sell-stock-ii.py

```
# DESC
# Constraints:
# Example 2:
# Example 3:
# Say you have an array prices for which the ith element is the price of a given s
# tock on day i.
# Note: You may not engage in multiple transactions at the same time (i.e., you mu
# st sell the stock before you buy again).
# Design an algorithm to find the maximum profit. You may complete as many transac
# tions as you like (i.e., buy one and sell one share of the stock multiple times)
# .
# Example 1:

# NOTE
# 0 <= prices[i] <= 10 ^ 4
# 1 <= prices.length <= 3 * 10 ^ 4

# EXAMPLE
# Input: [7,1,5,3,6,4]
# Output: 7
# Explanation: Buy on day 2 (price = 1) and sell on
#  day 3 (price = 5), profit = 5-1 = 4.
#              Then buy on day 4 (price = 3)
#  and sell on day 5 (price = 6), profit = 6-3 = 3.
# Input: [1,2,3,4,5]
# Output: 4
# Explanation: Buy on day 1 (price = 1) and sell on d
# ay 5 (price = 5), profit = 5-1 = 4.
#              Note that you cannot buy on day
#  1, buy on day 2 and sell them later, as you are
#              engaging multiple
# transactions at the same time. You must sell before buying again.
# Input: [7,6,4,3,1]
# Output: 0
# Explanation: In this case, no transaction is done,
# i.e. max profit = 0.

# Time:  O(n)
# Space: O(1)


class Solution(object):
    # @param prices, a list of integer
    # @return an integer
    def maxProfit(self, prices):
        profit = 0
        for i in xrange(len(prices) - 1):
            profit += max(0, prices[i + 1] - prices[i])
        return profit

    def maxProfit2(self, prices):
        return sum(map(lambda x: max(prices[x + 1] - prices[x], 0),
                       xrange(len(prices[:-1]))))
```

airplane-seat-assignment-probability.py

```
# airplane-seat-assignment-probabilit is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def nthPersonGetsNthSeat(self, n):
        """
        :type n: int
        :rtype: float
        """
        # p(k) = 1 * (prob that 1th passenger takes his own seat) +
        #        0 * (prob that 1th passenger takes kth one's seat) +
        #        1 * (prob that 1th passenger takes the others' seat) *
        #            (prob that the first k-1 passengers get a seat
        #             which is not kth one's seat)
        #      = 1/k + p(k-1)*(k-2)/k
        #
        # p(1) = 1
        # p(2) = 1/2 + p(1) * (2-2)/2 = 1/2
        # p(3) = 1/3 + p(2) * (3-2)/3 = 1/3 + 1/2 * (3-2)/3 = 1/2
        # ...
        # p(n) = 1/n + 1/2 * (n-2)/n = (2+n-2)/(2n) = 1/2
        return 0.5 if n != 1 else 1.0

# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def nthPersonGetsNthSeat(self, n):
        """
        :type n: int
        :rtype: float
        """
        dp = [0.0]*2
        dp[0] = 1.0  # zero-indexed
        for i in xrange(2, n+1):
            dp[(i-1)%2] = 1.0/i+dp[(i-2)%2]*(i-2)/i
        return dp[(n-1)%2]
```

squirrel-simulation.py

```
# squirrel-simulation is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minDistance(self, height, width, tree, squirrel, nuts):
        """
        :type height: int
        :type width: int
        :type tree: List[int]
        :type squirrel: List[int]
        :type nuts: List[List[int]]
        :rtype: int
        """
        def distance(a, b):
            return abs(a[0] - b[0]) + abs(a[1] - b[1])

        result = 0
        d = float("inf")
        for nut in nuts:
            result += (distance(nut, tree) * 2)
            d = min(d, distance(nut, squirrel) - distance(nut, tree))
        return result + d
```

sequence-reconstruction.py

```
# sequence-reconstruction is not found.
# Time:  O(n * s), n is the size of org, s is the size of seqs
# Space: O(n)

import collections


class Solution(object):
    def sequenceReconstruction(self, org, seqs):
        """
        :type org: List[int]
        :type seqs: List[List[int]]
        :rtype: bool
        """
        if not seqs:
            return False
        pos = [0] * (len(org) + 1)
        for i in xrange(len(org)):
            pos[org[i]] = i

        is_matched = [False] * (len(org) + 1)
        cnt_to_match = len(org) - 1
        for seq in seqs:
            for i in xrange(len(seq)):
                if not 0 < seq[i] <= len(org):
                    return False
                if i == 0:
                    continue
                if pos[seq[i-1]] >= pos[seq[i]]:
                    return False
                if is_matched[seq[i-1]] == False and pos[seq[i-1]] + 1 == pos[seq[i]]:
                    is_matched[seq[i-1]] = True
                    cnt_to_match -= 1

        return cnt_to_match == 0


# Time:  O(|V| + |E|)
# Space: O(|E|)
class Solution2(object):
    def sequenceReconstruction(self, org, seqs):
        """
        :type org: List[int]
        :type seqs: List[List[int]]
        :rtype: bool
        """
        graph = collections.defaultdict(set)
        indegree = collections.defaultdict(int)
        integer_set = set()
        for seq in seqs:
            for i in seq:
                integer_set.add(i)
            if len(seq) == 1:
                if seq[0] not in indegree:
                    indegree[seq[0]] = 0
                continue
            for i in xrange(len(seq)-1):
                if seq[i] not in indegree:
```

```
            indegree[seq[i]] = 0
        if seq[i+1] not in graph[seq[i]]:
            graph[seq[i]].add(seq[i+1])
            indegree[seq[i+1]] += 1

cnt_of_zero_indegree = 0
res = []
q = []
for i in indegree:
    if indegree[i] == 0:
        cnt_of_zero_indegree += 1
        if cnt_of_zero_indegree > 1:
            return False
        q.append(i)

while q:
    i = q.pop()
    res.append(i)
    cnt_of_zero_indegree = 0
    for j in graph[i]:
        indegree[j] -= 1
        if indegree[j] == 0:
            cnt_of_zero_indegree += 1
            if cnt_of_zero_indegree > 1:
                return False
            q.append(j)
return res == org and len(org) == len(integer_set)
```

design-hit-counter.py

```
# design-hit-counter is not found.
# Time:  O(1), amortized
# Space: O(k), k is the count of seconds.

from collections import deque

class HitCounter(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__k = 300
        self.__dq = deque()
        self.__count = 0

    def hit(self, timestamp):
        """
        Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
        :type timestamp: int
        :rtype: void
        """
        self.getHits(timestamp)
        if self.__dq and self.__dq[-1][0] == timestamp:
            self.__dq[-1][1] += 1
        else:
            self.__dq.append([timestamp, 1])
        self.__count += 1

    def getHits(self, timestamp):
        """
        Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity).
        :type timestamp: int
        :rtype: int
        """
        while self.__dq and self.__dq[0][0] <= timestamp - self.__k:
            self.__count -= self.__dq.popleft()[1]
        return self.__count
```

set-matrix-zeroes.py

```
# DESC
# Example 1:
# Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do
#  it in-place.
# Example 2:
# Follow up:

# NOTE
# A simple improvement uses O(m + n) space, but still not the best solution.
# Could you devise a constant space solution?
# A straight forward solution using O(mn) space is probably a bad idea.

# EXAMPLE
# Input:
# [
#   [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]
# ]
# Input:
# [
#   [1,1,1],
#   [1,0,1],
#   [1,1,1]
# ]
# Output:
# [
#   [1,0,1],
#   [0,0,0],
#   [
# 1,0,1]
# ]

from 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

```
# DESC
# Return the maximum score of a pair of sightseeing spots.
# Given an array A of positive integers, A[i] represents the value of the i-th sig
# htseeing spot, and two sightseeing spots i and j have distance j - i between the
# m.
# Example 1:
# 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.
# i < j
# Note:

# NOTE
# 1 <= A[i] <= 1000
# 2 <= A.length <= 50000

# EXAMPLE
# Input: [8,1,5,2,6]
# Output: 11
# Explanation: i = 0, j = 2, A[i] + A[j] + i - j = 8
#  + 5 + 0 - 2 = 11

# 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
```

largest-unique-number.py

```python
# largest-unique-number is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def largestUniqueNumber(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        A.append(-1)
        return max(k for k,v in collections.Counter(A).items() if v == 1)
```

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

```
# DESC
# Example 2:
# Example 3:
# Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.
# Example 1:
# You are given two integer arrays nums1 and nums2 sorted in ascending order and a
# n integer k.
# Define a pair (u,v) which consists of one element from the first array and one e
# lement from the second array.

# NOTE
#

# EXAMPLE
# Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
# Output: [[1,2],[1,4],[1,6]]
# Exp
# lanation: 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]
# Input: nums1 = [1,2], nums2 = [3], k = 3
# Output: [1,3],[2,3]
# Explanation: All po
# ssible pairs are returned from the sequence: [1,3],[2,3]
# 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]

# Time:  O(k * log(min(n, m, k))), where n is the size of num1, and m is the size of num2.
# 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
        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)
```

escape-a-large-maze.py

```
# DESC
# Note:
# Return true if and only if it is possible to reach the target square through a s
# equence of moves.
# In a 1 million by 1 million grid, the coordinates of each grid square are (x, y)
#  with 0 <= x, y < 10^6.
# source
# We start at the source square and want to reach the target square.  Each move, w
# e can walk to a 4-directionally adjacent square in the grid that isn't in the gi
# ven list of blocked squares.
# Example 2:
# Example 1:

# NOTE
# source != target
# source.length == target.length == 2
# 0 <= blocked.length <= 200
# 0 <= source[i][j], target[i][j] < 10^6
# blocked[i].length == 2
# 0 <= blocked[i][j] < 10^6

# EXAMPLE
# Input: blocked = [[0,1],[1,0]], source = [0,0], target = [0,2]
# Output: false
# Exp
# lanation:
# The target square is inaccessible starting from the source square, be
# cause we can't walk outside the grid.
# Input: blocked = [], source = [0,0], target = [999999,999999]
# Output: true
# Expla
# nation:
# Because there are no blocked cells, it's possible to reach the target s
# quare.

# Time:  O(n^2), n is the number of blocked
# Space: O(n)

import collections


class Solution(object):
    def isEscapePossible(self, blocked, source, target):
        """
        :type blocked: List[List[int]]
        :type source: List[int]
        :type target: List[int]
        :rtype: bool
        """
        R, C = 10**6, 10**6
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        def bfs(blocks, source, target):
            max_area_surrounded_by_blocks = len(blocks)*(len(blocks)-1)//2
            lookup = set([source])
            if len(lookup) > max_area_surrounded_by_blocks:
                return True
            q = collections.deque([source])
```

```
        while q:
            source = q.popleft()
            if source == target:
                return True
            for direction in directions:
                nr, nc = source[0]+direction[0], source[1]+direction[1]
                if not ((0 <= nr < R) and
                        (0 <= nc < C) and
                        (nr, nc) not in lookup and
                        (nr, nc) not in blocks):
                    continue
                lookup.add((nr, nc))
                if len(lookup) > max_area_surrounded_by_blocks:
                    return True
                q.append((nr, nc))
        return False

    return bfs(set(map(tuple, blocked)), tuple(source), tuple(target)) and \
           bfs(set(map(tuple, blocked)), tuple(target), tuple(source))
```

distribute-coins-in-binary-tree.py

```
# DESC
# 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.
# Example 1:
# 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.)
# Example 3:
# Return the number of moves required to make every node have exactly one coin.
# Example 4:
# Note:
# Example 2:

# NOTE
# 1<= N <= 100
# 0 <= node.val <= N

# EXAMPLE
# Input: [3,0,0]
# Output: 2
# Explanation: From the root of the tree, we move one coi
# n to its left child, and one coin to its right child.
# Input: [1,0,0,null,3]
# Output: 4
# Input: [1,0,2]
# Output: 2
# Input: [0,3,0]
# Output: 3
# Explanation: From the left child of the root, we move t
# wo coins to the root [taking two moves].  Then, we move one coin from the root o
# f the tree to the right child.

# 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 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

```
# sort-characters-by-frequenc is not found.
# 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

```
# DESC
# Example 2:
# If there isn't any rectangle, return 0.
# 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.
# Note:
# Example 1:

# NOTE
# 0 <= points[i][1] <= 40000
# 1 <= points.length <= 500
# All points are distinct.
# 0 <= points[i][0] <= 40000

# EXAMPLE
# Input: [[1,1],[1,3],[3,1],[3,3],[2,2]]
# Output: 4
# Input: [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]
# Output: 2

# 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])):
                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)
```

```python
# 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

```
# DESC
# [low, high]
# Example 2:
# Example 1:
# Constraints:
# Return a sorted list of all the integers in the range [low, high] inclusive that
#  have sequential digits.
# An integer has sequential digits if and only if each digit in the number is one
# more than the previous digit.

# NOTE
# 10 <= low <= high <= 10^9

# EXAMPLE
# Input: low = 100, high = 300
# Output: [123,234]
# Input: low = 1000, high = 13000
# Output: [1234,2345,3456,4567,5678,6789,12345]

# 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
```

html-entity-parser.py

```
# html-entity-parser is not found.
# Time:  O(n + m + z) = O(m), n is the total size of patterns
#                           , m is the total size of query string
#                           , z is the number of all matched strings
#                           , O(n) = O(1), O(z) = O(m) in this problem
# Space: O(t) = O(1), t is the total size of ac automata trie
#                   , O(t) = O(1) in this problem

import collections


class AhoNode(object):
    def __init__(self):
        self.children = collections.defaultdict(AhoNode)
        self.indices = []
        self.suffix = None
        self.output = None


class AhoTrie(object):

    def step(self, letter):
        while self.__node and letter not in self.__node.children:
            self.__node = self.__node.suffix
        self.__node = self.__node.children[letter] if self.__node else self.__root
        return self.__get_ac_node_outputs(self.__node)

    def __init__(self, patterns):
        self.__root = self.__create_ac_trie(patterns)
        self.__node = self.__create_ac_suffix_and_output_links(self.__root)

    def __create_ac_trie(self, patterns):  # Time:  O(n), Space: O(t)
        root = AhoNode()
        for i, pattern in enumerate(patterns):
            node = root
            for c in pattern:
                node = node.children[c]
            node.indices.append(i)
        return root

    def __create_ac_suffix_and_output_links(self, root):  # Time:  O(n), Space: O(t)
        queue = collections.deque()
        for node in root.children.itervalues():
            queue.append(node)
            node.suffix = root

        while queue:
            node = queue.popleft()
            for c, child in node.children.iteritems():
                queue.append(child)
                suffix = node.suffix
                while suffix and c not in suffix.children:
                    suffix = suffix.suffix
                child.suffix = suffix.children[c] if suffix else root
                child.output = child.suffix if child.suffix.indices else child.suffix.output

        return root
```

```
    def __get_ac_node_outputs(self, node):  # Time:  O(z)
        result = []
        for i in node.indices:
            result.append(i)
        output = node.output
        while output:
            for i in output.indices:
                result.append(i)
            output = output.output
        return result


class Solution(object):
    def entityParser(self, text):
        """
        :type text: str
        :rtype: str
        """
        patterns = ["&quot;", "&apos;", "&amp;", "&gt;", "&lt;", "&frasl;"]
        chars = ["\"", "'", "&", ">", "<", "/"]
        trie = AhoTrie(patterns)
        positions = []
        for i in xrange(len(text)):
            for j in trie.step(text[i]):
                positions.append([i-len(patterns[j])+1, j])
        result = []
        i, j = 0, 0
        while i != len(text):
            if j == len(positions) or i != positions[j][0]:
                result.append(text[i])
                i += 1
            else:
                result.append(chars[positions[j][1]])
                i += len(patterns[positions[j][1]])
                j += 1
        return "".join(result)


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def entityParser(self, text):
        """
        :type text: str
        :rtype: str
        """
        patterns = ["&quot;", "&apos;", "&amp;", "&gt;", "&lt;", "&frasl;"]
        chars = ["\"", "'", "&", ">", "<", "/"]
        result = []
        i, j = 0, 0
        while i != len(text):
            if text[i] != '&':
                result.append(text[i])
                i += 1
            else:
                for j, pattern in enumerate(patterns):
                    if pattern == text[i:i+len(pattern)]:
                        result.append(chars[j])
                        i += len(pattern)
                        break
```

```
        else:
            result.append(text[i])
            i += 1
    return "".join(result)
```

perfect-squares.py

```
# DESC
# Example 1:
# Example 2:
# Given a positive integer n, find the least number of perfect square numbers (for
#  example, 1, 4, 9, 16, ...) which sum to n.

# NOTE
#

# EXAMPLE
# Input: n = 13
# Output: 2
# Explanation: 13 = 4 + 9.
# Input: n = 12
# Output: 3
# Explanation: 12 = 4 + 4 + 4.

# 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]
```

construct-binary-tree-from-string.py

```python
# construct-binary-tree-from-string is not found.
# 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 str2tree(self, s):
        """
        :type s: str
        :rtype: TreeNode
        """
        def str2treeHelper(s, i):
            start = i
            if s[i] == '-': i += 1
            while i < len(s) and s[i].isdigit(): i += 1
            node = TreeNode(int(s[start:i]))
            if i < len(s) and s[i] == '(':
                i += 1
                node.left, i = str2treeHelper(s, i)
                i += 1
            if i < len(s) and s[i] == '(':
                i += 1
                node.right, i = str2treeHelper(s, i)
                i += 1
            return node, i

        return str2treeHelper(s, 0)[0] if s else None
```

rectangle-area-ii.py

```
# DESC
# We are given a list of (axis-aligned) rectangles.  Each rectangle[i] = [x1, y1,
# x2, y2] , where (x1, y1) are the coordinates of the bottom-left corner, and (x2,
#  y2) are the coordinates of the top-right corner of the ith rectangle.
# Example 2:
# Find the total area covered by all rectangles in the plane.  Since the answer ma
# y be too large, return it modulo 10^9 + 7.
# rectangles
# Example 1:
# Note:

# NOTE
# The total area covered by all rectangles will never exceed 2^63 - 1 and thus wil
# l fit in a 64-bit signed integer.
# 1 <= rectangles.length <= 200
# rectanges[i].length = 4
# 0 <= rectangles[i][j] <= 10^9

# EXAMPLE
# Input: [[0,0,1000000000,1000000000]]
# Output: 49
# Explanation: The answer is 10^18
#  modulo (10^9 + 7), which is (10^9)^2 = (-7)^2 = 49.
# Input: [[0,0,2,2],[1,0,2,3],[1,0,3,1]]
# Output: 6
# Explanation: As illustrated in
# the picture.

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

class SegmentTreeNode(object):
    def __init__(self, start, end):
        self.start, self.end = start, end
        self.total = self.count = 0
        self._left = self._right = None

    def mid(self):
        return (self.start+self.end) // 2

    def left(self):
        self._left = self._left or SegmentTreeNode(self.start, self.mid())
        return self._left

    def right(self):
        self._right = self._right or SegmentTreeNode(self.mid(), self.end)
        return self._right

    def update(self, X, i, j, val):
        if i >= j:
            return 0
        if self.start == i and self.end == j:
            self.count += val
        else:
            self.left().update(X, i, min(self.mid(), j), val)
            self.right().update(X, max(self.mid(), i), j, val)
        if self.count > 0:
            self.total = X[self.end]-X[self.start]
```

```
        else:
            self.total = self.left().total + self.right().total
        return self.total


class Solution(object):
    def rectangleArea(self, rectangles):
        """
        :type rectangles: List[List[int]]
        :rtype: int
        """
        OPEN, CLOSE = 1, -1
        events = []
        X = set()
        for x1, y1, x2, y2 in rectangles:
            events.append((y1, OPEN, x1, x2))
            events.append((y2, CLOSE, x1, x2))
            X.add(x1)
            X.add(x2)
        events.sort()
        X = sorted(X)
        Xi = {x: i for i, x in enumerate(X)}

        st = SegmentTreeNode(0, len(X)-1)
        result = 0
        cur_x_sum = 0
        cur_y = events[0][0]
        for y, typ, x1, x2 in events:
            result += cur_x_sum * (y-cur_y)
            cur_x_sum = st.update(X, Xi[x1], Xi[x2], typ)
            cur_y = y
        return result % (10**9+7)
```

reverse-linked-list.py

```
# DESC
# Example:
# Reverse a singly linked list.
# Follow up:
# A linked list can be reversed either iteratively or recursively. Could you imple
# ment both?

# NOTE
#

# EXAMPLE
# Input: 1->2->3->4->5->NULL
# Output: 5->4->3->2->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))

# Iterative solution.
class Solution(object):
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        dummy = ListNode(float("-inf"))
        while head:
            dummy.next, head.next, head = head, dummy.next, head.next
        return dummy.next

# Time:  O(n)
# Space: O(n)
# Recursive solution.
class Solution2(object):
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        [begin, end] = self.reverseListRecu(head)
        return begin

    def reverseListRecu(self, head):
        if not head:
            return [None, None]

        [begin, end] = self.reverseListRecu(head.next)

        if end:
            end.next = head
            head.next = None
            return [begin, head]
        else:
            return [head, head]
```

all-nodes-distance-k-in-binary-tree.py

```
# DESC
# Return a list of the values of all nodes that have a distance K from the target
# node.  The answer can be returned in any order.
# Note:
# We are given a binary tree (with root node root), a target node, and an integer
# value K.
# Example 1:

# NOTE
# The target node is a node in the tree.
# Each node in the tree has unique values 0 <= node.val <= 500.
# The given tree is non-empty.
# 0 <= K <= 1000.

# EXAMPLE
# Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2
#
# Output: [7,4,1]
#
#
# Explanation:
# The nodes that are a distance 2 from the target node (with value
# 5)
# have values 7, 4, and 1.
#
#
#
# Note that the inputs "root" and "target" are actu
# ally TreeNodes.
# The descriptions of the inputs above are just serializations of
# these objects.

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

import collections


class Solution(object):
    def distanceK(self, root, target, K):
        """
        :type root: TreeNode
        :type target: TreeNode
        :type K: int
        :rtype: List[int]
        """
        def dfs(parent, child, neighbors):
            if not child:
                return
            if parent:
                neighbors[parent.val].append(child.val)
                neighbors[child.val].append(parent.val)
            dfs(child, child.left, neighbors)
            dfs(child, child.right, neighbors)

        neighbors = collections.defaultdict(list)
        dfs(None, root, neighbors)
        bfs = [target.val]
```

```
    lookup = set(bfs)
    for _ in xrange(K):
        bfs = [nei for node in bfs
               for nei in neighbors[node]
               if nei not in lookup]
        lookup |= set(bfs)
    return bfs
```

destination-city.py

```python
# destination-cit is not found.
# Time:  O(n)
# Space: O(n)

import itertools


class Solution(object):
    def destCity(self, paths):
        """
        :type paths: List[List[str]]
        :rtype: str
        """
        A, B = map(set, itertools.izip(*paths))
        return (B-A).pop()
```

longest-turbulent-subarray.py

```
# longest-turbulent-subarra is not found.
# 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
```

add-and-search-word-data-structure-design.py

```python
# add-and-search-word-data-structure-design is not found.
# Time:  O(min(n, h)), per operation
# Space: O(min(n, h))

class TrieNode(object):
    # Initialize your data structure here.
    def __init__(self):
        self.is_string = False
        self.leaves = {}


class WordDictionary(object):
    def __init__(self):
        self.root = TrieNode()

    # @param {string} word
    # @return {void}
    # Adds a word into the data structure.
    def addWord(self, word):
        curr = self.root
        for c in word:
            if c not in curr.leaves:
                curr.leaves[c] = TrieNode()
            curr = curr.leaves[c]
        curr.is_string = True

    # @param {string} word
    # @return {boolean}
    # Returns if the word is in the data structure. A word could
    # contain the dot character '.' to represent any one letter.
    def search(self, word):
        return self.searchHelper(word, 0, self.root)

    def searchHelper(self, word, start, curr):
        if start == len(word):
            return curr.is_string
        if word[start] in curr.leaves:
            return self.searchHelper(word, start+1, curr.leaves[word[start]])
        elif word[start] == '.':
            for c in curr.leaves:
                if self.searchHelper(word, start+1, curr.leaves[c]):
                    return True

        return False
```

odd-even-jump.py

```
# odd-even-jum is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def oddEvenJumps(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        def findNext(idx):
            result = [None]*len(idx)
            stack = []
            for i in idx:
                while stack and stack[-1] < i:
                    result[stack.pop()] = i
                stack.append(i)
            return result

        idx = sorted(range(len(A)), key = lambda i: A[i])
        next_higher = findNext(idx)
        idx.sort(key = lambda i: -A[i])
        next_lower = findNext(idx)

        odd, even = [False]*len(A), [False]*len(A)
        odd[-1], even[-1] = True, True
        for i in reversed(xrange(len(A)-1)):
            if next_higher[i]:
                odd[i] = even[next_higher[i]]
            if next_lower[i]:
                even[i] = odd[next_lower[i]]
        return sum(odd)
```

find-the-distance-value-between-two-arrays.py

```python
# find-the-distance-value-between-two-arrays is not found.
# Time:  O((n + m) * logm)
# Space: O(1)

import bisect


class Solution(object):
    def findTheDistanceValue(self, arr1, arr2, d):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :type d: int
        :rtype: int
        """
        arr2.sort()
        result, i, j = 0, 0, 0
        for x in arr1:
            j = bisect.bisect_left(arr2, x)
            left = arr2[j-1] if j-1 >= 0 else float("-inf")
            right = arr2[j] if j < len(arr2) else float("inf")
            result += left+d < x < right-d
        return result


# Time:  O(nlogn + mlogm)
# Space: O(1)
class Solution2(object):
    def findTheDistanceValue(self, arr1, arr2, d):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :type d: int
        :rtype: int
        """
        arr1.sort(), arr2.sort()
        result, i, j = 0, 0, 0
        while i < len(arr1) and j < len(arr2):
            if arr1[i]-arr2[j] > d:
                j += 1
                continue
            result += arr2[j]-arr1[i] > d
            i += 1
        return result+len(arr1)-i
```

palindrome-number.py

```python
# DESC
# Follow up:
# Example 2:
# Coud you solve it without converting the integer to a string?
# Example 3:
# Determine whether an integer is a palindrome. An integer is a palindrome when it
#  reads the same backward as forward.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: 121
# Output: true
# Input: 10
# Output: false
# Explanation: Reads 01 from right to left. Therefore it i
# s not a palindrome.
# Input: -121
# Output: false
# Explanation: From left to right, it reads -121. From r
# ight to left, it becomes 121-. Therefore it is not a palindrome.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    # @return a boolean
    def isPalindrome(self, x):
        if x < 0:
            return False
        copy, reverse = x, 0

        while copy:
            reverse *= 10
            reverse += copy % 10
            copy //= 10

        return x == reverse
```

strong-password-checker.py

```
# DESC
# Insertion, deletion or replace of any one character are all considered as one change.
# Write a function strongPasswordChecker(s), that takes a string s as input, and r
# eturn the MINIMUM change required to make s a strong password. If s is already s
# trong, return 0.
# A password is considered strong if below conditions are all met:

# NOTE
# It must NOT contain three repeating characters in a row ("...aaa..." is weak, bu
# t "...aa...a..." is strong, assuming other conditions are met).
# It has at least 6 characters and at most 20 characters.
# It must contain at least one lowercase letter, at least one uppercase letter, an
# d at least one digit.

# EXAMPLE
#

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def strongPasswordChecker(self, s):
        """
        :type s: str
        :rtype: int
        """
        missing_type_cnt = 3
        if any('a' <= c <= 'z' for c in s):
            missing_type_cnt -= 1
        if any('A' <= c <= 'Z' for c in s):
            missing_type_cnt -= 1
        if any(c.isdigit() for c in s):
            missing_type_cnt -= 1

        total_change_cnt = 0
        one_change_cnt, two_change_cnt, three_change_cnt = 0, 0, 0
        i = 2
        while i < len(s):
            if s[i] == s[i-1] == s[i-2]:
                length = 2
                while i < len(s) and s[i] == s[i-1]:
                    length += 1
                    i += 1

                total_change_cnt += length / 3
                if length % 3 == 0:
                    one_change_cnt += 1
                elif length % 3 == 1:
                    two_change_cnt += 1
                else:
                    three_change_cnt += 1
            else:
                i += 1

        if len(s) < 6:
            return max(missing_type_cnt, 6 - len(s))
        elif len(s) <= 20:
            return max(missing_type_cnt, total_change_cnt)
```

```
    else:
        delete_cnt = len(s) - 20

        total_change_cnt -= min(delete_cnt, one_change_cnt * 1) / 1
        total_change_cnt -= min(max(delete_cnt - one_change_cnt, 0), two_change_cnt * 2) / 2
        total_change_cnt -= min(max(delete_cnt - one_change_cnt - 2 * two_change_cnt, 0), three_change_cnt

        return delete_cnt + max(missing_type_cnt, total_change_cnt)
```

check-if-n-and-its-double-exist.py

```
# check-if-n-and-its-double-exist is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def checkIfExist(self, arr):
        """
        :type arr: List[int]
        :rtype: bool
        """
        lookup = set()
        for x in arr:
            if 2*x in lookup or \
               (x%2 == 0 and x//2 in lookup):
                return True
            lookup.add(x)
        return False
```

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

```
# DESC
# Return an array of integers answer, where answer[i] = next_larger(node_{i+1}).
# Note that in the example inputs (not outputs) below, arrays such as [2,1,5] repr
# esent the serialization of a linked list with a head node value of 2, second nod
# e value of 1, and third node value of 5.
# Example 3:
# We are given a linked list with head as the first node.  Let's number the nodes
# in the list: node_1, node_2, node_3, ... etc.
# Note:
# Example 1:
# Example 2:
# node_i
# Each node may have a next larger value: for node_i, next_larger(node_i) is the n
# ode_j.val such that j > i, node_j.val > node_i.val, and j is the smallest possib
# le choice.  If such a j does not exist, the next larger value is 0.

# NOTE
# The given list has length in the range [0, 10000].
# 1 <= node.val <= 10^9 for each node in the linked list.

# EXAMPLE
# Input: [2,7,4,3,5]
# Output: [7,0,5,5,0]
# Input: [2,1,5]
# Output: [5,5,0]
# Input: [1,7,5,1,9,2,5,1]
# Output: [7,9,9,9,0,5,0,0]

# 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
```

encode-and-decode-strings.py

```python
# encode-and-decode-strings is not found.
# Time:  O(n)
# Space: O(1)

class Codec(object):

    def encode(self, strs):
        """Encodes a list of strings to a single string.

        :type strs: List[str]
        :rtype: str
        """
        encoded_str = ""
        for s in strs:
            encoded_str += "%0*x" % (8, len(s)) + s
        return encoded_str


    def decode(self, s):
        """Decodes a single string to a list of strings.

        :type s: str
        :rtype: List[str]
        """
        i = 0
        strs = []
        while i < len(s):
            l = int(s[i:i+8], 16)
            strs.append(s[i+8:i+8+l])
            i += 8+l
        return strs
```

patching-array.py

```python
# patching-arra is not found.
# Time:  O(s + logn), s is the number of elements in the array
# Space: O(1)

class Solution(object):
    def minPatches(self, nums, n):
        """
        :type nums: List[int]
        :type n: int
        :rtype: int
        """
        patch, miss, i = 0, 1, 0
        while miss <= n:
            if i < len(nums) and nums[i] <= miss:
                miss += nums[i]
                i += 1
            else:
                miss += miss
                patch += 1

        return patch
```

happy-number.py

```python
# DESC
# Return True if n is a happy number, and False if not.
# Example:
# Write an algorithm to determine if a number n is "happy".
# A happy number is a number defined by the following process: Starting with any p
# ositive integer, replace the number by the sum of the squares of its digits, and
#  repeat the process until the number equals 1 (where it will stay), or it loops
# endlessly in a cycle which does not include 1. Those numbers for which this proc
# ess ends in 1 are happy numbers.

# NOTE
#

# EXAMPLE
# Input: 19
# Output: true
# Explanation:
# 12 + 92 = 82
# 82 + 22 = 68
# 62 + 82 = 100
# 12
# + 02 + 02 = 1

# Time:  O(k), where k is the steps to be happy number
# Space: O(k)

class Solution(object):
    # @param {integer} n
    # @return {boolean}
    def isHappy(self, n):
        lookup = {}
        while n != 1 and n not in lookup:
            lookup[n] = True
            n = self.nextNumber(n)
        return n == 1

    def nextNumber(self, n):
        new = 0
        for char in str(n):
            new += int(char)**2
        return new
```

product-of-the-last-k-numbers.py

```python
# product-of-the-last-k-numbers is not found.
# Time:  ctor: O(1)
#        add : O(1)
#        get : O(1)
# Space: O(n)

class ProductOfNumbers(object):

    def __init__(self):
        self.__accu = [1]

    def add(self, num):
        """
        :type num: int
        :rtype: None
        """
        if not num:
            self.__accu = [1]
            return
        self.__accu.append(self.__accu[-1]*num)

    def getProduct(self, k):
        """
        :type k: int
        :rtype: int
        """
        if len(self.__accu) <= k:
            return 0
        return self.__accu[-1] // self.__accu[-1-k]
```

implement-trie-prefix-tree.py

```
# implement-trie-prefix-tree is not found.
# Time:  O(n), per operation
# Space: O(1)

class TrieNode(object):
    # Initialize your data structure here.
    def __init__(self):
        self.is_string = False
        self.leaves = {}


class Trie(object):

    def __init__(self):
        self.root = TrieNode()

    # @param {string} word
    # @return {void}
    # Inserts a word into the trie.
    def insert(self, word):
        cur = self.root
        for c in word:
            if not c in cur.leaves:
                cur.leaves[c] = TrieNode()
            cur = cur.leaves[c]
        cur.is_string = True

    # @param {string} word
    # @return {boolean}
    # Returns if the word is in the trie.
    def search(self, word):
        node = self.childSearch(word)
        if node:
            return node.is_string
        return False

    # @param {string} prefix
    # @return {boolean}
    # Returns if there is any word in the trie
    # that starts with the given prefix.
    def startsWith(self, prefix):
        return self.childSearch(prefix) is not None

    def childSearch(self, word):
        cur = self.root
        for c in word:
            if c in cur.leaves:
                cur = cur.leaves[c]
            else:
                return None
        return cur
```

friend-circles.py

```
# DESC
# There are N students in a class. Some of them are friends, while some are not. T
# heir 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 def
# ined a friend circle is a group of students who are direct or indirect friends.
# Constraints:
# Example 1:
# Given a N*N matrix M representing the friend relationship between students in th
# e class. If M[i][j] = 1, then the ith and jth students are direct friends with e
# ach other, otherwise not. And you have to output the total number of friend circ
# les among all the students.
# Example 2:

# NOTE
# 1 <= N <= 200
# M[i][j] == M[j][i]
# M[i][i] == 1

# EXAMPLE
# Input:
# [[1,1,0],
#  [1,1,1],
#  [0,1,1]]
# Output: 1
# Explanation:The 0th and 1st stud
# ents are direct friends, the 1st and 2nd students are direct friends,
# so the 0t
# h and 2nd students are indirect friends. All of them are in the same friend circ
# le, so return 1.
# Input:
# [[1,1,0],
#  [1,1,0],
#  [0,0,1]]
# Output: 2
# Explanation:The 0th and 1st stud
# ents are direct friends, so they are in a friend circle.
# The 2nd student himsel
# f is in a friend circle. So return 2.

# 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

```
# DESC
# Example 2:
# Example 4:
# Given a balanced parentheses string S, compute the score of the string based on
# the following rule:
# Note:
# Example 1:
# Example 3:

# NOTE
# () has score 1
# 2 <= S.length <= 50
# S is a balanced parentheses string, containing only ( and ).
# (A) has score 2 * A, where A is a balanced parentheses string.
# AB has score A + B, where A and B are balanced parentheses strings.

# EXAMPLE
# Input: "(())"
# Output: 2
# Input: "()()"
# Output: 2
# Input: "()"
# Output: 1
# Input: "((()(()))"
# Output: 6

# 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] == '(':
                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

```
# DESC
# Example 2:
# Given two integers n and k, you need to construct a list which contains n differ
# ent 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 - a
# 3|, |a3 - a4|, ... , |an-1 - an|] has exactly k distinct integers.
# If there are multiple answers, print any of them.
# Example 1:
# Note:

# NOTE
# The n and k are in the range 1 <= k < n <= 104.

# EXAMPLE
# Input: n = 3, k = 1
# Output: [1, 2, 3]
# Explanation: The [1, 2, 3] has three diffe
# rent positive integers ranging from 1 to 3, and the [1, 1] has exactly 1 distinc
# t integer: 1.
# Input: n = 3, k = 2
# Output: [1, 3, 2]
# Explanation: The [1, 3, 2] has three diffe
# rent positive integers ranging from 1 to 3, and the [2, 1] has exactly 2 distinc
# t integers: 1 and 2.

# 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

```
# DESC
# Constraints:
# Example 2:
# Example 1:

# NOTE
# m == mat.length
# n == mat[i].length
# 1 <= m, n, K <= 100
# 1 <= mat[i][j] <= 100

# EXAMPLE
# Input: mat = [[1,2,3],[4,5,6],[7,8,9]], K = 1
# Output: [[12,21,16],[27,45,33],[24
# ,39,28]]
# Input: mat = [[1,2,3],[4,5,6],[7,8,9]], K = 2
# Output: [[45,45,45],[45,45,45],[45
# ,45,45]]

# 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
```

bomb-enemy.py

```
# bomb-enem is not found.
# Time:  O(m * n)
# Space: O(m * n)


class Solution(object):
    def maxKilledEnemies(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        result = 0
        if not grid or not grid[0]:
            return result

        down = [[0 for _ in xrange(len(grid[0]))] for _ in xrange(len(grid))]
        right = [[0 for _ in xrange(len(grid[0]))] for _ in xrange(len(grid))]
        for i in reversed(xrange(len(grid))):
            for j in reversed(xrange(len(grid[0]))):
                if grid[i][j] != 'W':
                    if i + 1 < len(grid):
                        down[i][j] = down[i + 1][j]
                    if j + 1 < len(grid[0]):
                        right[i][j] = right[i][j + 1]
                    if grid[i][j] == 'E':
                        down[i][j] += 1
                        right[i][j] += 1

        up = [0 for _ in xrange(len(grid[0]))]
        for i in xrange(len(grid)):
            left = 0
            for j in xrange(len(grid[0])):
                if grid[i][j] == 'W':
                    up[j], left = 0, 0
                elif grid[i][j] == 'E':
                    up[j] += 1
                    left += 1
                else:
                    result = max(result,
                                 left + up[j] + right[i][j] + down[i][j])

        return result
```

excel-sheet-column-title.py

```
# DESC
# Example 2:
# For example:
# Example 3:
# Example 1:
# Given a positive integer, return its corresponding column title as appear in an
# Excel sheet.

# NOTE
#

# EXAMPLE
# Input: 701
# Output: "ZY"
# 1 -> A
#     2 -> B
#     3 -> C
#     ...
#     26 -> Z
#     27 -> AA
#     28 -> AB
#     ...
# Input: 1
# Output: "A"
# Input: 28
# Output: "AB"

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def convertToTitle(self, n):
        """
        :type n: int
        :rtype: str
        """
        result, dvd = "", n

        while dvd:
            result += chr((dvd - 1) % 26 + ord('A'))
            dvd = (dvd - 1) / 26

        return result[::-1]
```

maximum-of-absolute-value-expression.py

```python
# DESC
# |arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|
# |arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|
# Example 2:
# Constraints:
# Example 1:
# Given two arrays of integers with equal lengths, return the maximum value of:
# where the maximum is taken over all 0 <= i, j < arr1.length.

# NOTE
# -10^6 <= arr1[i], arr2[i] <= 10^6
# 2 <= arr1.length == arr2.length <= 40000

# EXAMPLE
# Input: arr1 = [1,2,3,4], arr2 = [-1,4,5,6]
# Output: 13
# Input: arr1 = [1,-2,-5,0,10], arr2 = [0,-2,-1,-7,-4]
# Output: 20

# 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):
        """
        :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])
```

design-hashmap.py

```
# design-hashma is not found.
# Time:  O(1)
# Space: O(n)

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:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        node.next, node.prev = None, None  # make node clean

    def find(self, key):
        curr = self.head
        while curr:
            if curr.key == key:
                break
            curr = curr.next
        return curr


class MyHashMap(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__data = [LinkedList() for _ in xrange(10000)]

    def put(self, key, value):
        """
        value will always be positive.
        :type key: int
```

```
        :type value: int
        :rtype: void
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        if node:
            node.val = value
        else:
            l.insert(ListNode(key, value))

    def get(self, key):
        """
        Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the
        :type key: int
        :rtype: int
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        if node:
            return node.val
        else:
            return -1

    def remove(self, key):
        """
        Removes the mapping of the specified value key if this map contains a mapping for the key
        :type key: int
        :rtype: void
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        if node:
            l.delete(node)
```

course-schedule.py

```
# DESC
# Example 2:
# Some courses may have prerequisites, for example to take course 0 you have to fi
# rst take course 1, which is expressed as a pair: [0,1]
# Given the total number of courses and a list of prerequisite pairs, is it possib
# le for you to finish all courses?
# Example 1:
# Constraints:
# There are a total of numCourses courses you have to take, labeled from 0 to numC
# ourses-1.
# [0,1]

# NOTE
# You may assume that there are no duplicate edges in the input prerequisites.
# 1 <= numCourses <= 10^5
# The input prerequisites is a graph represented by a list of edges, not adjacency
#  matrices. Read more about how a graph is represented.

# EXAMPLE
# Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
# Output: false
# Explanation:
# There are a total of 2 courses to take.
#              To take course 1 you shoul
# d have finished course 0, and to take course 0 you should
#              also have
#  finished course 1. So it is impossible.
# Input: numCourses = 2, prerequisites = [[1,0]]
# Output: true
# Explanation: There a
# re a total of 2 courses to take.
#              To take course 1 you should have
# finished course 0. So it is possible.

# Time:  O(|V| + |E|)
# Space: O(|E|)

from collections import defaultdict, deque


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

        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()

    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 False

return True
```

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

```
# single-element-in-a-sorted-arra is not found.
# 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]
```

queens-that-can-attack-the-king.py

```
# queens-that-can-attack-the-king is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def queensAttacktheKing(self, queens, king):
        """
        :type queens: List[List[int]]
        :type king: List[int]
        :rtype: List[List[int]]
        """
        dirctions = [(-1, 0), (0, 1), (1, 0), (0, -1),
                     (-1, 1), (1, 1), (1, -1), (-1, -1)]
        result = []
        lookup = {(i, j) for i, j in queens}
        for dx, dy in dirctions:
            for i in xrange(1, 8):
                x, y = king[0] + dx*i, king[1] + dy*i
                if (x, y) in lookup:
                    result.append([x, y])
                    break
        return result
```

minimum-cost-to-hire-k-workers.py

```
# DESC
# Now we want to hire exactly K workers to form a paid group.  When hiring a group
#  of K workers, we must pay them according to the following rules:
# There are N workers.  The i-th worker has a quality[i] and a minimum wage expect
# ation wage[i].
# Return the least amount of money needed to form a paid group satisfying the abov
# e conditions.
# Note:
# Example 2:
# Example 1:

# NOTE
# Every worker in the paid group should be paid in the ratio of their quality comp
# ared to other workers in the paid group.
# 1 <= quality[i] <= 10000
# 1 <= K <= N <= 10000, where N = quality.length = wage.length
# 1 <= wage[i] <= 10000
# Answers within 10^-5 of the correct answer will be considered correct.
# Every worker in the paid group must be paid at least their minimum wage expectation.

# EXAMPLE
# Input: quality = [10,20,5], wage = [70,50,30], K = 2
# Output: 105.00000
# Explanati
# on: We pay 70 to 0-th worker and 35 to 2-th worker.
# Input: quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3
# Output: 30.66667
# Expla
# nation: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.

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

import itertools
import heapq


class Solution(object):
    def mincostToHireWorkers(self, quality, wage, K):
        """
        :type quality: List[int]
        :type wage: List[int]
        :type K: int
        :rtype: float
        """
        result, qsum = float("inf"), 0
        max_heap = []
        for r, q in sorted([float(w)/q, q] for w, q in itertools.izip(wage, quality)):
            qsum += q
            heapq.heappush(max_heap, -q)
            if len(max_heap) > K:
                qsum -= -heapq.heappop(max_heap)
            if len(max_heap) == K:
                result = min(result, qsum*r)
        return result
```

group-anagrams.py

```python
# DESC
# Note:
# Example:
# Given an array of strings, group anagrams together.

# NOTE
# The order of your output does not matter.
# All inputs will be in lowercase.

# EXAMPLE
# Input: ["eat", "tea", "tan", "ate", "nat", "bat"],
# Output:
# [
#   ["ate","eat","tea
# "],
#   ["nat","tan"],
#   ["bat"]
# ]

# 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

```
# DESC
# Example 2:
# Example 1:
# Note:
# Now, we may draw connecting lines: a straight line connecting two numbers A[i] a
# nd B[j] such that:
# Return the maximum number of connecting lines we can draw in this way.
# Example 3:
# We write the integers of A and B (in the order they are given) on two separate h
# orizontal lines.
# Note that a connecting lines cannot intersect even at the endpoints: each number
#  can only belong to one connecting line.

# NOTE
# A[i] == B[j];
# 1 <= B.length <= 500
# The line we draw does not intersect any other connecting (non-horizontal) line.
# 1 <= A.length <= 500
# 1 <= A[i], B[i] <= 2000

# EXAMPLE
# Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]
# Output: 2
# Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]
# Output: 3
# Input: A = [1,4,2], B = [1,2,4]
# Output: 2
# Explanation: We can draw 2 uncrossed l
# ines 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.

# 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]
        :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)]
```

letter-case-permutation.py

```
# DESC
# Constraints:
# Given a string S, we can transform every letter individually to be lowercase or
# uppercase to create another string.  Return a list of all possible strings we co
# uld create.

# NOTE
# S will consist only of letters or digits.
# S will be a string with length between 1 and 12.

# EXAMPLE
# Examples:
# Input: S = "a1b2"
# Output: ["a1b2", "a1B2", "A1b2", "A1B2"]
#
# Input: S =
#  "3z4"
# Output: ["3z4", "3Z4"]
#
# Input: S = "12345"
# Output: ["12345"]

# Time:  O(n * 2^n)
# Space: O(n * 2^n)

class Solution(object):
    def letterCasePermutation(self, S):
        """
        :type S: str
        :rtype: List[str]
        """
        result = [[]]
        for c in S:
            if c.isalpha():
                for i in xrange(len(result)):
                    result.append(result[i][:])
                    result[i].append(c.lower())
                    result[-1].append(c.upper())
            else:
                for s in result:
                    s.append(c)
        return map("".join, result)
```

partition-array-into-three-parts-with-equal-sum.py

```
# DESC
# Given an array A of integers, return true if and only if we can partition the ar
# ray into three non-empty parts with equal sums.
# Constraints:
# Example 2:
# Formally, we can partition the array if we can find indexes i+1 < j with (A[0] +
#  A[1] + ... + A[i] == A[i+1] + A[i+2] + ... + A[j-1] == A[j] + A[j-1] + ... + A[
# A.length - 1])
# Example 1:
# Example 3:
# i+1 < j

# NOTE
# -10^4 <= A[i] <= 10^4
# 3 <= A.length <= 50000

# EXAMPLE
# Input: A = [0,2,1,-6,6,-7,9,1,2,0,1]
# Output: true
# Explanation: 0 + 2 + 1 = -6 +
# 6 - 7 + 9 + 1 = 2 + 0 + 1
# Input: A = [3,3,6,5,-2,2,5,1,-9,4]
# Output: true
# Explanation: 3 + 3 = 6 = 5 - 2 +
#  2 + 5 + 1 - 9 + 4
# Input: A = [0,2,1,-6,6,7,9,-1,2,0,1]
# Output: false

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canThreePartsEqualSum(self, A):
        """
        :type A: List[int]
        :rtype: bool
        """
        total = sum(A)
        if total % 3 != 0:
            return False
        parts, curr = 0, 0
        for x in A:
            curr += x
            if curr == total//3:
                parts += 1
                curr = 0
        return parts >= 3
```

find-the-winner-of-an-array-game.py

```python
# find-the-winner-of-an-array-game is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def getWinner(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: int
        """
        result = arr[0]
        count = 0
        for i in xrange(1, len(arr)):
            if arr[i] > result:
                result = arr[i]
                count = 0
            count += 1
            if (count == k):
                break
        return result
```

web-crawler-multithreaded.py

```
# web-crawler-multithreaded is not found.
# Time:  O(|V| + |E|)
# Space: O(|V|)

import threading
import Queue


# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
class HtmlParser(object):
   def getUrls(self, url):
       """
       :type url: str
       :rtype List[str]
       """
       pass


class Solution(object):
    NUMBER_OF_WORKERS = 8

    def __init__(self):
        self.__cv = threading.Condition()
        self.__q = Queue.Queue()

    def crawl(self, startUrl, htmlParser):
        """
        :type startUrl: str
        :type htmlParser: HtmlParser
        :rtype: List[str]
        """
        SCHEME = "http://"
        def hostname(url):
            pos = url.find('/', len(SCHEME))
            if pos == -1:
                return url
            return url[:pos]

        def worker(htmlParser, lookup):
            while True:
                from_url = self.__q.get()
                if from_url is None:
                    break
                name = hostname(from_url)
                for to_url in htmlParser.getUrls(from_url):
                    if name != hostname(to_url):
                        continue
                    with self.__cv:
                        if to_url not in lookup:
                            lookup.add(to_url)
                            self.__q.put(to_url)
                self.__q.task_done()

        workers = []
        self.__q = Queue.Queue()
```

```
        self.__q.put(startUrl)
        lookup = set([startUrl])
        for i in xrange(self.NUMBER_OF_WORKERS):
            t = threading.Thread(target=worker, args=(htmlParser, lookup))
            t.start()
            workers.append(t)
        self.__q.join()
        for t in workers:
            self.__q.put(None)
        for t in workers:
            t.join()
        return list(lookup)


# Time:  O(|V| + |E|)
# Space: O(|V|)
import threading
import collections


class Solution2(object):
    NUMBER_OF_WORKERS = 8

    def __init__(self):
        self.__cv = threading.Condition()
        self.__q = collections.deque()
        self.__working_count = 0

    def crawl(self, startUrl, htmlParser):
        """
        :type startUrl: str
        :type htmlParser: HtmlParser
        :rtype: List[str]
        """
        SCHEME = "http://"
        def hostname(url):
            pos = url.find('/', len(SCHEME))
            if pos == -1:
                return url
            return url[:pos]

        def worker(htmlParser, lookup):
            while True:
                with self.__cv:
                    while not self.__q:
                        self.__cv.wait()
                    from_url = self.__q.popleft()
                    if from_url is None:
                        break
                    self.__working_count += 1
                name = hostname(from_url)
                for to_url in htmlParser.getUrls(from_url):
                    if name != hostname(to_url):
                        continue
                    with self.__cv:
                        if to_url not in lookup:
                            lookup.add(to_url)
                            self.__q.append(to_url)
                            self.__cv.notifyAll()
                with self.__cv:
```

```
            self.__working_count -= 1
            if not self.__q and not self.__working_count:
                self.__cv.notifyAll()

workers = []
self.__q = collections.deque([startUrl])
lookup = set([startUrl])
for i in xrange(self.NUMBER_OF_WORKERS):
    t = threading.Thread(target=worker, args=(htmlParser, lookup))
    t.start()
    workers.append(t)
with self.__cv:
    while self.__q or self.__working_count:
        self.__cv.wait()
    for i in xrange(self.NUMBER_OF_WORKERS):
        self.__q.append(None)
    self.__cv.notifyAll()
for t in workers:
    t.join()
return list(lookup)
```

longest-valid-parentheses.py

```
# DESC
# Example 1:
# Example 2:
# Given a string containing just the characters '(' and ')', find the length of th
# e longest valid (well-formed) parentheses substring.

# NOTE
#

# EXAMPLE
# Input: "(()"
# Output: 2
# Explanation: The longest valid parentheses substring is "()"
# Input: ")()())"
# Output: 4
# Explanation: The longest valid parentheses substring i
# s "()()"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def longestValidParentheses(self, s):
        """
        :type s: str
        :rtype: int
        """
        def length(it, start, c):
            depth, longest = 0, 0
            for i in it:
                if s[i] == c:
                    depth += 1
                else:
                    depth -= 1
                    if depth < 0:
                        start, depth = i, 0
                    elif depth == 0:
                        longest = max(longest, abs(i - start))
            return longest

        return max(length(xrange(len(s)), -1, '(', \
                   length(reversed(xrange(len(s))), len(s), ')'))


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param s, a string
    # @return an integer
    def longestValidParentheses(self, s):
        longest, last, indices = 0, -1, []
        for i in xrange(len(s)):
            if s[i] == '(':
                indices.append(i)
            elif not indices:
                last = i
            else:
                indices.pop()
```

```
        if not indices:
            longest = max(longest, i - last)
        else:
            longest = max(longest, i - indices[-1])
    return longest
```

smallest-range-i.py

```
# DESC
# Return the smallest possible difference between the maximum value of B and the m
# inimum value of B.
# Note:
# Example 3:
# After this process, we have some array B.
# Given an array A of integers, for each integer A[i] we may choose any x with -K
# <= x <= K, and add x to A[i].
# Example 2:
# Example 1:

# NOTE
# 0 <= A[i] <= 10000
# 0 <= K <= 10000
# 1 <= A.length <= 10000

# EXAMPLE
# Input: A = [1], K = 0
# Output: 0
# Explanation: B = [1]
# Input: A = [0,10], K = 2
# Output: 6
# Explanation: B = [2,8]
# Input: A = [1,3,6], K = 3
# Output: 0
# Explanation: B = [3,3,3] or B = [4,4,4]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def smallestRangeI(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        return max(0, max(A) - min(A) - 2*K)
```

partition-list.py

```python
# DESC
# Example:
# You should preserve the original relative order of the nodes in each of the two
# partitions.
# 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.

# NOTE
#

# 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
```

asteroid-collision.py

```
# DESC
# Example 4:
# For each asteroid, the absolute value represents its size, and the sign represen
# ts its direction (positive meaning right, negative meaning left).  Each asteroid
#  moves at the same speed.
# Note:
# Example 3:
# Find out the state of the asteroids after all collisions.  If two asteroids meet
# , the smaller one will explode.  If both are the same size, both will explode.
# Two asteroids moving in the same direction will never meet.
# Example 1:
# Example 2:
# We are given an array asteroids of integers representing asteroids in a row.

# NOTE
# Each asteroid will be a non-zero integer in the range [-1000, 1000]..
# The length of asteroids will be at most 10000.

# EXAMPLE
# Input:
# asteroids = [8, -8]
# Output: []
# Explanation:
# The 8 and -8 collide explod
# ing each other.
# Input:
# asteroids = [5, 10, -5]
# Output: [5, 10]
# Explanation:
# The 10 and -5 coll
# ide resulting in 10.  The 5 and 10 never collide.
# Input:
# asteroids = [-2, -1, 1, 2]
# Output: [-2, -1, 1, 2]
# Explanation:
# The -2 a
# nd -1 are moving left, while the 1 and 2 are moving right.
# Asteroids moving the
# same direction never meet, so no asteroids will meet each other.
# Input:
# asteroids = [10, 2, -5]
# Output: [10]
# Explanation:
# The 2 and -5 collide
# resulting in -5.  The 10 and -5 collide resulting in 10.

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


class Solution(object):
    def asteroidCollision(self, asteroids):
        """
        :type asteroids: List[int]
        :rtype: List[int]
        """
        result = []
        for asteroid in asteroids:
```

```
    while result and asteroid < 0 < result[-1]:
        if result[-1] < -asteroid:
            result.pop()
            continue
        elif result[-1] == -asteroid:
            result.pop()
        break
    else:
        result.append(asteroid)
return result
```

number-of-good-ways-to-split-a-string.py

```python
# number-of-good-ways-to-split-a-string is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def numSplits(self, s):
        """
        :type s: str
        :rtype: int
        """
        left_count, right_count = collections.Counter(), collections.Counter(s)
        result = 0
        for c in s:
            left_count[c] += 1
            right_count[c] -= 1
            if not right_count[c]:
                del right_count[c]
            if len(left_count) == len(right_count):
                result += 1
        return result
```

student-attendance-record-i.py

```
# DESC
# Example 1:
# You need to return whether the student could be rewarded according to his attend
# ance record.
# A student could be rewarded if his attendance record doesn't contain more than o
# ne 'A' (absent) or more than two continuous 'L' (late).
# Example 2:

# NOTE
# 'L' : Late.
# 'P' : Present.
# 'A' : Absent.

# EXAMPLE
# Input: "PPALLP"
# Output: True
# Input: "PPALLL"
# Output: False

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def checkRecord(self, s):
        """
        :type s: str
        :rtype: bool
        """
        count_A = 0
        for i in xrange(len(s)):
            if s[i] == 'A':
                count_A += 1
                if count_A == 2:
                    return False
            if i < len(s) - 2 and s[i] == s[i+1] == s[i+2] == 'L':
                return False
        return True
```

lonely-pixel-i.py

```
# lonely-pixel-i is not found.
# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    def findLonelyPixel(self, picture):
        """
        :type picture: List[List[str]]
        :rtype: int
        """
        rows, cols = [0] * len(picture),  [0] * len(picture[0])
        for i in xrange(len(picture)):
            for j in xrange(len(picture[0])):
                if picture[i][j] == 'B':
                    rows[i] += 1
                    cols[j] += 1

        result = 0
        for i in xrange(len(picture)):
            if rows[i] == 1:
                for j in xrange(len(picture[0])):
                     result += picture[i][j] == 'B' and cols[j] == 1
        return result


class Solution2(object):
    def findLonelyPixel(self, picture):
        """
        :type picture: List[List[str]]
        :type N: int
        :rtype: int
        """
        return sum(col.count('B') == 1 == picture[col.index('B')].count('B') \
               for col in zip(*picture))
```

valid-parentheses.py

```python
# DESC
# Example 1:
# Given a string containing just the characters '(', ')', '{', '}', '[' and ']', d
# etermine if the input string is valid.
# Example 3:
# An input string is valid if:
# Example 4:
# Note that an empty string is also considered valid.
# Example 5:
# Example 2:

# NOTE
# Open brackets must be closed by the same type of brackets.
# Open brackets must be closed in the correct order.

# EXAMPLE
# Input: "([)]"
# Output: false
# Input: "()"
# Output: true
# Input: "()[]{}"
# Output: true
# Input: "{[]}"
# Output: true
# Input: "(]"
# Output: false

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

class Solution(object):
    # @return a boolean
    def isValid(self, s):
        stack, lookup = [], {"(": ")", "{": "}", "[": "]"}
        for parenthese in s:
            if parenthese in lookup:
                stack.append(parenthese)
            elif len(stack) == 0 or lookup[stack.pop()] != parenthese:
                return False
        return len(stack) == 0
```

valid-mountain-array.py

```python
# valid-mountain-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def validMountainArray(self, A):
        """
        :type A: List[int]
        :rtype: bool
        """
        i = 0
        while i+1 < len(A) and A[i] < A[i+1]:
            i += 1
        j = len(A)-1
        while j-1 >= 0 and A[j-1] > A[j]:
            j -= 1
        return 0 < i == j < len(A)-1
```

filling-bookcase-shelves.py

```
# DESC
# Constraints:
# We want to place these books in order onto bookcase shelves that have total widt
# h shelf_width.
# We choose some of the books to place on this shelf (such that the sum of their t
# hickness is <= shelf_width), then build another level of shelf of the bookcase s
# o that the total height of the bookcase has increased by the maximum height of t
# he books we just put down.  We repeat this process until there are no more books
#  to place.
# We have a sequence of books: the i-th book has thickness books[i][0] and height
# books[i][1].
# Note again that at each step of the above process, the order of the books we pla
# ce 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 firs
# t shelf, the third book on the second shelf, and the fourth and fifth book on th
# e last shelf.
# Return the minimum possible height that the total bookshelf can be after placing
#  shelves in this manner.
# Example 1:
# shelf_width

# NOTE
# 1 <= books[i][0] <= shelf_width <= 1000
# 1 <= books[i][1] <= 1000
# 1 <= books.length <= 1000

# EXAMPLE
# Input: books = [[1,1],[2,3],[2,3],[1,1],[1,1],[1,1],[1,2]], shelf_width = 4
# Outp
# ut: 6
# Explanation:
# The sum of the heights of the 3 shelves are 1 + 3 + 2 = 6.
# No
# tice that book number 2 does not have to be on the first shelf.

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

masking-personal-information.py

```
# DESC
# To mask an email, all names must be converted to lowercase and all letters betwe
# en the first and last letter of the first name must be replaced by 5 asterisks '
# *'.
# 1. Email address:
# We are given a personal information string S, which may represent either an emai
# l address or a phone number.
# Note that extraneous characters like "(", ")", " ", as well as extra dashes or p
# lus signs not part of the above formatting scheme should be removed.
# An email address starts with a name, followed by the symbol '@', followed by a n
# ame, followed by the dot '.' and followed by a name.
# Return the correct "mask" of the information provided.
# Example 4:
# To mask a phone number with country code like "+111 111 111 1111", we write it i
# n the form "+***-***-***-1111".  The '+' sign and the first '-' sign before the
# local number should only exist if there is a country code.  For example, a 12 di
# git phone number mask should start with "+**-".
# A phone number is a string consisting of only the digits 0-9 or the characters f
# rom the set {'+', '-', '(', ')', ' '}. You may assume a phone number contains 10
#  to 13 digits.
# The local number should be formatted and masked as "***-***-1111", where 1 repre
# sents the exposed digits.
# All email addresses are guaranteed to be valid and in the format of "name1@name2
# .name3".
# Example 1:
# Example 3:
# We would like to mask this personal information according to the following rules:
# The last 10 digits make up the local number, while the digits before those make
# up the country code. Note that the country code is optional. We want to expose o
# nly the last 4 digits and mask all other digits.
# We define a name to be a string of length  2 consisting of only lowercase lette
# rs a-z or uppercase letters A-Z.
# Example 2:
# 2. Phone number:
# Notes:

# NOTE
# Phone numbers have length at least 10.
# S.length <= 40.
# Emails have length at least 8.

# EXAMPLE
# Input: "LeetCode@LeetCode.com"
# Output: "l******e@leetcode.com"
# Explanation: All n
# ames are converted to lowercase, and the letters between the
#              first
# and last letter of the first name is replaced by 5 asterisks.
#              There
# fore, "leetcode" -> "l******e".
# Input: "AB@qq.com"
# Output: "a*****b@qq.com"
# Explanation: There must be 5 asteris
# ks between the first and last letter
#              of the first name "ab". There
# fore, "ab" -> "a*****b".
# Input: "86-(10)12345678"
# Output: "+**-***-***-5678"
```

```
# Explanation: 12 digits, 2 di
# gits for country code and 10 digits for local number.
# Input: "1(234)567-890"
# Output: "***-***-7890"
# Explanation: 10 digits in the phon
# e number, which means all digits make up the local number.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def maskPII(self, S):
        """
        :type S: str
        :rtype: str
        """
        if '@' in S:
            first, after = S.split('@')
            return "{}*****{}@{}".format(first[0], first[-1], after).lower()

        digits = filter(lambda x: x.isdigit(), S)
        local = "***-***-{}".format(digits[-4:])
        if len(digits) == 10:
            return local
        return "+{}-{}".format('*' * (len(digits) - 10), local)
```

element-appearing-more-than-25-in-sorted-array.py

```
# element-appearing-more-than-25-in-sorted-arra is not found.
# Time:  O(logn)
# Space: O(1)

import bisect


class Solution(object):
    def findSpecialInteger(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        for x in [arr[len(arr)//4], arr[len(arr)//2], arr[len(arr)*3//4]]:
            if (bisect.bisect_right(arr, x) - bisect.bisect_left(arr, x)) * 4 > len(arr):
                return x
        return -1
```

burst-balloons.py

```
# DESC
# Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number o
# n it represented by array nums. You are asked to burst all the balloons. If the
# you burst balloon i you will get nums[left] * nums[i] * nums[right] coins. Here
# left and right are adjacent indices of i. After the burst, the left and right th
# en becomes adjacent.
# Find the maximum coins you can collect by bursting the balloons wisely.
# Note:
# Example:

# NOTE
# 0  n  500, 0  nums[i]  100
# You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not
# burst them.

# EXAMPLE
# Input: [3,1,5,8]
# Output: 167
# Explanation: nums = [3,1,5,8] --> [3,5,8] -->   [3
# ,8]   -->  [8]  --> []
#              coins =  3*1*5      +  3*5*8    +  1*3*8
#   + 1*8*1   = 167

# Time:  O(n^3)
# Space: O(n^2)


class Solution(object):
    def maxCoins(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        coins = [1] + [i for i in nums if i > 0] + [1]
        n = len(coins)
        max_coins = [[0 for _ in xrange(n)] for _ in xrange(n)]

        for k in xrange(2, n):
            for left in xrange(n - k):
                right = left + k
                for i in xrange(left + 1, right):
                    max_coins[left][right] = \
                        max(max_coins[left][right],
                            coins[left] * coins[i] * coins[right] +
                            max_coins[left][i] +
                            max_coins[i][right])

        return max_coins[0][-1]
```

basic-calculator.py

```
# DESC
# Example 1:
# The expression string may contain open ( and closing parentheses ), the plus + o
# r minus sign -, non-negative integers and empty spaces  .
# Implement a basic calculator to evaluate a simple expression string.
# Example 3:
# Example 2:

# NOTE
# You may assume that the given expression is always valid.
# Do not use the eval built-in library function.

# EXAMPLE
# Input: "1 + 1"
# Output: 2
# Input: " 2-1 + 2 "
# Output: 3
# Input: "(1+(4+5+2)-3)+(6+8)"
# Output: 23

# 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] == '(':
                while operators[-1] != ')':
                    self.compute(operands, operators)
                operators.pop()

        while operators:
            self.compute(operands, operators)

        return operands[-1]

    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)
```

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

```
# DESC
# Example 2:
# In case of a tie, return the minimum such integer.
# Example 1:
# Constraints:
# Example 3:
# Given an integer array arr and a target value target, return the integer value s
# uch 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.
# Notice that the answer is not neccesarilly a number from arr.

# NOTE
# 1 <= arr[i], target <= 10^5
# 1 <= arr.length <= 10^4

# EXAMPLE
# Input: arr = [4,9,3], target = 10
# Output: 3
# Explanation: When using 3 arr conver
# ts to [3, 3, 3] which sums 9 and that's the optimal answer.
# Input: arr = [60864,25176,27249,21296,20204], target = 56803
# Output: 11361
# Input: arr = [2,3,5], target = 10
# Output: 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))


# 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
# 2-keys-keyboard is not found.
# 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
```

base-7.py

```
# DESC
# Example 2:
# Note:
# The input will be in range of [-1e7, 1e7].
# Example 1:
# Given an integer, return its base 7 string representation.

# NOTE
#

# EXAMPLE
# Input: 100
# Output: "202"
# Input: -7
# Output: "-10"

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def convertToBase7(self, num):
        if num < 0:
            return '-' + self.convertToBase7(-num)
        result = ''
        while num:
            result = str(num % 7) + result
            num //= 7
        return result if result else '0'


class Solution2(object):
    def convertToBase7(self, num):
        """
        :type num: int
        :rtype: str
        """
        if num < 0:
            return '-' + self.convertToBase7(-num)
        if num < 7:
            return str(num)
        return self.convertToBase7(num // 7) + str(num % 7)
```

final-prices-with-a-special-discount-in-a-shop.py

```python
# final-prices-with-a-special-discount-in-a-sho is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def finalPrices(self, prices):
        """
        :type prices: List[int]
        :rtype: List[int]
        """
        stk = []
        for i, p in enumerate(prices):
            while stk and prices[stk[-1]] >= p:
                prices[stk.pop()] -= p
            stk.append(i)
        return prices
```

count-vowels-permutation.py

```
# count-vowels-permutation is not found.
# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def countVowelPermutation(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)) % MOD \
                     for col in ZB] for row in A]

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


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def countVowelPermutation(self, n):
        """
        :type n: int
        :rtype: int
        """
        MOD = 10**9 + 7
        a, e, i, o, u = 1, 1, 1, 1, 1
        for _ in xrange(1, n):
            a, e, i, o, u = (e+i+u) % MOD, (a+i) % MOD, (e+o) % MOD, i, (i+o) % MOD
        return (a+e+i+o+u) % MOD
```

line-reflection.py

```
# line-reflection is not found.
# Time:  O(n)
# Space: O(n)

import collections


# Hash solution.
class Solution(object):
    def isReflected(self, points):
        """
        :type points: List[List[int]]
        :rtype: bool
        """
        if not points:
            return True
        groups_by_y = collections.defaultdict(set)
        left, right = float("inf"), float("-inf")
        for p in points:
            groups_by_y[p[1]].add(p[0])
            left, right = min(left, p[0]), max(right, p[0])
        mid = left + right
        for group in groups_by_y.values():
            for x in group:
                if mid - x not in group:
                    return False
        return True


# Time:  O(nlogn)
# Space: O(n)
# Two pointers solution.
class Solution2(object):
    def isReflected(self, points):
        """
        :type points: List[List[int]]
        :rtype: bool
        """
        if not points:
            return True
        points.sort()
        # Space: O(n)
        points[len(points)/2:] = sorted(points[len(points)/2:], \
                                        lambda x, y: y[1] - x[1] if x[0] == y[0] else \
                                                     x[0] - y[0])
        mid = points[0][0] + points[-1][0]
        left, right = 0, len(points) - 1
        while left <= right:
            if (mid != points[left][0] + points[right][0]) or \
               (points[left][0] != points[right][0] and \
                points[left][1] != points[right][1]):
                return False
            left += 1
            right -= 1
        return True
```

24-game.py

```
# 24-game is not found.
# Time:  O(n^3 * 4^n) = O(1), n = 4
# Space: O(n^2) = O(1)

from operator import add, sub, mul, truediv
from fractions import Fraction


class Solution(object):
    def judgePoint24(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        if len(nums) == 1:
            return abs(nums[0]-24) < 1e-6
        ops = [add, sub, mul, truediv]
        for i in xrange(len(nums)):
            for j in xrange(len(nums)):
                if i == j:
                    continue
                next_nums = [nums[k] for k in xrange(len(nums)) if i != k != j]
                for op in ops:
                    if ((op is add or op is mul) and j > i) or \
                       (op == truediv and nums[j] == 0):
                        continue
                    next_nums.append(op(nums[i], nums[j]))
                    if self.judgePoint24(next_nums):
                        return True
                    next_nums.pop()
        return False


# Time:  O(n^3 * 4^n) = O(1), n = 4
# Space: O(n^2) = O(1)
class Solution2(object):
    def judgePoint24(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        def dfs(nums):
            if len(nums) == 1:
                return nums[0] == 24
            ops = [add, sub, mul, truediv]
            for i in xrange(len(nums)):
                for j in xrange(len(nums)):
                    if i == j:
                        continue
                    next_nums = [nums[k] for k in xrange(len(nums))
                                 if i != k != j]
                    for op in ops:
                        if ((op is add or op is mul) and j > i) or \
                           (op == truediv and nums[j] == 0):
                            continue
                        next_nums.append(op(nums[i], nums[j]))
                        if dfs(next_nums):
                            return True
```

```
                next_nums.pop()
        return False

    return dfs(map(Fraction, nums))
```

binary-search-tree-iterator.py

```
# DESC
# next()
# Note:
# Calling next() will return the next smallest number in the BST.
# Implement an iterator over a binary search tree (BST). Your iterator will be ini
# tialized with the root node of a BST.
# Example:

# 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.

# EXAMPLE
# BSTIterator iterator = new BSTIterator(root);
# iterator.next();    // return 3
# it
# erator.next();    // return 7
# iterator.hasNext(); // return true
# iterator.next()
# ;    // return 9
# iterator.hasNext(); // return true
# iterator.next();    // retur
# n 15
# iterator.hasNext(); // return true
# iterator.next();    // return 20
# iterato
# r.hasNext(); // return false

# 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
```

contains-duplicate.py

```python
# DESC
# Your function should return true if any value appears at least twice in the arra
# y, and it should return false if every element is distinct.
# Given an array of integers, find if the array contains any duplicates.
# Example 3:
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: [1,2,3,1]
# Output: true
# Input: [1,2,3,4]
# Output: false
# Input: [1,1,1,3,3,4,3,2,4,2]
# Output: true

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

class Solution(object):
    # @param {integer[]} nums
    # @return {boolean}
    def containsDuplicate(self, nums):
        return len(nums) > len(set(nums))
```

minimum-depth-of-binary-tree.py

```
# DESC
# Note: A leaf is a node with no children.
# Given binary tree [3,9,20,null,null,15,7],
# Given a binary tree, find its minimum depth.
# Example:
# return its minimum depth = 2.
# The minimum depth is the number of nodes along the shortest path from the root n
# ode down to the nearest leaf node.

# NOTE
#

# EXAMPLE
# 3
#    / \
#   9  20
#     /  \
#    15   7

# 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 minDepth(self, root):
        if root is None:
            return 0

        if root.left and root.right:
            return min(self.minDepth(root.left), self.minDepth(root.right)) + 1
        else:
            return max(self.minDepth(root.left), self.minDepth(root.right)) + 1
```

find-the-duplicate-number.py

```
# DESC
# Example 1:
# Given an array nums containing n + 1 integers where each integer is between 1 an
# d n (inclusive), prove that at least one duplicate number must exist. Assume tha
# t there is only one duplicate number, find the duplicate one.
# Example 2:
# Note:

# NOTE
# There is only one duplicate number in the array, but it could be repeated more t
# han once.
# You must not modify the array (assume the array is read only).
# Your runtime complexity should be less than O(n2).
# You must use only constant, O(1) extra space.

# EXAMPLE
# Input: [1,3,4,2,2]
# Output: 2
# Input: [3,1,3,4,2]
# Output: 3

# 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
```

```
        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

```
# DESC
# Given a collection of intervals, merge all overlapping intervals.
# NOTE: input types have been changed on April 15, 2019. Please reset to default c
# ode definition to get new method signature.
# Example 1:
# Example 2:
# Constraints:

# NOTE
# intervals[i][0] <= intervals[i][1]

# EXAMPLE
# Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
# Output: [[1,6],[8,10],[15,18]]
# E
# xplanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
# Input: intervals = [[1,4],[4,5]]
# Output: [[1,5]]
# Explanation: Intervals [1,4] an
# d [4,5] are considered overlapping.

# 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
```

recover-binary-search-tree.py

```
# DESC
# Follow up:
# Recover the tree without changing its structure.
# Two elements of a binary search tree (BST) are swapped by mistake.
# Example 1:
# Example 2:

# NOTE
# A solution using O(n) space is pretty straight forward.
# Could you devise a constant space solution?

# EXAMPLE
# Input: [3,1,4,null,null,2]
#
#   3
#  / \
# 1   4
#    /
#   2
#
# Output: [2,1,4,null,null,3]
#
#
#   2
#  / \
# 1   4
#    /
#   3
# Input: [1,3,null,null,2]
#
#    1
#   /
#  3
#   \
#    2
#
# Output: [3,1,null,null,2]
#
#    3
#
#   /
#  1
#   \
#    2

# Time:  O(n)
# Space: O(1)

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)

        else:
            return None

class Solution(object):
    # @param root, a tree node
    # @return a tree node
    def recoverTree(self, root):
        return self.MorrisTraversal(root)

    def MorrisTraversal(self, root):
        if root is None:
            return
        broken = [None, None]
        pre, cur = None, root

        while cur:
            if cur.left is None:
                self.detectBroken(broken, pre, cur)
                pre = cur
                cur = cur.right
            else:
                node = cur.left
                while node.right and node.right != cur:
                    node = node.right

                if node.right is None:
                    node.right =cur
                    cur = cur.left
                else:
                    self.detectBroken(broken, pre, cur)
                    node.right = None
                    pre = cur
                    cur = cur.right

        broken[0].val, broken[1].val = broken[1].val, broken[0].val

        return root

    def detectBroken(self, broken, pre, cur):
        if pre and pre.val > cur.val:
            if broken[0] is None:
                broken[0] = pre
```

```
broken[1] = cur
```

largest-values-from-labels.py

```
# DESC
# Example 3:
# We have a set of items: the i-th item has value values[i] and label labels[i].
# Return the largest possible sum of the subset S.
# Example 2:
# Then, we choose a subset S of these items, such that:
# Example 1:
# Note:
# Example 4:

# NOTE
# For every label L, the number of items in S with label L is <= use_limit.
# 0 <= values[i], labels[i] <= 20000
# 1 <= values.length == labels.length <= 20000
# 1 <= num_wanted, use_limit <= values.length
# |S| <= num_wanted

# EXAMPLE
# 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.
# 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.
# 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.
# 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
# .

# 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)
        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

```
# 3sum-closest is not found.
# 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
```

paint-fence.py

```
# paint-fence is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numWays(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        if n == 0:
            return 0
        elif n == 1:
            return k
        ways = [0] * 3
        ways[0] = k
        ways[1] = (k - 1) * ways[0] + k
        for i in xrange(2, n):
            ways[i % 3] = (k - 1) * (ways[(i - 1) % 3] + ways[(i - 2) % 3])
        return ways[(n - 1) % 3]

# Time:  O(n)
# Space: O(n)
# DP solution.
class Solution2(object):
    def numWays(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        if n == 0:
            return 0
        elif n == 1:
            return k
        ways = [0] * n
        ways[0] = k
        ways[1] = (k - 1) * ways[0] + k
        for i in xrange(2, n):
            ways[i] = (k - 1) * (ways[i - 1] + ways[i - 2])
        return ways[n - 1]
```

word-break-ii.py

```
# DESC
# Given a non-empty string s and a dictionary wordDict containing a list of non-em
# pty words, add spaces in s to construct a sentence where each word is a valid di
# ctionary word. Return all such possible sentences.
# Note:
# Example 2:
# Example 1:
# Example 3:

# 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
# Input:
# s = "catsanddog"
# wordDict = ["cat", "cats", "and", "sand", "dog"]
# Output:
#
# [
#   "cats and dog",
#   "cat sand dog"
# ]
# Input:
# s = "pineapplepenapple"
# wordDict = ["apple", "pen", "applepen", "pine", "
# pineapple"]
# Output:
# [
#   "pine apple pen apple",
#   "pineapple pen apple",
#   "pine
#  applepen apple"
# ]
# Explanation: Note that you are allowed to reuse a dictionary
# word.
# Input:
# s = "catsandog"
# wordDict = ["cats", "dog", "sand", "and", "cat"]
# Output:
# []

# Time:  O(n * l^2 + n * r), l is the max length of the words,
#                            r is the number of the results.
# Space: O(n^2)

class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: Set[str]
        :rtype: List[str]
        """
        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)]
    valid = [[False] * n for _ in xrange(n)]
    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:
                valid[i-l][i-1] = True
                can_break[i] = True

    result = []
    if can_break[-1]:
        self.wordBreakHelper(s, valid, 0, [], result)
    return result

def wordBreakHelper(self, s, valid, start, path, result):
    if start == len(s):
        result.append(" ".join(path))
        return
    for i in xrange(start, len(s)):
        if valid[start][i]:
            path += [s[start:i+1]]
            self.wordBreakHelper(s, valid, i + 1, path, result)
            path.pop()
```

play-with-chips.py

```python
# play-with-chips is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minCostToMoveChips(self, chips):
        """
        :type chips: List[int]
        :rtype: int
        """
        count = [0]*2
        for p in chips:
            count[p%2] += 1
        return min(count)
```

split-linked-list-in-parts.py

```
# DESC
# Example 1:
# Note:
# Return a List of ListNode's representing the linked list parts that are formed.
# The length of each part should be as equal as possible: no two parts should have
#  a size differing by more than 1.  This may lead to some parts being null.
# Given a (singly) linked list with head node root, write a function to split the
# linked list into k consecutive linked list "parts".
# The parts should be in order of occurrence in the input list, and parts occurrin
# g earlier should always have a size greater than or equal parts occurring later.
# Example 2:

# NOTE
# Each value of a node in the input will be an integer in the range [0, 999].
# The length of root will be in the range [0, 1000].
# k will be an integer in the range [1, 50].

# EXAMPLE
# Input:
# root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3
# Output: [[1, 2, 3, 4], [5,
#  6, 7], [8, 9, 10]]
# Explanation:
# The input has been split into consecutive parts
#  with size difference at most 1, and earlier parts are a larger size than the la
# ter parts.
# Input:
# root = [1, 2, 3], k = 5
# Output: [[1],[2],[3],[],[]]
# Explanation:
# The inp
# ut and each element of the output are ListNodes, not arrays.
# For example, the in
# put root has root.val = 1, root.next.val = 2, \root.next.next.val = 3, and root.
# next.next.next = null.
# The first element output[0] has output[0].val = 1, output
# [0].next = null.
# The last element output[4] is null, but it's string representat
# ion as a ListNode is [].

# Time:  O(n + k)
# Space: O(1)

class Solution(object):
    def splitListToParts(self, root, k):
        """
        :type root: ListNode
        :type k: int
        :rtype: List[ListNode]
        """
        n = 0
        curr = root
        while curr:
            curr = curr.next
            n += 1
        width, remainder = divmod(n, k)

        result = []
```

```
curr = root
for i in xrange(k):
    head = curr
    for j in xrange(width-1+int(i < remainder)):
        if curr:
            curr = curr.next
    if curr:
        curr.next, curr = None, curr.next
    result.append(head)
return result
```

solve-the-equation.py

```python
# DESC
# If there are infinite solutions for the equation, return "Infinite solutions".
# 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 coeffic
# ient.
# Example 4:
# If there is no solution for the equation, return "No solution".
# Example 5:
# Example 1:
# If there is exactly one solution for the equation, we ensure that the value of x
#  is an integer.
# Example 3:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: "x=x+2"
# Output: "No solution"
# Input: "x=x"
# Output: "Infinite solutions"
# Input: "2x=x"
# Output: "x=0"
# Input: "x+5-3+x=6+x-2"
# Output: "x=2"
# Input: "2x+3x-6x=x+2"
# Output: "x=-1"

# 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'
```

number-of-distinct-islands.py

```
# number-of-distinct-islands is not found.
# Time:  O(m * n)
# Space: O(m * n)

class Solution(object):
    def numDistinctIslands(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = {'l':[-1,  0], 'r':[ 1,  0], \
                      'u':[ 0,  1], 'd':[ 0, -1]}

        def dfs(i, j, grid, island):
            if not (0 <= i < len(grid) and \
                    0 <= j < len(grid[0]) and \
                    grid[i][j] > 0):
                return False
            grid[i][j] *= -1
            for k, v in directions.iteritems():
                island.append(k)
                dfs(i+v[0], j+v[1], grid, island)
            return True

        islands = set()
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                island = []
                if dfs(i, j, grid, island):
                    islands.add("".join(island))
        return len(islands)
```

flip-game.py

```
# flip-game is not found.
# Time:  O(c * n + n) = O(n * (c+1))
# Space: O(n)

class Solution(object):
    def generatePossibleNextMoves(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        res = []
        i, n = 0, len(s) - 1
        while i < n:                                    # O(n) time
            if s[i] == '+':
                while i < n and s[i+1] == '+':          # O(c) time
                    res.append(s[:i] + '--' + s[i+2:])  # O(n) time and space
                    i += 1
            i += 1
        return res


# Time:  O(c * m * n + n) = O(c * n + n), where m = 2 in this question
# Space: O(n)
# This solution compares O(m) = O(2) times for two consecutive "+", where m is length of the pattern
class Solution2(object):
  def generatePossibleNextMoves(self, s):
      """
      :type s: str
      :rtype: List[str]
      """
      return [s[:i] + "--" + s[i+2:] for i in xrange(len(s) - 1) if s[i:i+2] == "++"]
```

minimum-height-trees.py

```
# DESC
# Format
#
# The graph contains n nodes which are labeled from 0 to n - 1. You will b
# e given the number n and a list of undirected edges (each edge is a pair of labe
# ls).
# Example 1 :
# 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 e
# dges.
# 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 g
# raph, write a function to find all the MHTs and return a list of their root labe
# ls.
# Example 2 :
# Note:

# NOTE
# The height of a rooted tree is the number of edges on the longest downward path
# between the root and a leaf.
# 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, an
# y connected graph without simple cycles is a tree."

# EXAMPLE
# Input: n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]
#
#      0  1  2
#
#    \ | /
#         3
#         |
#         4
#         |
#         5
#
# Output: [3, 4]
# Input: n = 4, edges = [[1, 0], [1, 2], [1, 3]]
#
#         0
#         |
#         1
#
#      / \
#       2   3
#
# Output: [1]

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

import collections


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

```
        :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)
```

number-of-days-between-two-dates.py

```python
# number-of-days-between-two-dates is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def __init__(self):
        def dayOfMonth(M):
            return (28 if (M == 2) else 31-(M-1)%7%2)

        self.__lookup = [0]*12
        for M in xrange(1, len(self.__lookup)):
            self.__lookup[M] += self.__lookup[M-1]+dayOfMonth(M)

    def daysBetweenDates(self, date1, date2):
        """
        :type date1: str
        :type date2: str
        :rtype: int
        """
        def num_days(date):
            Y, M, D = map(int, date.split("-"))
            leap = 1 if M > 2 and (((Y % 4 == 0) and (Y % 100 != 0)) or (Y % 400 == 0)) else 0
            return (Y-1)*365 + ((Y-1)//4 - (Y-1)//100 + (Y-1)//400) + self.__lookup[M-1]+D+leap

        return abs(num_days(date1) - num_days(date2))


# Time:  O(1)
# Space: O(1)
import datetime


class Solution2(object):
    def daysBetweenDates(self, date1, date2):
        delta = datetime.datetime.strptime(date1, "%Y-%m-%d")
        delta -= datetime.datetime.strptime(date2, "%Y-%m-%d")
        return abs(delta.days)
```

sort-integers-by-the-power-value.py

```python
# sort-integers-by-the-power-value is not found.
# Time:  O(n) on average
# Space: O(n)

import random


class Solution(object):
    dp = {}

    def getKth(self, lo, hi, k):
        """
        :type lo: int
        :type hi: int
        :type k: int
        :rtype: int
        """
        def nth_element(nums, n, compare=lambda a, b: a < b):
            def partition_around_pivot(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:
                pivot_idx = random.randint(left, right)
                new_pivot_idx = partition_around_pivot(left, right, pivot_idx, nums, compare)
                if new_pivot_idx == n:
                    return
                elif new_pivot_idx > n:
                    right = new_pivot_idx - 1
                else:  # new_pivot_idx < n
                    left = new_pivot_idx + 1

        def power_value(x):
            y, result = x, 0
            while x > 1 and x not in Solution.dp:
                result += 1
                if x%2:
                    x = 3*x + 1
                else:
                    x //= 2
            Solution.dp[y] = result + (Solution.dp[x] if x > 1 else 0)
            return Solution.dp[y], y

        arr = map(power_value, range(lo, hi+1))
        nth_element(arr, k-1)
        return arr[k-1][1]


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

```python
class Solution2(object):
    dp = {}

    def getKth(self, lo, hi, k):
        """
        :type lo: int
        :type hi: int
        :type k: int
        :rtype: int
        """
        def power_value(x):
            y, result = x, 0
            while x > 1 and x not in Solution2.dp:
                result += 1
                if x%2:
                    x = 3*x + 1
                else:
                    x //= 2
            Solution2.dp[y] = result + (Solution2.dp[x] if x > 1 else 0)
            return Solution2.dp[y], y

        return sorted(range(lo, hi+1), key=power_value)[k-1]
```

constrained-subset-sum.py

```
# constrained-subset-sum is not found.
# Time:  O(n)
# Space: O(k)

import collections


class Solution(object):
    def constrainedSubsetSum(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        result, dq = float("-inf"), collections.deque()
        for i in xrange(len(nums)):
            if dq and i-dq[0][0] == k+1:
                dq.popleft()
            curr = nums[i] + (dq[0][1] if dq else 0)
            while dq and dq[-1][1] <= curr:
                dq.pop()
            if curr > 0:
                dq.append((i, curr))
            result = max(result, curr)
        return result
```

custom-sort-string.py

```
# DESC
# Return any permutation of T (as a string) that satisfies this property.
# S was sorted in some custom order previously. We want to permute the characters
# of T so that they match the order that S was sorted. More specifically, if x occ
# urs before y in S, then x should occur before y in the returned string.
# Note:
# S and T are strings composed of lowercase letters. In S, no letter occurs more t
# han once.

# NOTE
# S has length at most 26, and no character is repeated in S.
# T has length at most 200.
# S and T consist of lowercase letters only.

# EXAMPLE
# Example :
# Input:
# S = "cba"
# T = "abcd"
# Output: "cbad"
# Explanation:
# "a", "b", "c
# " appear in S, so the order of "a", "b", "c" should be "c", "b", and "a".
# Since
#  "d" does not appear in S, it can be at any position in T. "dcba", "cdba", "cbda
# " are also valid outputs.

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def customSortString(self, S, T):
        """
        :type S: str
        :type T: str
        :rtype: str
        """
        counter, s = collections.Counter(T), set(S)
        result = [c*counter[c] for c in S]
        result.extend([c*counter for c, counter in counter.iteritems() if c not in s])
        return "".join(result)
```

binary-search.py

```
# DESC
# Example 1:
# Given a sorted (in ascending order) integer array nums of n elements and a targe
# t value, write a function to search target in nums. If target exists, then retur
# n its index, otherwise return -1.
# Note:
# Example 2:

# NOTE
# The value of each element in nums will be in the range [-9999, 9999].
# You may assume that all elements in nums are unique.
# n will be in the range [1, 10000].

# EXAMPLE
# Input: nums = [-1,0,3,5,9,12], target = 2
# Output: -1
# Explanation: 2 does not exi
# st in nums so return -1
# Input: nums = [-1,0,3,5,9,12], target = 9
# Output: 4
# Explanation: 9 exists in num
# s and its index is 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:
                right = mid-1
            elif nums[mid] < target:
                left = mid+1
            else:
                return mid
        return -1
```

continuous-subarray-sum.py

```
# DESC
# Example 1:
# Example 2:
# 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.
# Constraints:

# NOTE
# You may assume the sum of all the numbers is in the range of a signed 32-bit integer.
# The length of the array won't exceed 10,000.

# EXAMPLE
# 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.
# Input: [23, 2, 4, 6, 7],  k=6
# Output: True
# Explanation: Because [2, 4] is a cont
# inuous subarray of size 2 and sums up to 6.

# 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
```

running-sum-of-1d-array.py

```python
# running-sum-of-1d-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def runningSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        for i in xrange(len(nums)-1):
            nums[i+1] += nums[i]
        return nums
```

sort-integers-by-the-number-of-1-bits.py

```python
# sort-integers-by-the-number-of-1-bits is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def sortByBits(self, arr):
        """
        :type arr: List[int]
        :rtype: List[int]
        """
        def popcount(n):  # Time: O(logn) ~= O(1) if n is a 32-bit number
            result = 0
            while n:
                n &= n - 1
                result += 1
            return result

        arr.sort(key=lambda x: (popcount(x), x))
        return arr
```

brick-wall.py

```
# DESC
# 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:
# Note:
# 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 an
# d return the number of crossed bricks.
# There is a brick wall in front of you. The wall is rectangular and has several r
# ows 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.

# 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 i
# n range [1,10,000]. Total number of bricks of the wall won't exceed 20,000.

# EXAMPLE
# Input: [[1,2,2,1],
#         [3,1,2],
#         [1,3,2],
#         [2,4],
#         [3,1
# ,2],
#         [1,3,1,1]]
#
# Output: 2
#
# Explanation:

# 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

```
# DESC
# Example 2:
# Example 1:
# 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.
# The gray code is a binary numeral system where two successive values differ in o
# nly one bit.

# NOTE
#

# EXAMPLE
# 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 si
# ze is 20 = 1.
#              Therefore, for n = 0 the gray code sequence is [0].
# 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

# 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)]
```

unique-word-abbreviation.py

```
# unique-word-abbreviation is not found.
# Time:  ctor:   O(n), n is number of words in the dictionary.
#        lookup: O(1)
# Space: O(k), k is number of unique words.

import collections


class ValidWordAbbr(object):
    def __init__(self, dictionary):
        """
        initialize your data structure here.
        :type dictionary: List[str]
        """
        self.lookup_ = collections.defaultdict(set)
        for word in dictionary:
            abbr = self.abbreviation(word)
            self.lookup_[abbr].add(word)


    def isUnique(self, word):
        """
        check if a word is unique.
        :type word: str
        :rtype: bool
        """
        abbr = self.abbreviation(word)
        return self.lookup_[abbr] <= {word}


    def abbreviation(self, word):
        if len(word) <= 2:
            return word
        return word[0] + str(len(word)-2) + word[-1]
```

sort-transformed-array.py

```python
# sort-transformed-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def sortTransformedArray(self, nums, a, b, c):
        """
        :type nums: List[int]
        :type a: int
        :type b: int
        :type c: int
        :rtype: List[int]
        """
        f = lambda x, a, b, c : a * x * x + b * x + c

        result = []
        if not nums:
            return result

        left, right = 0, len(nums) - 1
        d = -1 if a > 0 else 1
        while left <= right:
            if d * f(nums[left], a, b, c) < d * f(nums[right], a, b, c):
                result.append(f(nums[left], a, b, c))
                left += 1
            else:
                result.append(f(nums[right], a, b, c))
                right -= 1

        return result[::d]
```

odd-even-linked-list.py

```
# DESC
# You should try to do it in place. The program should run in O(1) space complexit
# y and O(nodes) time complexity.
# Example 1:
# Given a singly linked list, group all odd nodes together followed by the even no
# des. Please note here we are talking about the node number and not the value in
# the nodes.
# Example 2:
# Constraints:

# NOTE
# The relative order inside both the even and odd groups should remain as it was i
# n 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].

# EXAMPLE
# Input: 1->2->3->4->5->NULL
# Output: 1->3->5->2->4->NULL
# Input: 2->1->3->5->6->4->7->NULL
# Output: 2->3->6->7->1->5->4->NULL

# 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
```

basic-calculator-iii.py

```
# basic-calculator-iii is not found.
# Time:  O(n)
# Space: O(n)


class Solution(object):
    def calculate(self, s):
        """
        :type s: str
        :rtype: int
        """
        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]

    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)
```

simplified-fractions.py

```
# simplified-fractions is not found.
# Time:  O(n^2 * logn)
# Space: O(n^2)

import fractions

class Solution(object):
    def simplifiedFractions(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        lookup = set()
        for b in xrange(1, n+1):
            for a in xrange(1, b):
                g = fractions.gcd(a, b)
                lookup.add((a//g, b//g))
        return map(lambda x: "{}/{}".format(*x), lookup)
```

kth-missing-positive-number.py

```python
# DESC
# Example 2:
# Given an array arr of positive integers sorted in a strictly increasing order, a
# nd an integer k.
# Example 1:
# Constraints:
# Find the kth positive integer that is missing from this array.

# NOTE
# 1 <= arr.length <= 1000
# 1 <= arr[i] <= 1000
# 1 <= k <= 1000
# arr[i] < arr[j] for 1 <= i < j <= arr.length

# EXAMPLE
# Input: arr = [2,3,4,7,11], k = 5
# Output: 9
# Explanation: The missing positive int
# egers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.
# Input: arr = [1,2,3,4], k = 2
# Output: 6
# Explanation: The missing positive intege
# rs are [5,6,7,...]. The 2nd missing positive integer is 6.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def findKthPositive(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: int
        """
        def check(arr, k, x):
            return arr[x]-(x+1) < k

        left, right = 0, len(arr)-1
        while left <= right:
            mid = left + (right-left)//2
            if not check(arr, k, mid):
                right = mid-1
            else:
                left = mid+1
        return right+1+k  # arr[right] + (k-(arr[right]-(right+1))) if right >= 0 else k
```

boundary-of-binary-tree.py

```python
# boundary-of-binary-tree is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def boundaryOfBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        def leftBoundary(root, nodes):
            if not root or (not root.left and not root.right):
                return
            nodes.append(root.val)
            if not root.left:
                leftBoundary(root.right, nodes)
            else:
                leftBoundary(root.left, nodes)

        def rightBoundary(root, nodes):
            if not root or (not root.left and not root.right):
                return
            if not root.right:
                rightBoundary(root.left, nodes)
            else:
                rightBoundary(root.right, nodes)
            nodes.append(root.val)

        def leaves(root, nodes):
            if not root:
                return
            if not root.left and not root.right:
                nodes.append(root.val)
                return
            leaves(root.left, nodes)
            leaves(root.right, nodes)

        if not root:
            return []

        nodes = [root.val]
        leftBoundary(root.left, nodes)
        leaves(root.left, nodes)
        leaves(root.right, nodes)
        rightBoundary(root.right, nodes)
        return nodes
```

find-the-derangement-of-an-array.py

```python
# find-the-derangement-of-an-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findDerangement(self, n):
        """
        :type n: int
        :rtype: int
        """
        M = 1000000007
        mul, total = 1, 0
        for i in reversed(xrange(n+1)):
            total = (total + M + (1 if i % 2 == 0 else -1) * mul) % M
            mul = (mul * i) % M
        return total
```

domino-and-tromino-tiling.py

```
# DESC
# (In a tiling, every square must be covered by a tile. Two tilings are different
# if and only if there are two 4-directionally adjacent cells on the board such th
# at exactly one of the tilings has both squares occupied by a tile.)
# Note:
# We have two types of tiles: a 2x1 domino shape, and an "L" tromino shape. These
# shapes may be rotated.
# Given N, how many ways are there to tile a 2 x N board? Return your answer modul
# o 10^9 + 7.

# NOTE
# N  will be in range [1, 1000].

# EXAMPLE
# Example:
# Input: 3
# Output: 5
# Explanation:
# The five different ways are listed bel
# ow, different letters indicates different tiles:
# XYZ XXZ XYY XXY XYY
# XYZ YYZ XZZ
#  XYY XXY
# XX  <- domino
#
# XX  <- "L" tromino
# X

# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def numTilings(self, N):
        """
        :type N: int
        :rtype: int
        """
        M = int(1e9+7)

        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]

        T = [[1, 0, 0, 1],  # #(|) = #(|) + #(=)
```

```
            [1, 0, 1, 0],  # #() = #(|) + #(L)
            [1, 1, 0, 0],  # #(L) = #(|) + #()
            [1, 1, 1, 0]]  # #(=) = #(|) + #() + #(L)

        return matrix_mult([[1, 0, 0, 0]], matrix_expo(T, N))[0][0] # [a0, a(-1), a(-2), a(-3)] * T^N


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def numTilings(self, N):
        """
        :type N: int
        :rtype: int
        """
        # Prove:
        # dp[n] = dp[n-1](|) + dp[n-2](=) + 2*(dp[n-3]() + ... + d[0]( = ... = ))
        #       = dp[n-1] + dp[n-2] + dp[n-3] + dp[n-3] + 2*(dp[n-4] + ... + d[0])
        #       = dp[n-1] + dp[n-3] + (dp[n-2] + dp[n-3] + 2*(dp[n-4] + ... + d[0])
        #       = dp[n-1] + dp[n-3] + dp[n-1]
        #       = 2*dp[n-1] + dp[n-3]
        M = int(1e9+7)
        dp = [1, 1, 2]
        for i in xrange(3, N+1):
            dp[i%3] = (2*dp[(i-1)%3]%M + dp[(i-3)%3])%M
        return dp[N%3]
```

longest-uncommon-subsequence-ii.py

```
# DESC
# 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 so
# me characters without changing the order of the remaining elements. Trivially, a
# ny string is a subsequence of itself and an empty string is a subsequence of any
#  string.
# Example 1:
# Note:
# The input will be a list of strings, and the output needs to be the length of th
# e longest uncommon subsequence. If the longest uncommon subsequence doesn't exis
# t, return -1.

# NOTE
# The length of the given list will be in the range of [2, 50].
# All the given strings' lengths will not exceed 10.

# EXAMPLE
# Input: "aba", "cdc", "eae"
# Output: 3

# Time:  O(l * n^2)
# Space: O(1)

class Solution(object):
    def findLUSlength(self, strs):
        """
        :type strs: List[str]
        :rtype: int
        """
        def isSubsequence(a, b):
            i = 0
            for j in xrange(len(b)):
                if i >= len(a):
                    break
                if a[i] == b[j]:
                    i += 1
            return i == len(a)

        strs.sort(key=len, reverse=True)
        for i in xrange(len(strs)):
            all_of = True
            for j in xrange(len(strs)):
                if len(strs[j]) < len(strs[i]):
                    break
                if i != j and isSubsequence(strs[i], strs[j]):
                    all_of = False
                    break
            if all_of:
                return len(strs[i])
        return -1
```

freedom-trail.py

```
# DESC
# Initially, the first character of the ring is aligned at 12:00 direction. You ne
# ed to spell all the characters in the string key one by one by rotating the ring
#  clockwise or anticlockwise to make each character of the string key aligned at
# 12:00 direction and then by pressing the center button.
# Given a string ring, which represents the code engraved on the outer ring and an
# other string key, which represents the keyword needs to be spelled. You need to
# find the minimum number of steps in order to spell all the characters in the key
# word.
# At the stage of rotating the ring to spell the key character key[i]:
# In the video game Fallout 4, the quest "Road to Freedom" requires players to rea
# ch a metal dial called the "Freedom Trail Ring", and use the dial to spell a spe
# cific keyword in order to open the door.
# Note:
# Example:

# NOTE
# Length of both ring and key will be in range 1 to 100.
# If the character key[i] has been aligned at the 12:00 direction, you need to pre
# ss the center button to spell, which also counts as 1 step. After the pressing,
# you could begin to spell the next character in the key (next stage), otherwise,
# you've finished all the spelling.
# There are only lowercase letters in both strings and might be some duplcate char
# acters in both strings.
# You can rotate the ring clockwise or anticlockwise one place, which counts as 1
# step. The final purpose of the rotation is to align one of the string ring's cha
# racters at the 12:00 direction, where this character must equal to the character
#  key[i].
# It's guaranteed that string key could always be spelled by rotating the string ring.

# EXAMPLE
# Input: ring = "godding", key = "gd"
# Output: 4
# Explanation:
# For the first key cha
# racter 'g', since it is already in place, we just need 1 step to spell this char
# acter.
# For the second key character 'd', we need to rotate the ring "godding" a
# nticlockwise by two steps to make it become "ddinggo".
# Also, we need 1 more step
#  for spelling.
# So the final output is 4.

# Time:  O(k) ~ O(k * r^2)
# Space: O(r)

import collections


class Solution(object):
    def findRotateSteps(self, ring, key):
        """
        :type ring: str
        :type key: str
        :rtype: int
        """
        lookup = collections.defaultdict(list)
        for i in xrange(len(ring)):
```

```
    lookup[ring[i]].append(i)

dp = [[0] * len(ring) for _ in xrange(2)]
prev = [0]
for i in xrange(1, len(key)+1):
    dp[i%2] = [float("inf")] * len(ring)
    for j in lookup[key[i-1]]:
        for k in prev:
            dp[i%2][j] = min(dp[i%2][j],
                             min((k+len(ring)-j) % len(ring), \
                                 (j+len(ring)-k) % len(ring)) + \
                             dp[(i-1) % 2][k])
    prev = lookup[key[i-1]]
return min(dp[len(key)%2]) + len(key)
```

champagne-tower.py

```
# DESC
# Note:
# Now after pouring some non-negative integer cups of champagne, return how full t
# he j-th glass in the i-th row is (both i and j are 0 indexed.)
# Then, some champagne is poured in the first glass at the top.  When the top most
#  glass is full, any excess liquid poured will fall equally to the glass immediat
# ely to the left and right of it.  When those glasses become full, any excess cha
# mpagne will fall equally to the left and right of those glasses, and so on.  (A
# glass at the bottom row has it's excess champagne fall on the floor.)
# For example, after one cup of champagne is poured, the top most glass is full.
# After two cups of champagne are poured, the two glasses on the second row are ha
# lf full.  After three cups of champagne are poured, those two cups become full -
#  there are 3 full glasses total now.  After four cups of champagne are poured, t
# he third row has the middle glass half full, and the two outside glasses are a q
# uarter full, as pictured below.
# We stack glasses in a pyramid, where the first row has 1 glass, the second row h
# as 2 glasses, and so on until the 100th row.  Each glass holds one cup (250ml) o
# f champagne.

# NOTE
# poured will be in the range of [0, 10 ^ 9].
# query_glass and query_row will be in the range of [0, 99].

# EXAMPLE
# Example 1:
# Input: poured = 1, query_glass = 1, query_row = 1
# Output: 0.0
# Explana
# tion: We poured 1 cup of champange to the top glass of the tower (which is index
# ed as (0, 0)). There will be no excess liquid so all the glasses under the top g
# lass will remain empty.
#
# Example 2:
# Input: poured = 2, query_glass = 1, query_ro
# w = 1
# Output: 0.5
# Explanation: We poured 2 cups of champange to the top glass of
#  the tower (which is indexed as (0, 0)). There is one cup of excess liquid. The
# glass indexed as (1, 0) and the glass indexed as (1, 1) will share the excess li
# quid equally, and each will get half cup of champange.

# Time:  O(n^2) = O(1), since n is at most 99
# Space: O(n) = O(1)

class Solution(object):
    def champagneTower(self, poured, query_row, query_glass):
        """
        :type poured: int
        :type query_row: int
        :type query_glass: int
        :rtype: float
        """
        result = [poured] + [0] * query_row
        for i in xrange(1, query_row+1):
            for j in reversed(xrange(i+1)):
                result[j] = max(result[j]-1, 0)/2.0 + \
                            max(result[j-1]-1, 0)/2.0
        return min(result[query_glass], 1)
```

number-of-squareful-arrays.py

```
# DESC
# Note:
# Example 2:
# Given an array A of non-negative integers, the array is squareful if for every p
# air of adjacent elements, their sum is a perfect square.
# Return the number of permutations of A that are squareful.  Two permutations A1
# and A2 differ if and only if there is some index i such that A1[i] != A2[i].
# Example 1:

# NOTE
# 0 <= A[i] <= 1e9
# 1 <= A.length <= 12

# EXAMPLE
# Input: [1,17,8]
# Output: 2
# Explanation:
# [1,8,17] and [17,8,1] are the valid perm
# utations.
# Input: [2,2,2]
# Output: 1

# Time:  O(n!)
# Space: O(n^2)

import collections


class Solution(object):
    def numSquarefulPerms(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        def dfs(candidate, x, left, count, result):
            count[x] -= 1
            if left == 0:
                result[0] += 1
            for y in candidate[x]:
                if count[y]:
                    dfs(candidate, y, left-1, count, result)
            count[x] += 1

        count = collections.Counter(A)
        candidate = {i: {j for j in count if int((i+j)**0.5) ** 2 == i+j}
                         for i in count}

        result = [0]
        for x in count:
            dfs(candidate, x, len(A)-1, count, result)
        return result[0]
```

reverse-vowels-of-a-string.py

```
# DESC
# Example 2:
# Example 1:
# Write a function that takes a string as input and reverse only the vowels of a s
# tring.
# Note:
#
# The vowels does not include the letter "y".

# NOTE
#

# EXAMPLE
# Input: "leetcode"
# Output: "leotcede"
# Input: "hello"
# Output: "holle"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reverseVowels(self, s):
        """
        :type s: str
        :rtype: str
        """
        vowels = "aeiou"
        string = list(s)
        i, j = 0, len(s) - 1
        while i < j:
            if string[i].lower() not in vowels:
                i += 1
            elif string[j].lower() not in vowels:
                j -= 1
            else:
                string[i], string[j] = string[j], string[i]
                i += 1
                j -= 1
        return "".join(string)
```

path-with-maximum-minimum-value.py

```
# path-with-maximum-minimum-value is not found.
# Time:  O(m * n * log(m * n))
# Space: O(m * n)

# binary search + dfs solution
class Solution(object):
    def maximumMinimumPath(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        def check(A, val, r, c, lookup):
            if r == len(A)-1 and c == len(A[0])-1:
                return True
            lookup.add((r, c))
            for d in directions:
                nr, nc = r + d[0], c + d[1]
                if 0 <= nr < len(A) and \
                   0 <= nc < len(A[0]) and \
                   (nr, nc) not in lookup and \
                   A[nr][nc] >= val and \
                   check(A, val, nr, nc, lookup):
                    return True
            return False

        vals, ceil = [], min(A[0][0], A[-1][-1])
        for i in xrange(len(A)):
            for j in xrange(len(A[0])):
                if A[i][j] <= ceil:
                    vals.append(A[i][j])
        vals = list(set(vals))
        vals.sort()
        left, right = 0, len(vals)-1
        while left <= right:
            mid = left + (right-left)//2
            if not check(A, vals[mid], 0, 0, set()):
                right = mid-1
            else:
                left = mid+1
        return vals[right]


# Time:  O(m * n * log(m * n))
# Space: O(m * n)
import heapq


# Dijkstra algorithm solution
class Solution2(object):
    def maximumMinimumPath(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        max_heap = [(-A[0][0], 0, 0)]
```

```
lookup = set([(0, 0)])
while max_heap:
    i, r, c = heapq.heappop(max_heap)
    if r == len(A)-1 and c == len(A[0])-1:
        return -i
    for d in directions:
        nr, nc = r+d[0], c+d[1]
        if 0 <= nr < len(A) and \
           0 <= nc < len(A[0]) and \
           (nr, nc) not in lookup:
            heapq.heappush(max_heap, (-min(-i, A[nr][nc]), nr, nc))
            lookup.add((nr, nc))
return -1
```

third-maximum-number.py

```
# DESC
# Example 1:
# Given a non-empty array of integers, return the third maximum number in this arr
# ay. If it does not exist, return the maximum number. The time complexity must be
#  in O(n).
# Example 3:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: [1, 2]
#
# Output: 2
#
# Explanation: The third maximum does not exist, so the
# maximum (2) is returned instead.
# Input: [2, 2, 3, 1]
#
# Output: 1
#
# Explanation: Note that the third maximum here me
# ans the third maximum distinct number.
# Both numbers with value 2 are both consid
# ered as second maximum.
# Input: [3, 2, 1]
#
# Output: 1
#
# Explanation: The third maximum is 1.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def thirdMax(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        count = 0
        top = [float("-inf")] * 3
        for num in nums:
            if num > top[0]:
                top[0], top[1], top[2] = num, top[0], top[1]
                count += 1
            elif num != top[0] and num > top[1]:
                top[1], top[2] = num, top[1]
                count += 1
            elif num != top[0] and num != top[1] and num >= top[2]:
                top[2] = num
                count += 1

        if count < 3:
            return top[0]

        return top[2]
```

making-file-names-unique.py

```python
# making-file-names-unique is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def getFolderNames(self, names):
        """
        :type names: List[str]
        :rtype: List[str]
        """
        count = collections.Counter()
        result, lookup = [], set()
        for name in names:
            while True:
                name_with_suffix = "{}({})".format(name, count[name]) if count[name] else name
                count[name] += 1
                if name_with_suffix not in lookup:
                    break
            result.append(name_with_suffix)
            lookup.add(name_with_suffix)
        return result
```

zuma-game.py

```
# DESC
# Example 2:
# Example 3:
# Example 1:
# Constraints:
# Think about Zuma Game. You have a row of balls on the table, colored red(R), yel
# low(Y), blue(B), green(G), and white(W). You also have several balls in your han
# d.
# Each time, you may choose a ball in your hand, and insert it into the row (inclu
# ding the leftmost place and rightmost place). Then, if there is a group of 3 or
# more balls in the same color touching, remove these balls. Keep doing this until
#  no more balls can be removed.
# Example 4:
# Find the minimal balls you have to insert to remove all the balls on the table.
# If you cannot remove all the balls, output -1.

# NOTE
# 1 <= board.length <= 16
# Both input strings will be non-empty and only contain characters 'R','Y','B','G','W'.
# 1 <= hand.length <= 5
# You may assume that the initial row of balls on the table won't have any 3 or mo
# re consecutive balls with the same color.

# EXAMPLE
# Input: board = "WRRBBW", hand = "RB"
# Output: -1
# Explanation: WRRBBW -> WRR[R]BBW
#  -> WBBW -> WBB[B]W -> WW
# Input: board = "G", hand = "GGGGG"
# Output: 2
# Explanation: G -> G[G] -> GG[G] -> empty
# Input: board = "WWRRBBWW", hand = "WRBRW"
# Output: 2
# Explanation: WWRRBBWW -> WWR
# R[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWWW -> empty
# Input: board = "RBYYBBRRB", hand = "YRBGB"
# Output: 3
# Explanation: RBYYBBRRB -> R
# BYY[Y]BBRRB -> RBBBRRB -> RRRB -> B -> B[B] -> BB[B] -> empty

# for this problem, it should either relax time limit or
# add a description that a ball can be only inserted beside a ball with same color

import collections


# Time:  O((b+h) * h!*(b+h-1)!/(b-1)!)
# Space: O((b+h) * h!*(b+h-1)!/(b-1)!)
# brute force solution
class Solution_TLE_BUT_CORRECT(object):
    def findMinStep(self, board, hand):
        """
        :type board: str
        :type hand: str
        :rtype: int
        """
        def shrink(s):  # Time: O(n), Space: O(n)
            stack = []
```

```
            start = 0
            for i in xrange(len(s)+1):
                if i == len(s) or s[i] != s[start]:
                    if stack and stack[-1][0] == s[start]:
                        stack[-1][1] += i - start
                        if stack[-1][1] >= 3:
                            stack.pop()
                    elif s and i - start < 3:
                        stack += [s[start], i - start],
                    start = i
            result = []
            for p in stack:
                result += [p[0]] * p[1]
            return result

        def findMinStepHelper(board, hand, lookup):
            if not board: return 0
            if not hand: return float("inf")
            if tuple(hand) in lookup[tuple(board)]: return lookup[tuple(board)][tuple(hand)]

            result = float("inf")
            for i in xrange(len(hand)):
                for j in xrange(len(board)+1):
                    next_board = shrink(board[0:j] + hand[i:i+1] + board[j:])
                    next_hand = hand[0:i] + hand[i+1:]
                    result = min(result, findMinStepHelper(next_board, next_hand, lookup) + 1)
            lookup[tuple(board)][tuple(hand)] = result
            return result

        lookup = collections.defaultdict(dict)
        board, hand = list(board), list(hand)
        result = findMinStepHelper(board, hand, lookup)
        return -1 if result == float("inf") else result


# Time:  O(b * b! * h!)
# Space: O(b * b! * h!)
# if a ball can be only inserted beside a ball with same color,
# we can do by this solution
class Solution_WRONG_GREEDY_BUT_ACCEPT(object):
    def findMinStep(self, board, hand):
        """
        :type board: str
        :type hand: str
        :rtype: int
        """
        def shrink(s):  # Time: O(n), Space: O(n)
            stack = []
            start = 0
            for i in xrange(len(s)+1):
                if i == len(s) or s[i] != s[start]:
                    if stack and stack[-1][0] == s[start]:
                        stack[-1][1] += i - start
                        if stack[-1][1] >= 3:
                            stack.pop()
                    elif s and i - start < 3:
                        stack += [s[start], i - start],
                    start = i
            result = []
            for p in stack:
```

```
        result += [p[0]] * p[1]
    return result

def find(board, c, j):
    for i in xrange(j, len(board)):
        if board[i] == c:
            return i
    return -1

def findMinStepHelper(board, hand, lookup):
    if not board: return 0
    if not hand: return float("inf")
    if tuple(hand) in lookup[tuple(board)]: return lookup[tuple(board)][tuple(hand)]

    result = float("inf")
    for i in xrange(len(hand)):
        j = 0
        while j < len(board):
            k = find(board, hand[i], j)
            if k == -1:
                break

            if k < len(board) - 1 and board[k] == board[k+1]:
                next_board = shrink(board[0:k] + board[k+2:])
                next_hand = hand[0:i] + hand[i+1:]
                result = min(result, findMinStepHelper(next_board, next_hand, lookup) + 1)
                k += 1
            elif i > 0 and hand[i] == hand[i-1]:
                next_board = shrink(board[0:k] + board[k+1:])
                next_hand = hand[0:i-1] + hand[i+1:]
                result = min(result, findMinStepHelper(next_board, next_hand, lookup) + 2)
            j = k+1

    lookup[tuple(board)][tuple(hand)] = result
    return result

lookup = collections.defaultdict(dict)
board, hand = list(board), list(hand)
hand.sort()
result = findMinStepHelper(board, hand, lookup)
return -1 if result == float("inf") else result
```

number-of-digit-one.py

```
# DESC
# Given an integer n, count the total number of digit 1 appearing in all non-negat
# ive integers less than or equal to n.
# Example:

# NOTE
#

# EXAMPLE
# Input: 13
# Output: 6
# Explanation: Digit 1 occurred in the following numbers: 1,
# 10, 11, 12, 13.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def countDigitOne(self, n):
        """
        :type n: int
        :rtype: int
        """
        pivot, result = 1, 0
        while n >= pivot:
            result += (n//(10*pivot))*pivot + \
                      min(pivot, max(n%(10*pivot) - pivot + 1, 0))
            pivot *= 10
        return result


# Time:  O(logn)
# Space: O(1)
class Solution2(object):
    # @param {integer} n
    # @return {integer}
    def countDigitOne(self, n):
        k = 1
        cnt, multiplier, left_part = 0, 1, n

        while left_part > 0:
            # split number into: left_part, curr, right_part
            curr = left_part % 10
            right_part = n % multiplier

            # count of (c000 ~ oooc000) = (ooo + (k < curr)? 1 : 0) * 1000
            cnt += (left_part / 10 + (k < curr)) * multiplier

            # if k == 0, oooc000 = (ooo - 1) * 1000
            if k == 0 and multiplier > 1:
                cnt -= multiplier

            # count of (oook000 ~ oookxxx): count += xxx + 1
            if curr == k:
                cnt += right_part + 1

            left_part /= 10
            multiplier *= 10
```

```
    return cnt
```

parallel-courses.py

```
# parallel-courses is not found.
# Time:  O(|V| + |E|)
# Space: O(|E|)

import collections


class Solution(object):
    def minimumSemesters(self, N, relations):
        """
        :type N: int
        :type relations: List[List[int]]
        :rtype: int
        """
        g = collections.defaultdict(list)
        in_degree = [0]*N
        for x, y in relations:
            g[x-1].append(y-1)
            in_degree[y-1] += 1
        q = collections.deque([(1, i) for i in xrange(N) if not in_degree[i]])

        result = 0
        count = N
        while q:
            level, u = q.popleft()
            count -= 1
            result = level
            for v in g[u]:
                in_degree[v] -= 1
                if not in_degree[v]:
                    q.append((level+1, v))
        return result if count == 0 else -1
```

hamming-distance.py

```
# DESC
# Example:
# Given two integers x and y, calculate the Hamming distance.
# The Hamming distance between two integers is the number of positions at which th
# e corresponding bits are different.
# Note:
#
# 0  x, y < 231.

# NOTE
#

# EXAMPLE
# Input: x = 1, y = 4
#
# Output: 2
#
# Explanation:
# 1   (0 0 0 1)
# 4   (0 1 0 0)
#
# ↑   ↑
#
# The above arrows point to positions where the corresponding bits are diff
# erent.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def hammingDistance(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        distance = 0
        z = x ^ y
        while z:
            distance += 1
            z &= z - 1
        return distance

    def hammingDistance2(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        return bin(x ^ y).count('1')
```

1-bit-and-2-bit-characters.py

```
# 1-bit-and-2-bit-characters is not found.
# Time:  O(n)
# Space: O(1)


class Solution(object):
    def isOneBitCharacter(self, bits):
        """
        :type bits: List[int]
        :rtype: bool
        """
        parity = 0
        for i in reversed(xrange(len(bits)-1)):
            if bits[i] == 0:
                break
            parity ^= bits[i]
        return parity == 0
```

reorganize-string.py

```
# DESC
# If possible, output any possible result.  If not possible, return the empty string.
# Given a string S, check if the letters can be rearranged so that two characters
# that are adjacent to each other are not the same.
# Note:
# Example 1:
# Example 2:

# NOTE
# S will consist of lowercase letters and have length in range [1, 500].

# EXAMPLE
# Input: S = "aaab"
# Output: ""
# Input: S = "aab"
# Output: "aba"

# Time:  O(nloga) = O(n), a is the size of alphabet
# Space: O(a) = O(1)

import collections
import heapq


class Solution(object):
    def reorganizeString(self, S):
        """
        :type S: str
        :rtype: str
        """
        counts = collections.Counter(S)
        if any(v > (len(S)+1)/2 for k, v in counts.iteritems()):
            return ""

        result = []
        max_heap = []
        for k, v in counts.iteritems():
            heapq.heappush(max_heap, (-v, k))
        while len(max_heap) > 1:
            count1, c1 = heapq.heappop(max_heap)
            count2, c2 = heapq.heappop(max_heap)
            if not result or c1 != result[-1]:
                result.extend([c1, c2])
                if count1+1: heapq.heappush(max_heap, (count1+1, c1))
                if count2+1: heapq.heappush(max_heap, (count2+1, c2))
        return "".join(result) + (max_heap[0][1] if max_heap else '')
```

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

```
# DESC
# For each node at position (X, Y), its left and right children respectively will
# be at positions (X-1, Y-1) and (X+1, Y-1).
# Return an list of non-empty reports in order of X coordinate.  Every report will
#  have a list of values of nodes.
# Given a binary tree, return the vertical order traversal of its nodes values.
# Example 1:
# Note:
# (X, Y)
# Running a vertical line from X = -infinity to X = +infinity, whenever the vertic
# al line touches some nodes, we report the values of the nodes in order from top
# to bottom (decreasing Y coordinates).
# Example 2:
# If two nodes have the same position, then the value of the node that is reported
#  first is the value that is smaller.

# NOTE
# Each node's value will be between 0 and 1000.
# The tree will have between 1 and 1000 nodes.

# EXAMPLE
# Input: [3,9,20,null,null,15,7]
# Output: [[9],[3,15],[20],[7]]
# Explanation:
# Witho
# ut loss of generality, we can assume the root node is at position (0, 0):
# Then,
# the node with value 9 occurs at position (-1, -1);
# The nodes with values 3 and 1
# 5 occur at positions (0, 0) and (0, -2);
# The node with value 20 occurs at positi
# on (1, -1);
# The node with value 7 occurs at position (2, -2).
# Input: [1,2,3,4,5,6,7]
# 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 g
# iven scheme.
# However, in the report "[1,5,6]", the node value of 5 comes first s
# ince 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
```

count-substrings-with-only-one-distinct-letter.py

```python
# count-substrings-with-only-one-distinct-letter is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countLetters(self, S):
        """
        :type S: str
        :rtype: int
        """
        result = len(S)
        left = 0
        for right in xrange(1, len(S)):
            if S[right] == S[left]:
                result += right-left
            else:
                left = right
        return result
```

basic-calculator-ii.py

```
# DESC
# Example 2:
# The expression string contains only non-negative integers, +, -, *, / operators
# and empty spaces  . The integer division should truncate toward zero.
# Example 1:
# Note:
# Implement a basic calculator to evaluate a simple expression string.
# Example 3:

# NOTE
# You may assume that the given expression is always valid.
# Do not use the eval built-in library function.

# EXAMPLE
# Input: " 3/2 "
# Output: 1
# Input: "3+2*2"
# Output: 7
# Input: " 3+5 / 2 "
# Output: 5

# 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]

    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)
```

number-of-islands.py

```
# DESC
# Example 2:
# Example 1:
# Given a 2d grid map of '1's (land) and '0's (water), count the number of islands
# . An island is surrounded by water and is formed by connecting adjacent lands ho
# rizontally or vertically. You may assume all four edges of the grid are all surr
# ounded by water.

# NOTE
#

# EXAMPLE
# Input: grid = [
#   ["1","1","0","0","0"],
#   ["1","1","0","0","0"],
#   ["0","0","1"
# ,"0","0"],
#   ["0","0","0","1","1"]
# ]
# Output: 3
# Input: grid = [
#   ["1","1","1","1","0"],
#   ["1","1","0","1","0"],
#   ["1","1","0"
# ,"0","0"],
#   ["0","0","0","0","0"]
# ]
# Output: 1

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

class Solution(object):
    # @param {boolean[][]} grid a boolean 2D matrix
    # @return {int} an integer
    def numIslands(self, grid):
        if not grid:
            return 0

        row = len(grid)
        col = len(grid[0])
        count = 0
        for i in xrange(row):
            for j in xrange(col):
                if grid[i][j] == '1':
                    self.dfs(grid, row, col, i, j)
                    count += 1
        return count

    def dfs(self, grid, row, col, x, y):
        if grid[x][y] == '0':
            return
        grid[x][y] = '0'

        if x != 0:
            self.dfs(grid, row, col, x - 1, y)
        if x != row - 1:
            self.dfs(grid, row, col, x + 1, y)
```

```
if y != 0:
    self.dfs(grid, row, col, x, y - 1)
if y != col - 1:
    self.dfs(grid, row, col, x, y + 1)
```

find-the-town-judge.py

```
# DESC
# Constraints:
# If the town judge exists and can be identified, return the label of the town jud
# ge.  Otherwise, return -1.
# In a town, there are N people labelled from 1 to N.  There is a rumor that one o
# f these people is secretly the town judge.
# Example 1:
# Example 2:
# Example 5:
# Example 4:
# If the town judge exists, then:
# You are given trust, an array of pairs trust[i] = [a, b] representing that the p
# erson labelled a trusts the person labelled b.
# Example 3:

# NOTE
# Everybody (except for the town judge) trusts the town judge.
# 1 <= N <= 1000
# trust[i][0] != trust[i][1]
# trust[i] are all different
# trust[i].length == 2
# 1 <= trust[i][0], trust[i][1] <= N
# The town judge trusts nobody.
# 0 <= trust.length <= 10^4
# There is exactly one person that satisfies properties 1 and 2.

# EXAMPLE
# Input: N = 2, trust = [[1,2]]
# Output: 2
# Input: N = 3, trust = [[1,3],[2,3],[3,1]]
# Output: -1
# Input: N = 3, trust = [[1,2],[2,3]]
# Output: -1
# Input: N = 3, trust = [[1,3],[2,3]]
# Output: 3
# Input: N = 4, trust = [[1,3],[1,4],[2,3],[2,4],[4,3]]
# Output: 3

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

class Solution(object):
    def findJudge(self, N, trust):
        """
        :type N: int
        :type trust: List[List[int]]
        :rtype: int
        """
        degrees = [0]*N
        for i, j in trust:
            degrees[i-1] -= 1
            degrees[j-1] += 1
        for i in xrange(len(degrees)):
            if degrees[i] == N-1:
                return i+1
        return -1
```

minimum-number-of-steps-to-make-two-strings-anagram.py

```
# minimum-number-of-steps-to-make-two-strings-anagram is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def minSteps(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: int
        """
        diff = collections.Counter(s) - collections.Counter(t)
        return sum(diff.itervalues())
```

word-pattern.py

```
# DESC
# Example 4:
# Example 2:
# Example 3:
# Example 1:
# Given a pattern and a string str, find if str follows the same pattern.
# pattern
# Here follow means a full match, such that there is a bijection between a letter
# in pattern and a non-empty word in str.
# Notes:
#
# You may assume pattern contains only lowercase letters, and str contains
#  lowercase letters that may be separated by a single space.

# NOTE
#

# EXAMPLE
# Input: pattern = "aaaa", str = "dog cat cat dog"
# Output: false
# Input: pattern = "abba", str = "dog cat cat dog"
# Output: true
# Input: pattern = "abba", str = "dog dog dog dog"
# Output: false
# Input:pattern = "abba", str = "dog cat cat fish"
# Output: false

# Time:  O(n)
# Space: O(c), c is unique count of pattern

from itertools import izip  # Generator version of zip.

class Solution(object):
    def wordPattern(self, pattern, str):
        """
        :type pattern: str
        :type str: str
        :rtype: bool
        """
        if len(pattern) != self.wordCount(str):
            return False

        w2p, p2w = {}, {}
        for p, w in izip(pattern, self.wordGenerator(str)):
            if w not in w2p and p not in p2w:
                # Build mapping. Space: O(c)
                w2p[w] = p
                p2w[p] = w
            elif w not in w2p or w2p[w] != p:
                # Contradict mapping.
                return False
        return True

    def wordCount(self, str):
        cnt = 1 if str else 0
        for c in str:
            if c == ' ':
                cnt += 1
```

```
        return cnt

    # Generate a word at a time without saving all the words.
    def wordGenerator(self, str):
        w = ""
        for c in str:
            if c == ' ':
                yield w
                w = ""
            else:
                w += c
        yield w


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def wordPattern(self, pattern, str):
        """
        :type pattern: str
        :type str: str
        :rtype: bool
        """
        words = str.split()  # Space: O(n)
        if len(pattern) != len(words):
            return False

        w2p, p2w = {}, {}
        for p, w in izip(pattern, words):
            if w not in w2p and p not in p2w:
                # Build mapping. Space: O(c)
                w2p[w] = p
                p2w[p] = w
            elif w not in w2p or w2p[w] != p:
                # Contradict mapping.
                return False
        return True
```

strobogrammatic-number.py

```python
# strobogrammatic-number is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    lookup = {'0':'0', '1':'1', '6':'9', '8':'8', '9':'6'}

    # @param {string} num
    # @return {boolean}
    def isStrobogrammatic(self, num):
        n = len(num)
        for i in xrange((n+1) / 2):
            if num[n-1-i] not in self.lookup or \
               num[i] != self.lookup[num[n-1-i]]:
                return False
        return True
```

bitwise-and-of-numbers-range.py

```
# DESC
# Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of
# all numbers in this range, inclusive.
# Example 2:
# Example 1:

# NOTE
#

# EXAMPLE
# Input: [5,7]
# Output: 4
# Input: [0,1]
# Output: 0

# Time:  O(1)
# Space: O(1)

class Solution(object):
    # @param m, an integer
    # @param n, an integer
    # @return an integer
    def rangeBitwiseAnd(self, m, n):
        while m < n:
            n &= n - 1
        return n


class Solution2(object):
    # @param m, an integer
    # @param n, an integer
    # @return an integer
    def rangeBitwiseAnd(self, m, n):
        i, diff = 0, n-m
        while diff:
            diff >>= 1
            i += 1
        return n & m >> i << i
```

connecting-cities-with-minimum-cost.py

```python
# connecting-cities-with-minimum-cost is not found.
# Time:  O(nlogn)
# Space: O(n)

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:
            return False
        self.set[min(x_root, y_root)] = max(x_root, y_root)
        self.count -= 1
        return True


class Solution(object):
    def minimumCost(self, N, connections):
        """
        :type N: int
        :type connections: List[List[int]]
        :rtype: int
        """
        connections.sort(key = lambda x: x[2])
        union_find = UnionFind(N)
        result = 0
        for u, v, val in connections:
            if union_find.union_set(u-1, v-1):
                result += val
        return result if union_find.count == 1 else -1
```

count-odd-numbers-in-an-interval-range.py

```python
# count-odd-numbers-in-an-interval-range is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countOdds(self, low, high):
        """
        :type low: int
        :type high: int
        :rtype: int
        """
        return (high+1)//2 - ((low-1)+1)//2
```

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

```
# DESC
# Follow up:
# Constraints:
# 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.
# Example 1:
# Given a binary tree
# Initially, all next pointers are set to NULL.

# NOTE
# The number of nodes in the given tree is less than 6000.
# You may only use constant extra space.
# -100 <= node.val <= 100
# Recursive approach is fine, you may assume implicit stack space does not count a
# s extra space for this problem.

# EXAMPLE
# struct Node {
#   int val;
#   Node *left;
#   Node *right;
#   Node *next;
# }
# 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 pointe
# r 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.

# 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
    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
```

reconstruct-a-2-row-binary-matrix.py

```
# reconstruct-a-2-row-binary-matrix is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reconstructMatrix(self, upper, lower, colsum):
        """
        :type upper: int
        :type lower: int
        :type colsum: List[int]
        :rtype: List[List[int]]
        """
        upper_matrix, lower_matrix = [0]*len(colsum), [0]*len(colsum)
        for i in xrange(len(colsum)):
            upper_matrix[i] = int(upper > 0 and colsum[i] != 0)
            lower_matrix[i] = colsum[i]-upper_matrix[i]
            upper -= upper_matrix[i]
            lower -= lower_matrix[i]
        return [upper_matrix, lower_matrix] if upper == lower == 0 else []
```

maximum-area-of-a-piece-of-cake-after-horizontal-and-vertical-cuts.py

```python
# maximum-area-of-a-piece-of-cake-after-horizontal-and-vertical-cuts is not found.
# Time:  O(hlogh + wlogw)
# Space: O(1)

class Solution(object):
    def maxArea(self, h, w, horizontalCuts, verticalCuts):
        """
        :type h: int
        :type w: int
        :type horizontalCuts: List[int]
        :type verticalCuts: List[int]
        :rtype: int
        """
        def max_len(l, cuts):
            cuts.sort()
            l = max(cuts[0]-0, l-cuts[-1])
            for i in xrange(1, len(cuts)):
                l = max(l, cuts[i]-cuts[i-1])
            return l

        MOD = 10**9+7
        return max_len(h, horizontalCuts) * max_len(w, verticalCuts) % MOD
```

subarrays-with-k-different-integers.py

```
# DESC
# Example 1:
# Given an array A of positive integers, call a (contiguous, not necessarily disti
# nct) subarray of A good if the number of different integers in that subarray is
# exactly K.
# (For example, [1,2,3,1,2] has 3 different integers: 1, 2, and 3.)
# Example 2:
# [1,2,3,1,2]
# Return the number of good subarrays of A.
# Note:

# NOTE
# 1 <= A[i] <= A.length
# 1 <= K <= A.length
# 1 <= A.length <= 20000

# EXAMPLE
# Input: A = [1,2,1,3,4], K = 3
# Output: 3
# Explanation: Subarrays formed with exact
# ly 3 different integers: [1,2,1,3], [2,1,3], [1,3,4].
# Input: A = [1,2,1,2,3], K = 2
# Output: 7
# Explanation: Subarrays formed with exact
# ly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]
# .

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

import collections


class Solution(object):
    def subarraysWithKDistinct(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        def atMostK(A, K):
            count = collections.defaultdict(int)
            result, left = 0, 0
            for right in xrange(len(A)):
                count[A[right]] += 1
                while len(count) > K:
                    count[A[left]] -= 1
                    if count[A[left]] == 0:
                        count.pop(A[left])
                    left += 1
                result += right-left+1
            return result

        return atMostK(A, K) - atMostK(A, K-1)


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

```
class Window(object):
    def __init__(self):
        self.__count = collections.defaultdict(int)

    def add(self, x):
        self.__count[x] += 1

    def remove(self, x):
        self.__count[x] -= 1
        if self.__count[x] == 0:
            self.__count.pop(x)

    def size(self):
        return len(self.__count)


class Solution2(object):
    def subarraysWithKDistinct(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        window1, window2 = Window(), Window()
        result, left1, left2 = 0, 0, 0
        for i in A:
            window1.add(i)
            while window1.size() > K:
                window1.remove(A[left1])
                left1 += 1
            window2.add(i)
            while window2.size() >= K:
                window2.remove(A[left2])
                left2 += 1
            result += left2-left1
        return result
```

sum-of-subarray-minimums.py

```
# DESC
# Given an array of integers A, find the sum of min(B), where B ranges over every
# (contiguous) subarray of A.
# Example 1:
# Since the answer may be large, return the answer modulo 10^9 + 7.
# Note:
# 10^9 + 7

# NOTE
# 1 <= A.length <= 30000
# 1 <= A[i] <= 30000

# EXAMPLE
# Input: [3,1,2,4]
# Output: 17
# Explanation: Subarrays are [3], [1], [2], [4], [3,1]
# , [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
# Minimums are 3, 1, 2, 4, 1, 1, 2,
# 1, 1, 1.  Sum is 17.

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

import itertools


# Ascending stack solution
class Solution(object):
    def sumSubarrayMins(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        M = 10**9 + 7

        left, s1 = [0]*len(A), []
        for i in xrange(len(A)):
            count = 1
            while s1 and s1[-1][0] > A[i]:
                count += s1.pop()[1]
            left[i] = count
            s1.append([A[i], count])

        right, s2 = [0]*len(A), []
        for i in reversed(xrange(len(A))):
            count = 1
            while s2 and s2[-1][0] >= A[i]:
                count += s2.pop()[1]
            right[i] = count
            s2.append([A[i], count])

        return sum(a*l*r for a, l, r in itertools.izip(A, left, right)) % M
```

k-th-smallest-in-lexicographical-order.py

```
# DESC
# Example:
# Given integers n and k, find the lexicographically k-th smallest integer in the
# range from 1 to n.
# Note: 1  k  n  109.

# NOTE
#

# EXAMPLE
# Input:
# n: 13   k: 2
#
# Output:
# 10
#
# Explanation:
# The lexicographical order is [1, 1
# 0, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9], so the second smallest number is 10.

# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def findKthNumber(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        result = 0

        cnts = [0] * 10
        for i in xrange(1, 10):
            cnts[i] = cnts[i - 1] * 10 + 1

        nums = []
        i = n
        while i:
            nums.append(i % 10)
            i /= 10

        total, target = n, 0
        i = len(nums) - 1
        while i >= 0 and k > 0:
            target = target*10 + nums[i]
            start = int(i == len(nums)-1)
            for j in xrange(start, 10):
                candidate = result*10 + j
                if candidate < target:
                    num = cnts[i+1]
                elif candidate > target:
                    num = cnts[i]
                else:
                    num = total - cnts[i + 1]*(j-start) - cnts[i]*(9-j)
                if k > num:
                    k -= num
                else:
```

```
                    result = candidate
                    k -= 1
                    total = num-1
                    break
            i -= 1

        return result


# Time:  O(logn * logn)
# Space: O(logn)
class Solution2(object):
    def findKthNumber(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """
        def count(n, prefix):
            result, number = 0, 1
            while prefix <= n:
                result += number
                prefix *= 10
                number *= 10
            result -= max(number/10 - (n - prefix/10 + 1), 0)
            return result

        def findKthNumberHelper(n, k, cur, index):
            if cur:
                index += 1
                if index == k:
                    return (cur, index)

            i = int(cur == 0)
            while i <= 9:
                cur = cur * 10 + i
                cnt = count(n, cur)
                if k > cnt + index:
                    index += cnt
                elif cur <= n:
                    result = findKthNumberHelper(n, k, cur, index)
                    if result[0]:
                        return result
                i += 1
                cur /= 10
            return (0, index)

        return findKthNumberHelper(n, k, 0, 0)[0]
```

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

```python
# DESC
# Given two binary search trees root1 and root2.
# Example 4:
# Constraints:
# Return a list containing all the integers from both trees sorted in ascending order.
# Example 5:
# Example 2:
# Example 3:
# Example 1:

# NOTE
# Each node's value is between [-10^5, 10^5].
# Each tree has at most 5000 nodes.

# EXAMPLE
# Input: root1 = [0,-10,10], root2 = [5,1,7,0,2]
# Output: [-10,0,0,1,2,5,7,10]
# Input: root1 = [1,null,8], root2 = [8,1]
# Output: [1,1,8,8]
# Input: root1 = [], root2 = [5,1,7,0,2]
# Output: [0,1,2,5,7]
# Input: root1 = [0,-10,10], root2 = []
# Output: [-10,0,10]
# Input: root1 = [2,1,4], root2 = [1,0,3]
# Output: [0,1,1,2,3,4]

# 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 getAllElements(self, root1, root2):
        """
        :type root1: TreeNode
        :type root2: TreeNode
        :rtype: List[int]
        """
        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
```

n-queens-ii.py

```
# DESC
# Example:
# The n-queens puzzle is the problem of placing n queens on an n×n chessboard such
#  that no two queens attack each other.
# Given an integer n, return the number of distinct solutions to the n-queens puzzle.

# NOTE
#

# EXAMPLE
# Input: 4
# Output: 2
# Explanation: There are two distinct solutions to the 4-queens
#  puzzle as shown below.
# [
#  [".Q..",  // Solution 1
#   "...Q",
#   "Q...",
#   "..Q."]
# ,
#
#  ["..Q.",  // Solution 2
#   "Q...",
#   "...Q",
#   ".Q.."]
# ]

from functools import reduce
# Time:  O(n!)
# Space: O(n)

class Solution(object):
    # @return an integer
    def totalNQueens(self, n):
        self.cols = [False] * n
        self.main_diag = [False] * (2 * n)
        self.anti_diag = [False] * (2 * n)
        return self.totalNQueensRecu([], 0, n)

    def totalNQueensRecu(self, solution, row, n):
        if row == n:
            return 1
        result = 0
        for i in xrange(n):
            if not self.cols[i] and not self.main_diag[row + i] and not self.anti_diag[row - i + n]:
                self.cols[i] = self.main_diag[row + i] = self.anti_diag[row - i + n] = True
                result += self.totalNQueensRecu(solution + [i], row + 1, n)
                self.cols[i] = self.main_diag[row + i] = self.anti_diag[row - i + n] = False
        return result

# slower solution
class Solution2(object):
    # @return an integer
    def totalNQueens(self, n):
        return self.totalNQueensRecu([], 0, n)

    def totalNQueensRecu(self, solution, row, n):
        if row == n:
```

```
    return 1
result = 0
for i in xrange(n):
    if i not in solution and reduce(lambda acc, j: abs(row - j) != abs(i - solution[j]) and acc, xrang
        result += self.totalNQueensRecu(solution + [i], row + 1, n)
return result
```

number-of-students-doing-homework-at-a-given-time.py

```python
# number-of-students-doing-homework-at-a-given-time is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def busyStudent(self, startTime, endTime, queryTime):
        """
        :type startTime: List[int]
        :type endTime: List[int]
        :type queryTime: int
        :rtype: int
        """
        return sum(s <= queryTime <= e for s, e in itertools.izip(startTime, endTime))
```

prime-arrangements.py

```python
# prime-arrangements is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def numPrimeArrangements(self, n):
        """
        :type n: int
        :rtype: int
        """
        def count_primes(n):
            if n <= 1:
                return 0
            is_prime = [True]*((n+1)//2)
            cnt = len(is_prime)
            for i in xrange(3, n+1, 2):
                if i*i > n:
                    break
                if not is_prime[i//2]:
                    continue
                for j in xrange(i*i, n+1, 2*i):
                    if not is_prime[j//2]:
                        continue
                    cnt -= 1
                    is_prime[j//2] = False
            return cnt

        def factorial(n):
            result = 1
            for i in xrange(2, n+1):
                result = (result*i)%MOD
            return result

        MOD = 10**9+7
        cnt = count_primes(n)
        return factorial(cnt) * factorial(n-cnt) % MOD
```

find-median-from-data-stream.py

```
# DESC
# Median is the middle value in an ordered integer list. If the size of the list i
# s even, there is no middle value. So the median is the mean of the two middle va
# lue.
# [2,3,4], the median is 3
# Design a data structure that supports the following two operations:
# Follow up:
# [2,3,4]
# [2,3], the median is (2 + 3) / 2 = 2.5
# Example:

# NOTE
# If 99% of all integer numbers from the stream are between 0 and 100, how would y
# ou optimize it?
# double findMedian() - Return the median of all elements so far.
# void addNum(int num) - Add a integer number from the data stream to the data str
# ucture.
# If all integer numbers from the stream are between 0 and 100, how would you opti
# mize it?

# EXAMPLE
# addNum(1)
# addNum(2)
# findMedian() -> 1.5
# addNum(3)
# findMedian() -> 2

# Time:  O(nlogn) for total n addNums, O(logn) per addNum, O(1) per findMedian.
# Space: O(n), total space

from heapq import heappush, heappop

class MedianFinder(object):
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__max_heap = []
        self.__min_heap = []

    def addNum(self, num):
        """
        Adds a num into the data structure.
        :type num: int
        :rtype: void
        """
        # Balance smaller half and larger half.
        if not self.__max_heap or num > -self.__max_heap[0]:
            heappush(self.__min_heap, num)
            if len(self.__min_heap) > len(self.__max_heap) + 1:
                heappush(self.__max_heap, -heappop(self.__min_heap))
        else:
            heappush(self.__max_heap, -num)
            if len(self.__max_heap) > len(self.__min_heap):
                heappush(self.__min_heap, -heappop(self.__max_heap))

    def findMedian(self):
        """
```

```
        Returns the median of current data stream
        :rtype: float
        """
        return (-self.__max_heap[0] + self.__min_heap[0]) / 2.0 \
               if len(self.__min_heap) == len(self.__max_heap) \
               else self.__min_heap[0]
```

minimum-unique-word-abbreviation.py

```python
# minimum-unique-word-abbreviation is not found.
# Time:  O(2^n)
# Space: O(n)

class Solution(object):
    def minAbbreviation(self, target, dictionary):
        """
        :type target: str
        :type dictionary: List[str]
        :rtype: str
        """
        def bits_len(target, bits):
            return sum(((bits >> i) & 3) == 0 for i in xrange(len(target)-1))

        diffs = []
        for word in dictionary:
            if len(word) != len(target):
                continue
            diffs.append(sum(2**i for i, c in enumerate(word) if target[i] != c))

        if not diffs:
            return str(len(target))

        bits = 2**len(target) - 1
        for i in xrange(2**len(target)):
            if all(d & i for d in diffs) and bits_len(target, i) > bits_len(target, bits):
                bits = i

        abbr = []
        pre = 0
        for i in xrange(len(target)):
            if bits & 1:
                if i - pre > 0:
                    abbr.append(str(i - pre))
                pre = i + 1
                abbr.append(str(target[i]))
            elif i == len(target) - 1:
                abbr.append(str(i - pre + 1))
            bits >>= 1

        return "".join(abbr)
```

remove-duplicate-letters.py

```python
# DESC
# Given a string which contains only lowercase letters, remove duplicate letters s
# o that every letter appears once and only once. You must make sure your result i
# s the smallest in lexicographical order among all possible results.
# Example 2:
# Example 1:
# Note: This question is the same as 1081: https://leetcode.com/problems/smallest-
# subsequence-of-distinct-characters/

# NOTE
#

# EXAMPLE
# Input: "cbacdcbc"
# Output: "acdb"
# Input: "bcabc"
# Output: "abc"

# Time:  O(n)
# Space: O(k), k is size of the alphabet

from collections import Counter


class Solution(object):
    def removeDuplicateLetters(self, s):
        """
        :type s: str
        :rtype: str
        """
        remaining = Counter(s)

        in_stack, stk = set(), []
        for c in s:
            if c not in in_stack:
                while stk and stk[-1] > c and remaining[stk[-1]]:
                    in_stack.remove(stk.pop())
                stk += c
                in_stack.add(c)
            remaining[c] -= 1
        return "".join(stk)
```

heaters.py

```
# DESC
# So, your input will be the positions of houses and heaters seperately, and your
# expected output will be the minimum radius standard of heaters.
# Note:
# Example 2:
# Winter is coming! Your first job during the contest is to design a standard heat
# er with fixed warm radius to warm all the houses.
# Example 1:
# Now, you are given positions of houses and heaters on a horizontal line, find ou
# t minimum radius of heaters so that all houses could be covered by those heaters
# .

# NOTE
# Numbers of houses and heaters you are given are non-negative and will not exceed
#  25000.
# As long as a house is in the heaters' warm radius range, it can be warmed.
# Positions of houses and heaters you are given are non-negative and will not exce
# ed 10^9.
# All the heaters follow your radius standard and the warm radius will the same.

# EXAMPLE
# Input: [1,2,3],[2]
# Output: 1
# Explanation: The only heater was placed in the posi
# tion 2, and if we use the radius 1 standard, then all the houses can be warmed.
# Input: [1,2,3,4],[1,4]
# Output: 1
# Explanation: The two heater was placed in the p
# osition 1 and 4. We need to use radius 1 standard, then all the houses can be wa
# rmed.

# Time:  O((m + n) * logn), m is the number of the houses, n is the number of the heaters.
# Space: O(1)

import bisect


class Solution(object):
    def findRadius(self, houses, heaters):
        """
        :type houses: List[int]
        :type heaters: List[int]
        :rtype: int
        """
        heaters.sort()
        min_radius = 0
        for house in houses:
                equal_or_larger = bisect.bisect_left(heaters, house)
                curr_radius = float("inf")
                if equal_or_larger != len(heaters):
                    curr_radius = heaters[equal_or_larger] - house
                if equal_or_larger != 0:
                    smaller = equal_or_larger-1
                    curr_radius = min(curr_radius, house - heaters[smaller])
                min_radius = max(min_radius, curr_radius)
        return min_radius
```

search-a-2d-matrix.py

```
# DESC
# Example 2:
# Example 1:
# Write an efficient algorithm that searches for a value in an m x n matrix. This
# matrix has the following properties:

# NOTE
# 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
# Input:
# matrix = [
#   [1,   3,  5,  7],
#   [10, 11, 16, 20],
#   [23, 30, 34, 50]
# ]
# t
# arget = 3
# Output: true
# Input:
# matrix = [
#   [1,   3,  5,  7],
#   [10, 11, 16, 20],
#   [23, 30, 34, 50]
# ]
# t
# arget = 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
```

transform-to-chessboard.py

```
# DESC
# Note:
# What is the minimum number of moves to transform the board into a "chessboard" -
#  a board where no 0s and no 1s are 4-directionally adjacent? If the task is impo
# ssible, return -1.
# An N x N board contains only 0s and 1s. In each move, you can swap any 2 rows wi
# th each other, or any 2 columns with each other.

# NOTE
# board will have the same number of rows and columns, a number in the range [2, 30].
# board[i][j] will be only 0s or 1s.

# EXAMPLE
# Examples:
# Input: board = [[0,1,1,0],[0,1,1,0],[1,0,0,1],[1,0,0,1]]
# Output: 2
# Exp
# lanation:
# One potential sequence of moves is shown below, from left to right:
#
# 0
# 110     1010     1010
# 0110 --> 1010 --> 0101
# 1001     0101     1010
# 1001     010
# 1     0101
#
# The first move swaps the first and second column.
# The second move sw
# aps the second and third row.
#
#
# Input: board = [[0, 1], [1, 0]]
# Output: 0
# Explan
# ation:
# Also note that the board with 0 in the top left corner,
# 01
# 10
#
# is also a
# valid chessboard.
#
# Input: board = [[1, 0], [1, 0]]
# Output: -1
# Explanation:
# No ma
# tter what sequence of moves you make, you cannot end with a valid chessboard.

# Time:  O(n^2)
# Space: O(n^2), used by Counter, this could be reduced to O(n) by skipping invalid input

import collections
import itertools


class Solution(object):
    def movesToChessboard(self, board):
```

```
    """
    :type board: List[List[int]]
    :rtype: int
    """
    N = len(board)
    result = 0
    for count in (collections.Counter(map(tuple, board)), \
                  collections.Counter(itertools.izip(*board))):
        if len(count) != 2 or \
           sorted(count.values()) != [N/2, (N+1)/2]:
            return -1

        seq1, seq2 = count
        if any(x == y for x, y in itertools.izip(seq1, seq2)):
            return -1
        begins = [int(seq1.count(1) * 2 > N)] if N%2 else [0, 1]
        result += min(sum(int(i%2 != v) for i, v in enumerate(seq1, begin)) \
                      for begin in begins) / 2
    return result
```

integer-to-english-words.py

```
# DESC
# Convert a non-negative integer to its english words representation. Given input
# is guaranteed to be less than 231 - 1.
# Example 2:
# Example 1:
# Example 3:
# Example 4:

# NOTE
#

# EXAMPLE
# Input: 1234567891
# Output: "One Billion Two Hundred Thirty Four Million Five Hund
# red Sixty Seven Thousand Eight Hundred Ninety One"
# Input: 12345
# Output: "Twelve Thousand Three Hundred Forty Five"
# Input: 1234567
# Output: "One Million Two Hundred Thirty Four Thousand Five Hundre
# d Sixty Seven"
# Input: 123
# Output: "One Hundred Twenty Three"

# Time:  O(logn) = O(1), n is the value of the integer, which is less than 2^31 - 1
# Space: O(1)

class Solution(object):
    def numberToWords(self, num):
        """
        :type num: int
        :rtype: str
        """
        if num == 0:
            return "Zero"

        lookup = {0: "Zero", 1:"One", 2: "Two", 3: "Three", 4: "Four", \
                  5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine", \
                  10: "Ten", 11: "Eleven", 12: "Twelve", 13: "Thirteen", 14: "Fourteen", \
                  15: "Fifteen", 16: "Sixteen", 17: "Seventeen", 18: "Eighteen", 19: "Nineteen", \
                  20: "Twenty", 30: "Thirty", 40: "Forty", 50: "Fifty", 60: "Sixty", \
                  70: "Seventy", 80: "Eighty", 90: "Ninety"}
        unit = ["", "Thousand", "Million", "Billion"]

        res, i = [], 0
        while num:
            cur = num % 1000
            if num % 1000:
                res.append(self.threeDigits(cur, lookup, unit[i]))
            num //= 1000
            i += 1
        return " ".join(res[::-1])

    def threeDigits(self, num, lookup, unit):
        res = []
        if num / 100:
            res = [lookup[num / 100] + " " + "Hundred"]
        if num % 100:
            res.append(self.twoDigits(num % 100, lookup))
```

```
        if unit != "":
            res.append(unit)
        return " ".join(res)

    def twoDigits(self, num, lookup):
        if num in lookup:
            return lookup[num]
        return lookup[(num / 10) * 10] + " " + lookup[num % 10]
```

delete-node-in-a-linked-list.py

```
# DESC
# Example 2:
# Given linked list -- head = [4,5,1,9], which looks like following:
# Write a function to delete a node (except the tail) in a singly linked list, giv
# en only access to that node.
# Example 1:
# Note:

# NOTE
# The linked list will have at least two elements.
# Do not return anything from your function.
# The given node will not be the tail and it will always be a valid node of the li
# nked list.
# All of the nodes' values will be unique.

# EXAMPLE
# Input: head = [4,5,1,9], node = 5
# Output: [4,1,9]
# Explanation: You are given the
#  second node with value 5, the linked list should become 4 -> 1 -> 9 after calli
# ng your function.
# Input: head = [4,5,1,9], node = 1
# Output: [4,5,9]
# Explanation: You are given the
#  third node with value 1, the linked list should become 4 -> 5 -> 9 after callin
# g your function.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    # @param {ListNode} node
    # @return {void} Do not return anything, modify node in-place instead.
    def deleteNode(self, node):
        if node and node.next:
            node_to_delete = node.next
            node.val = node_to_delete.val
            node.next = node_to_delete.next
            del node_to_delete
```

circle-and-rectangle-overlapping.py

```
# circle-and-rectangle-overlapping is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def checkOverlap(self, radius, x_center, y_center, x1, y1, x2, y2):
        """
        :type radius: int
        :type x_center: int
        :type y_center: int
        :type x1: int
        :type y1: int
        :type x2: int
        :type y2: int
        :rtype: bool
        """
        x1 -= x_center
        y1 -= y_center
        x2 -= x_center
        y2 -= y_center
        x = x1 if x1 > 0 else x2 if x2 < 0 else 0
        y = y1 if y1 > 0 else y2 if y2 < 0 else 0
        return x**2 + y**2 <= radius**2


# Time:  O(1)
# Space: O(1)
class Solution2(object):
    def checkOverlap(self, radius, x_center, y_center, x1, y1, x2, y2):
        """
        :type radius: int
        :type x_center: int
        :type y_center: int
        :type x1: int
        :type y1: int
        :type x2: int
        :type y2: int
        :rtype: bool
        """
        x1 -= x_center
        y1 -= y_center
        x2 -= x_center
        y2 -= y_center
        x = min(abs(x1), abs(x2)) if x1*x2 > 0 else 0
        y = min(abs(y1), abs(y2)) if y1*y2 > 0 else 0
        return x**2 + y**2 <= radius**2
```

count-the-repetitions.py

```
# DESC
# Define S = [s,n] as the string S which consists of n connected strings s. For ex
# ample, ["abc", 3] ="abcabcabc".
# On the other hand, we define that string s1 can be obtained from string s2 if we
#  can remove some characters from s2 such that it becomes s1. For example, "abc"
#  can be obtained from "abdbec" based on our definition, but it can not be obtain
# ed from "acbbe".
# You are given two non-empty strings s1 and s2 (each at most 100 characters long)
#  and two integers 0  n1  106 and 1  n2  106. Now consider the strings S1 and
#  S2, where S1=[s1,n1] and S2=[s2,n2]. Find the maximum integer M such that [S2,M
# ] can be obtained from S1.
# Example:

# NOTE
#

# EXAMPLE
# Input:
# s1="acb", n1=4
# s2="ab", n2=2
#
# Return:
# 2

# Time:  O(s1 * min(s2, n1))
# Space: O(s2)

class Solution(object):
    def getMaxRepetitions(self, s1, n1, s2, n2):
        """
        :type s1: str
        :type n1: int
        :type s2: str
        :type n2: int
        :rtype: int
        """
        repeat_count = [0] * (len(s2)+1)
        lookup = {}
        j, count = 0, 0
        for k in xrange(1, n1+1):
            for i in xrange(len(s1)):
                if s1[i] == s2[j]:
                    j = (j + 1) % len(s2)
                    count += (j == 0)

            if j in lookup:   # cyclic
                i = lookup[j]
                prefix_count = repeat_count[i]
                pattern_count = (count - repeat_count[i]) * ((n1 - i) // (k - i))
                suffix_count = repeat_count[i + (n1 - i) % (k - i)] - repeat_count[i]
                return (prefix_count + pattern_count + suffix_count) / n2
            lookup[j] = k
            repeat_count[k] = count

        return repeat_count[n1] / n2  # not cyclic iff n1 <= s2
```

x-of-a-kind-in-a-deck-of-cards.py

```
# DESC
# Example 4:
# Constraints:
# In a deck of cards, each card has an integer written on it.
# Return true if and only if you can choose X >= 2 such that it is possible to spl
# it the entire deck into 1 or more groups of cards, where:
# Example 5:
# Example 2:
# Example 3:
# Example 1:

# NOTE
# 0 <= deck[i] < 10^4
# 1 <= deck.length <= 10^4
# Each group has exactly X cards.
# All the cards in each group have the same integer.

# EXAMPLE
# Input: deck = [1,1]
# Output: true
# Explanation: Possible partition [1,1].
# Input: deck = [1,2,3,4,4,3,2,1]
# Output: true
# Explanation: Possible partition [1,
# 1],[2,2],[3,3],[4,4].
# Input: deck = [1,1,2,2,2,2]
# Output: true
# Explanation: Possible partition [1,1],[
# 2,2],[2,2].
# Input: deck = [1,1,1,2,2,2,3,3]
# Output: false´
# Explanation: No possible partition.
# Input: deck = [1]
# Output: false
# Explanation: No possible partition.

# Time:  O(n * (logn)^2)
# Space: O(n)

import collections


class Solution(object):
    def hasGroupsSizeX(self, deck):
        """
        :type deck: List[int]
        :rtype: bool
        """
        def gcd(a, b):  # Time: O((logn)^2)
            while b:
                a, b = b, a % b
            return a

        vals = collections.Counter(deck).values()
        return reduce(gcd, vals) >= 2
```

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

```
# DESC
# The order of output does not matter.
# Example 2:
# Strings consists of lowercase English letters only and the length of both string
# s s and p will not be larger than 20,100.
# Example 1:
# Given a string s and a non-empty string p, find all the start indices of p's ana
# grams in s.

# NOTE
#

# EXAMPLE
# 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 star
# t index = 6 is "bac", which is an anagram of "abc".
# Input:
# s: "abab" p: "ab"
#
# Output:
# [0, 1, 2]
#
# Explanation:
# The substring with sta
# rt index = 0 is "ab", which is an anagram of "ab".
# The substring with start inde
# x = 1 is "ba", which is an anagram of "ab".
# The substring with start index = 2 i
# s "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
```

monotone-increasing-digits.py

```
# DESC
# Note:
# N is an integer in the range [0, 10^9].
# Given a non-negative integer N, find the largest number that is less than or equ
# al to N with monotone increasing digits.
# Example 1:
# (Recall that an integer has monotone increasing digits if and only if each pair
# of adjacent digits x and y satisfy x <= y.)
# Example 3:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: N = 10
# Output: 9
# Input: N = 332
# Output: 299
# Input: N = 1234
# Output: 1234

# Time:  O(logn) = O(1)
# Space: O(logn) = O(1)

class Solution(object):
    def monotoneIncreasingDigits(self, N):
        """
        :type N: int
        :rtype: int
        """
        nums = map(int, list(str(N)))
        leftmost_inverted_idx = len(nums)
        for i in reversed(xrange(1, len(nums))):
            if nums[i-1] > nums[i]:
                leftmost_inverted_idx = i
                nums[i-1] -= 1
        for i in xrange(leftmost_inverted_idx, len(nums)):
            nums[i] = 9
        return int("".join(map(str, nums)))
```

soup-servings.py

```
# DESC
# Return the probability that soup A will be empty first, plus half the probabilit
# y that A and B become empty at the same time.
# Notes:
# There are two types of soup: type A and type B. Initially we have N ml of each t
# ype of soup. There are four kinds of operations:
# When we serve some soup, we give it to someone and we no longer have it.  Each t
# urn, we will choose from the four operations with equal probability 0.25. If the
#  remaining volume of soup is not enough to complete the operation, we will serve
#  as much as we can.  We stop once we no longer have some quantity of both types
# of soup.
# Note that we do not have the operation where all 100 ml's of soup B are used first.

# NOTE
# Serve 100 ml of soup A and 0 ml of soup B
# Answers within 10^-6 of the true value will be accepted as correct.
# 0 <= N <= 10^9.
# Serve 50 ml of soup A and 50 ml of soup B
# Serve 75 ml of soup A and 25 ml of soup B
# Serve 25 ml of soup A and 75 ml of soup B

# EXAMPLE
# Example:
# Input: N = 50
# Output: 0.625
# Explanation:
# If we choose the first two op
# erations, A will become empty first. For the third operation, A and B will becom
# e empty at the same time. For the fourth operation, B will become empty first. S
# o the total probability of A becoming empty first plus half the probability that
#  A and B become empty at the same time, is 0.25 * (1 + 1 + 0.5 + 0) = 0.625.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def soupServings(self, N):
        """
        :type N: int
        :rtype: float
        """
        def dp(a, b, lookup):
            if (a, b) in lookup:
                return lookup[a, b]
            if a <= 0 and b <= 0:
                return 0.5
            if a <= 0:
                return 1.0
            if b <= 0:
                return 0.0
            lookup[a, b] = 0.25 * (dp(a-4, b, lookup) +
                                   dp(a-3, b-1, lookup) +
                                   dp(a-2, b-2, lookup) +
                                   dp(a-1, b-3, lookup))
            return lookup[a, b]

        if N >= 4800:
            return 1.0
```

```
lookup = {}
N = (N+24)//25
return dp(N, N, lookup)
```

fizz-buzz-multithreaded.py

```
# fizz-buzz-multithreaded is not found.
# 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()

    # 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

```
# DESC
# Example 3:
# A gene string can be represented by an 8-character long string, with choices fro
# m "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 stri
# ng.
# Example 1:
# Example 2:
# For example, "AACCGGTT" -> "AACCGGTA" is 1 mutation.
# Now, given 3 things - start, end, bank, your task is to determine what is the mi
# nimum number of mutations needed to mutate from "start" to "end". If there is no
#  such a mutation, return -1.
# Note:
# 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.

# NOTE
# You may assume start and end string is not the same.
# 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.

# EXAMPLE
# start: "AAAAACCC"
# end:   "AACCCCCC"
# bank: ["AAAACCCC", "AAACCCCC", "AACCCCCC"]
#
#
# return: 3
# start: "AACCGGTT"
# end:   "AACCGGTA"
# bank: ["AACCGGTA"]
#
# return: 1
# start: "AACCGGTT"
# end:   "AAACGGTA"
# bank: ["AACCGGTA", "AACCGCTA", "AAACGGTA"]
#
#
# return: 2

# Time:  O(n * b), n is the length of gene string, b is size of bank
# Space: O(b)

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
```

split-a-string-in-balanced-strings.py

```
# split-a-string-in-balanced-strings is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def balancedStringSplit(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, count = 0, 0
        for c in s:
            count += 1 if c == 'L' else -1
            if count == 0:
                result += 1
        return result
```

alphabet-board-path.py

```
# DESC
# (Here, the only positions that exist on the board are positions with letters on
# them.)
# We may make the following moves:
# Constraints:
# Here, board = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"], as shown in th
# e diagram below.
# On an alphabet board, we start at position (0, 0), corresponding to character bo
# ard[0][0].
# Example 2:
# Example 1:
# 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.
# board = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"]

# NOTE
# 'R' moves our position right one column, if the position exists on the board;
# 1 <= target.length <= 100
# 'D' moves our position down one row, if the position exists on the board;
# '!' adds the character board[r][c] at our current position (r, c) to the answer.
# target consists only of English lowercase letters.
# 'L' moves our position left one column, if the position exists on the board;
# 'U' moves our position up one row, if the position exists on the board;

# EXAMPLE
# Input: target = "leet"
# Output: "DDR!UURRR!!DDD!"
# Input: target = "code"
# Output: "RR!DDRR!UUL!R!"

# 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)
```

longest-repeating-substring.py

```
# longest-repeating-substring is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections


class Solution(object):
    def longestRepeatingSubstring(self, S):
        """
        :type S: str
        :rtype: int
        """
        M = 10**9+7
        D = 26

        def check(S, L):
            p = pow(D, L, M)
            curr = reduce(lambda x, y: (D*x+ord(y)-ord('a')) % M, S[:L], 0)
            lookup = collections.defaultdict(list)
            lookup[curr].append(L-1)
            result = 0
            for i in xrange(L, len(S)):
                curr = ((D*curr) % M + ord(S[i])-ord('a') -
                        ((ord(S[i-L])-ord('a'))*p) % M) % M
                if curr in lookup:
                    for j in lookup[curr]:
                        if S[j-L+1:j+1] == S[i-L+1:i+1]:
                            if result == 0:
                                result = i
                            return result-L+1
                lookup[curr].append(i)
            return result

        left, right = 0, len(S)-1
        while left <= right:
            mid = left + (right-left)//2
            if not check(S, mid):
                right = mid-1
            else:
                left = mid+1
        return right
```

replace-elements-with-greatest-element-on-right-side.py

```
# DESC
# After doing so, return the array.
# Example 1:
# Constraints:
# Given an array arr, replace every element in that array with the greatest elemen
# t among the elements to its right, and replace the last element with -1.

# NOTE
# 1 <= arr.length <= 10^4
# 1 <= arr[i] <= 10^5

# EXAMPLE
# Input: arr = [17,18,5,4,6,1]
# Output: [18,6,6,6,1,-1]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def replaceElements(self, arr):
        """
        :type arr: List[int]
        :rtype: List[int]
        """
        curr_max = -1
        for i in reversed(xrange(len(arr))):
            arr[i], curr_max = curr_max, max(curr_max, arr[i])
        return arr
```

stickers-to-spell-word.py

```
# DESC
# Output:
# Explanation:
# We are given N different types of stickers.  Each sticker has a lowercase Englis
# h word on it.
# Output:
# Example 1:
# You would like to spell out the given target string by cutting individual letter
# s from your collection of stickers and rearranging them.
# Example 2:
# Input:
# Explanation:
# Note:
# Input:
# What is the minimum number of stickers that you need to spell out the target?  I
# f the task is impossible, return -1.
# You can use each sticker more than once if you want, and you have infinite quant
# ities of each sticker.
# target

# NOTE
# stickers has length in the range [1, 50].
# The time limit may be more challenging than usual.  It is expected that a 50 sti
# cker test case can be solved within 35ms on average.
# target has length in the range [1, 15], and consists of lowercase English letters.
# In all test cases, all words were chosen randomly from the 1000 most common US E
# nglish words, and the target was chosen as a concatenation of two random words.
# stickers consists of lowercase English words (without apostrophes).

# EXAMPLE
# ["notice", "possible"], "basicbasic"
# We can't form the target "basicbasic" from cutting letters from the given stickers.
# We can use 2 "with" stickers, and 1 "example" sticker.
# After cutting and rearran
# ge the letters of those stickers, we can form the target "thehat".
# Also, this is
#  the minimum number of stickers necessary to form the target string.
# ["with", "example", "science"], "thehat"

# Time:  O(T * S^T)
# Space: O(T * S^T)

import collections


class Solution(object):
    def minStickers(self, stickers, target):
        """
        :type stickers: List[str]
        :type target: str
        :rtype: int
        """
        def minStickersHelper(sticker_counts, target, dp):
            if "".join(target) in dp:
                return dp["".join(target)]
            target_count = collections.Counter(target)
            result = float("inf")
            for sticker_count in sticker_counts:
```

```
            if sticker_count[target[0]] == 0:
                continue
            new_target = []
            for k in target_count.keys():
                if target_count[k] > sticker_count[k]:
                   new_target += [k]*(target_count[k] - sticker_count[k])
            if len(new_target) != len(target):
                num = minStickersHelper(sticker_counts, new_target, dp)
                if num != -1:
                    result = min(result, 1+num)
        dp["".join(target)] = -1 if result == float("inf") else result
        return dp["".join(target)]

    sticker_counts = map(collections.Counter, stickers)
    dp = { "":0 }
    return minStickersHelper(sticker_counts, target, dp)
```

candy-crush.py

```
# candy-crush is not found.
# Time:  O((R * C)^2)
# Space: O(1)

class Solution(object):
    def candyCrush(self, board):
        """
        :type board: List[List[int]]
        :rtype: List[List[int]]
        """
        R, C = len(board), len(board[0])
        changed = True

        while changed:
            changed = False

            for r in xrange(R):
                for c in xrange(C-2):
                    if abs(board[r][c]) == abs(board[r][c+1]) == abs(board[r][c+2]) != 0:
                        board[r][c] = board[r][c+1] = board[r][c+2] = -abs(board[r][c])
                        changed = True

            for r in xrange(R-2):
                for c in xrange(C):
                    if abs(board[r][c]) == abs(board[r+1][c]) == abs(board[r+2][c]) != 0:
                        board[r][c] = board[r+1][c] = board[r+2][c] = -abs(board[r][c])
                        changed = True

            for c in xrange(C):
                i = R-1
                for r in reversed(xrange(R)):
                    if board[r][c] > 0:
                        board[i][c] = board[r][c]
                        i -= 1
                for r in reversed(xrange(i+1)):
                    board[r][c] = 0

        return board
```

number-of-sub-arrays-with-odd-sum.py

```python
# number-of-sub-arrays-with-odd-sum is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numOfSubarrays(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        MOD = 10**9+7
        result, accu = 0, 0
        dp = [1, 0]
        for x in arr:
            accu ^= x&1
            dp[accu] += 1
            result = (result + dp[accu^1]) % MOD
        return result
```

n-ary-tree-level-order-traversal.py

```
# DESC
# Example 1:
# Given an n-ary tree, return the level order traversal of its nodes' values.
# Constraints:
# Example 2:
# Nary-Tree input serialization is represented in their level order traversal, eac
# h group of children is separated by the null value (See examples).

# NOTE
# The height of the n-ary tree is less than or equal to 1000
# The total number of nodes is between [0, 10^4]

# EXAMPLE
# Input: root = [1,null,3,2,4,null,5,6]
# Output: [[1],[3,2,4],[5,6]]
# Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,1
# 2,null,13,null,null,14]
# Output: [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

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

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


class Solution(object):
    def levelOrder(self, root):
        """
        :type root: Node
        :rtype: List[List[int]]
        """
        if not root:
            return []
        result, q = [], [root]
        while q:
            result.append([node.val for node in q])
            q = [child for node in q for child in node.children if child]
        return result
```

maximum-candies-you-can-get-from-boxes.py

```python
# maximum-candies-you-can-get-from-boxes is not found.
# Time:  O(n^2)
# Space: O(n)

import collections


class Solution(object):
    def maxCandies(self, status, candies, keys, containedBoxes, initialBoxes):
        """
        :type status: List[int]
        :type candies: List[int]
        :type keys: List[List[int]]
        :type containedBoxes: List[List[int]]
        :type initialBoxes: List[int]
        :rtype: int
        """
        result = 0
        q = collections.deque(initialBoxes)
        while q:
            changed = False
            for _ in xrange(len(q)):
                box = q.popleft()
                if not status[box]:
                    q.append(box)
                    continue
                changed = True
                result += candies[box]
                for contained_key in keys[box]:
                    status[contained_key] = 1
                for contained_box in containedBoxes[box]:
                    q.append(contained_box)
            if not changed:
                break
        return result
```

keys-and-rooms.py

```
# DESC
# Return true if and only if you can enter every room.
# Note:
# Initially, all the rooms start locked (except for room 0).
# Example 2:
# You can walk back and forth between rooms freely.
# Example 1:
# There are N rooms and you start in room 0.  Each room has a distinct number in 0
# , 1, 2, ..., N-1, and each room may have some keys to access the next room.
# Formally, each room i has a list of keys rooms[i], and each key rooms[i][j] is a
# n integer in [0, 1, ..., N-1] where N = rooms.length.  A key rooms[i][j] = v ope
# ns the room with number v.

# NOTE
# The number of keys in all rooms combined is at most 3000.
# 1 <= rooms.length <= 1000
# 0 <= rooms[i].length <= 1000

# EXAMPLE
# Input: [[1,3],[3,0,1],[2],[0]]
# Output: false
# Explanation: We can't enter the roo
# m with number 2.
# Input: [[1],[2],[3],[]]
# Output: true
# Explanation:
# We start in room 0, and pick
#  up key 1.
# We then go to room 1, and pick up key 2.
# We then go to room 2, and pi
# ck up key 3.
# We then go to room 3.  Since we were able to go to every room, we r
# eturn true.

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

class Solution(object):
    def canVisitAllRooms(self, rooms):
        """
        :type rooms: List[List[int]]
        :rtype: bool
        """
        lookup = set([0])
        stack = [0]
        while stack:
            node = stack.pop()
            for nei in rooms[node]:
                if nei not in lookup:
                    lookup.add(nei)
                    if len(lookup) == len(rooms):
                        return True
                    stack.append(nei)
        return len(lookup) == len(rooms)
```

all-oone-data-structure.py

```
# all-oone-data-structure is not found.
# Time:  O(1), per operation
# Space: O(k)

class Node(object):
    """
    double linked list node
    """
    def __init__(self, value, keys):
        self.value = value
        self.keys = keys
        self.prev = None
        self.next = None


class LinkedList(object):
    def __init__(self):
        self.head, self.tail = Node(0, set()), Node(0, set())
        self.head.next, self.tail.prev = self.tail, self.head

    def insert(self, pos, node):
        node.prev, node.next = pos.prev, pos
        pos.prev.next, pos.prev = node, node
        return node

    def erase(self, node):
        node.prev.next, node.next.prev = node.next, node.prev
        del node

    def empty(self):
        return self.head.next is self.tail

    def begin(self):
        return self.head.next

    def end(self):
        return self.tail

    def front(self):
        return self.head.next

    def back(self):
        return self.tail.prev


class AllOne(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.bucket_of_key = {}
        self.buckets = LinkedList()

    def inc(self, key):
        """
        Inserts a new key <Key> with value 1. Or increments an existing key by 1.
        :type key: str
```

```
        :rtype: void
        """
        if key not in self.bucket_of_key:
            self.bucket_of_key[key] = self.buckets.insert(self.buckets.begin(), Node(0, set([key])))

        bucket, next_bucket = self.bucket_of_key[key], self.bucket_of_key[key].next
        if next_bucket is self.buckets.end() or next_bucket.value > bucket.value+1:
            next_bucket = self.buckets.insert(next_bucket, Node(bucket.value+1, set()))
        next_bucket.keys.add(key)
        self.bucket_of_key[key] = next_bucket

        bucket.keys.remove(key)
        if not bucket.keys:
            self.buckets.erase(bucket)

    def dec(self, key):
        """
        Decrements an existing key by 1. If Key's value is 1, remove it from the data structure.
        :type key: str
        :rtype: void
        """
        if key not in self.bucket_of_key:
            return

        bucket, prev_bucket = self.bucket_of_key[key], self.bucket_of_key[key].prev
        self.bucket_of_key.pop(key, None)
        if bucket.value > 1:
            if bucket is self.buckets.begin() or prev_bucket.value < bucket.value-1:
                prev_bucket = self.buckets.insert(bucket, Node(bucket.value-1, set()))
            prev_bucket.keys.add(key)
            self.bucket_of_key[key] = prev_bucket

        bucket.keys.remove(key)
        if not bucket.keys:
            self.buckets.erase(bucket)

    def getMaxKey(self):
        """
        Returns one of the keys with maximal value.
        :rtype: str
        """
        if self.buckets.empty():
            return ""
        return iter(self.buckets.back().keys).next()

    def getMinKey(self):
        """
        Returns one of the keys with Minimal value.
        :rtype: str
        """
        if self.buckets.empty():
            return ""
        return iter(self.buckets.front().keys).next()
```

pancake-sorting.py

```
# DESC
# Note:
# Given an array A, we can perform a pancake flip: We choose some positive integer
#  k <= A.length, then reverse the order of the first k elements of A.  We want to
#  perform zero or more pancake flips (doing them one after another in succession)
#  to sort the array A.
# Example 1:
# Return the k-values corresponding to a sequence of pancake flips that sort A.  A
# ny valid answer that sorts the array within 10 * A.length flips will be judged a
# s correct.
# Example 2:

# NOTE
# 1 <= A.length <= 100
# A[i] is a permutation of [1, 2, ..., A.length]

# EXAMPLE
# Input: [3,2,4,1]
# Output: [4,2,4,3]
# Explanation:
# We perform 4 pancake flips, wit
# h 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.
# Input: [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.

# 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
```

maximum-length-of-a-concatenated-string-with-unique-characters.py

```python
# maximum-length-of-a-concatenated-string-with-unique-characters is not found.
# Time:  O(n) ~ O(2^n)
# Space: O(1) ~ O(2^n)

power = [1]
log2 = {1:0}
for i in xrange(1, 26):
    power.append(power[-1]<<1)
    log2[power[i]] = i


class Solution(object):
    def maxLength(self, arr):
        """
        :type arr: List[str]
        :rtype: int
        """
        def bitset(s):
            result = 0
            for c in s:
                if result & power[ord(c)-ord('a')]:
                    return 0
                result |= power[ord(c)-ord('a')]
            return result

        def number_of_one(n):
            result = 0
            while n:
                n &= n-1
                result += 1
            return result

        dp = [0]
        for x in arr:
            x_set = bitset(x)
            if not x_set:
                continue
            curr_len = len(dp)
            for i in xrange(curr_len):
                if dp[i] & x_set:
                    continue
                dp.append(dp[i] | x_set)
        return max(number_of_one(s_set) for s_set in dp)


# Time:  O(2^n)
# Space: O(1)
class Solution2(object):
    def maxLength(self, arr):
        """
        :type arr: List[str]
        :rtype: int
        """
        def bitset(s):
            result = 0
            for c in s:
                if result & power[ord(c)-ord('a')]:
                    return 0
```

```
        result |= power[ord(c)-ord('a')]
    return result

bitsets = [bitset(x) for x in arr]
result = 0
for i in xrange(power[len(arr)]):
    curr_bitset, curr_len = 0, 0
    while i:
        j = i & -i  # rightmost bit
        i ^= j
        j = log2[j]  # log2(j)
        if not bitsets[j] or (curr_bitset & bitsets[j]):
            break
        curr_bitset |= bitsets[j]
        curr_len += len(arr[j])
    else:
        result = max(result, curr_len)
return result
```

01-matrix.py

```
# 01-matrix is not found.
# 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))
                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
```

analyze-user-website-visit-pattern.py

```
# analyze-user-website-visit-pattern is not found.
# Time:  O(n^3)
# Space: O(n^3)

import collections
import itertools


class Solution(object):
    def mostVisitedPattern(self, username, timestamp, website):
        """
        :type username: List[str]
        :type timestamp: List[int]
        :type website: List[str]
        :rtype: List[str]
        """
        lookup = collections.defaultdict(list)
        A = zip(timestamp, username, website)
        A.sort()
        for t, u, w in A:
            lookup[u].append(w)
        count = sum([collections.Counter(set(itertools.combinations(lookup[u], 3))) for u in lookup], collecti
        return list(min(count, key=lambda x: (-count[x], x)))
```

shortest-way-to-form-string.py

```python
# shortest-way-to-form-string is not found.
# Time:  O(m + n), m is the length of source
#                , n is the length of target
# Space: O(m)

# greedy solution
class Solution(object):
    def shortestWay(self, source, target):
        """
        :type source: str
        :type target: str
        :rtype: int
        """
        lookup = [[None for _ in xrange(26)] for _ in xrange(len(source)+1)]
        find_char_next_pos = [None]*26
        for i in reversed(xrange(len(source))):
            find_char_next_pos[ord(source[i])-ord('a')] = i+1
            lookup[i] = list(find_char_next_pos)

        result, start = 1, 0
        for c in target:
            start = lookup[start][ord(c)-ord('a')]
            if start != None:
                continue
            result += 1
            start = lookup[0][ord(c)-ord('a')]
            if start == None:
                return -1
        return result
```

smallest-common-region.py

```python
# smallest-common-region is not found.
# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    def findSmallestRegion(self, regions, region1, region2):
        """
        :type regions: List[List[str]]
        :type region1: str
        :type region2: str
        :rtype: str
        """
        parents = {region[i] : region[0]
                   for region in regions
                   for i in xrange(1, len(region))}
        lookup = {region1}
        while region1 in parents:
            region1 = parents[region1]
            lookup.add(region1)
        while region2 not in lookup:
            region2 = parents[region2]
        return region2
```

rotated-digits.py

```
# DESC
# Now given a positive number N, how many numbers X from 1 to N are good?
# X is a good number if after rotating each digit individually by 180 degrees, we
# get a valid number that is different from X.  Each digit must be rotated - we ca
# nnot choose to leave it alone.
# A number is valid if each digit remains a digit after rotation. 0, 1, and 8 rota
# te to themselves; 2 and 5 rotate to each other (on this case they are rotated in
#  a different direction, in other words 2 or 5 gets mirrored); 6 and 9 rotate to
# each other, and the rest of the numbers do not rotate to any other number and be
# come invalid.
# Note:

# NOTE
# N  will be in range [1, 10000].

# EXAMPLE
# Example:
# Input: 10
# Output: 4
# Explanation:
# There are four good numbers in the ra
# nge [1, 10] : 2, 5, 6, 9.
# Note that 1 and 10 are not good numbers, since they re
# main unchanged after rotating.

# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def rotatedDigits(self, N):
        """
        :type N: int
        :rtype: int
        """
        A = map(int, str(N))
        invalid, diff = set([3, 4, 7]), set([2, 5, 6, 9])
        def dp(A, i, is_prefix_equal, is_good, lookup):
            if i == len(A): return int(is_good)
            if (i, is_prefix_equal, is_good) not in lookup:
                result = 0
                for d in xrange(A[i]+1 if is_prefix_equal else 10):
                    if d in invalid: continue
                    result += dp(A, i+1,
                                 is_prefix_equal and d == A[i],
                                 is_good or d in diff,
                                 lookup)
                lookup[i, is_prefix_equal, is_good] = result
            return lookup[i, is_prefix_equal, is_good]

        lookup = {}
        return dp(A, 0, True, False, lookup)


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def rotatedDigits(self, N):
        """
```

```
        :type N: int
        :rtype: int
        """
        INVALID, SAME, DIFF = 0, 1, 2
        same, diff = [0, 1, 8], [2, 5, 6, 9]
        dp = [0] * (N+1)
        dp[0] = SAME
        for i in xrange(N//10+1):
            if dp[i] != INVALID:
                for j in same:
                    if i*10+j <= N:
                        dp[i*10+j] = max(SAME, dp[i])
                for j in diff:
                    if i*10+j <= N:
                        dp[i*10+j] = DIFF
        return dp.count(DIFF)


# Time:  O(nlogn) = O(n), because O(logn) = O(32) by this input
# Space: O(logn) = O(1)
class Solution3(object):
    def rotatedDigits(self, N):
        """
        :type N: int
        :rtype: int
        """
        invalid, diff = set(['3', '4', '7']), set(['2', '5', '6', '9'])
        result = 0
        for i in xrange(N+1):
            lookup = set(list(str(i)))
            if invalid & lookup:
                continue
            if diff & lookup:
                result += 1
        return result
```

maximum-product-subarray.py

```python
# maximum-product-subarra is not found.
# 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
```

trapping-rain-water.py

```
# DESC
# Given n non-negative integers representing an elevation map where the width of e
# ach bar is 1, compute how much water it is able to trap after raining.
# The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In th
# is case, 6 units of rain water (blue section) are being trapped. Thanks Marcos f
# or contributing this image!
# Example:

# NOTE
#

# EXAMPLE
# Input: [0,1,0,2,1,0,1,3,2,1,2,1]
# Output: 6

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A, a list of integers
    # @return an integer
    def trap(self, A):
        result = 0
        top = 0
        for i in xrange(len(A)):
            if A[top] < A[i]:
                top = i

        second_top = 0
        for i in xrange(top):
            if A[second_top] < A[i]:
                second_top = i
            result += A[second_top] - A[i]

        second_top = len(A) - 1
        for i in reversed(xrange(top, len(A))):
            if A[second_top] < A[i]:
                second_top = i
            result += A[second_top] - A[i]

        return result

# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param A, a list of integers
    # @return an integer
    def trap(self, A):
        result = 0
        stack = []

        for i in xrange(len(A)):
            mid_height = 0
            while stack:
                [pos, height] = stack.pop()
                result += (min(height, A[i]) - mid_height) * (i - pos - 1)
                mid_height = height
```

```
            if A[i] < height:
                stack.append([pos, height])
                break
        stack.append([i, A[i]])

    return result
```

number-of-valid-subarrays.py

```python
# number-of-valid-subarrays is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def validSubarrays(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        s = []
        for num in nums:
            while s and s[-1] > num:
                s.pop()
            s.append(num);
            result += len(s)
        return result
```

distant-barcodes.py

```python
# DESC
# In a warehouse, there is a row of barcodes, where the i-th barcode is barcodes[i].
# Example 1:
# Note:
# Example 2:
# Rearrange the barcodes so that no two adjacent barcodes are equal.  You may retu
# rn any answer, and it is guaranteed an answer exists.

# NOTE
# 1 <= barcodes[i] <= 10000
# 1 <= barcodes.length <= 10000

# EXAMPLE
# Input: [1,1,1,1,2,2,3,3]
# Output: [1,3,1,3,2,1,2,1]
# Input: [1,1,1,2,2,2]
# Output: [2,1,2,1,2,1]

# 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

```
# DESC
# Note:
# In an array A of 0s and 1s, how many non-empty subarrays have sum S?
# Example 1:

# NOTE
# 0 <= S <= A.length
# A[i] is either 0 or 1.
# A.length <= 30000

# EXAMPLE
# 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]

# 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

```
# DESC
# Given a root node reference of a BST and a key, delete the node with the given k
# ey in the BST. Return the root node reference (possibly updated) of the BST.
# Example:
# Note: Time complexity should be O(height of tree).
# Basically, the deletion can be divided into two stages:

# NOTE
# If the node is found, delete the node.
# Search for a node to remove.

# EXAMPLE
# root = [5,3,6,2,4,null,7]
# key = 3
#
#     5
#    / \
#   3   6
#  / \   \
# 2   4   7
#
# Give
# n 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
                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

```
# DESC
# Given an array of integers with possible duplicates, randomly output the index o
# f a given target number. You can assume that the given target number must exist
# in the array.
# Example:
# Note:
#
# The array size can be very large. Solution that uses too much extra space
#  will not pass the judge.

# NOTE
#

# 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 e
# qual probability of returning.
# solution.pick(3);
#
# // pick(1) should return 0. Si
# nce 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

```
# DESC
# Example 2:
# Note:
# Given a non-empty string s and a dictionary wordDict containing a list of non-em
# pty words, determine if s can be segmented into a space-separated sequence of on
# e or more dictionary words.
# Example 1:
# Example 3:

# NOTE
# You may assume the dictionary does not contain duplicate words.
# The same word in the dictionary may be reused multiple times in the segmentation.

# EXAMPLE
# 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.
# Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
# Output:
#  false
# Input: s = "leetcode", wordDict = ["leet", "code"]
# Output: true
# Explanation: Ret
# urn true because "leetcode" can be segmented as "leet code".

# 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]
```

longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit.py

```
# longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def longestSubarray(self, nums, limit):
        """
        :type nums: List[int]
        :type limit: int
        :rtype: int
        """
        max_dq, min_dq = collections.deque(), collections.deque()
        left = 0
        for right, num in enumerate(nums):
            while max_dq and nums[max_dq[-1]] <= num:
                max_dq.pop()
            max_dq.append(right)
            while min_dq and nums[min_dq[-1]] >= num:
                min_dq.pop()
            min_dq.append(right)
            if nums[max_dq[0]]-nums[min_dq[0]] > limit:
                if max_dq[0] == left:
                    max_dq.popleft()
                if min_dq[0] == left:
                    min_dq.popleft()
                left += 1  # advance left by one to not count in result
        return len(nums)-left


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


class Solution2(object):
    def longestSubarray(self, nums, limit):
        """
        :type nums: List[int]
        :type limit: int
        :rtype: int
        """
        max_dq, min_dq = collections.deque(), collections.deque()
        result, left = 0, 0
        for right, num in enumerate(nums):
            while max_dq and nums[max_dq[-1]] <= num:
                max_dq.pop()
            max_dq.append(right)
            while min_dq and nums[min_dq[-1]] >= num:
                min_dq.pop()
            min_dq.append(right)
            while nums[max_dq[0]]-nums[min_dq[0]] > limit:  # both always exist "right" element
                if max_dq[0] == left:
                    max_dq.popleft()
                if min_dq[0] == left:
                    min_dq.popleft()
```

```
            left += 1
        result = max(result, right-left+1)
    return result
```

number-of-segments-in-a-string.py

```
# DESC
# Example:
# Please note that the string does not contain any non-printable characters.
# Count the number of segments in a string, where a segment is defined to be a con
# tiguous sequence of non-space characters.

# NOTE
#

# EXAMPLE
# Input: "Hello, my name is John"
# Output: 5

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countSegments(self, s):
        """
        :type s: str
        :rtype: int
        """
        result = int(len(s) and s[-1] != ' ')
        for i in xrange(1, len(s)):
            if s[i] == ' ' and s[i-1] != ' ':
                result += 1
        return result

    def countSegments2(self, s):
        """
        :type s: str
        :rtype: int
        """
        return len([i for i in s.strip().split(' ') if i])
```

video-stitching.py

```
# DESC
# Example 3:
# You are given a series of video clips from a sporting event that lasted T second
# s.  These video clips can be overlapping with each other and have varied lengths
# .
# Example 2:
# 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].
# Constraints:
# Example 1:
# Return the minimum number of clips needed so that we can cut the clips into segm
# ents that cover the entire sporting event ([0, T]).  If the task is impossible,
# return -1.
# clips[i]
# Example 4:

# NOTE
# 1 <= clips.length <= 100
# 0 <= T <= 100
# 0 <= clips[i][0] <= clips[i][1] <= 100

# EXAMPLE
# Input: clips = [[0,1],[1,2]], T = 5
# Output: -1
# Explanation:
# We can't cover [0,5
# ] with only [0,1] and [1,2].
# 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 clip
# s [0,4], [4,7], and [6,9].
# Input: clips = [[0,4],[2,8]], T = 5
# Output: 2
# Explanation:
# Notice you can have
# extra video after the event ends.
# Input: clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10
# Output: 3
# Explanat
# ion:
# We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.
# Then, we can r
# econstruct 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].

# Time:  O(nlogn)
# 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
```

single-row-keyboard.py

```python
# single-row-keyboard is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def calculateTime(self, keyboard, word):
        """
        :type keyboard: str
        :type word: str
        :rtype: int
        """
        lookup = {c:i for i, c in enumerate(keyboard)}
        result, prev = 0, 0
        for c in word:
            result += abs(lookup[c]-prev)
            prev = lookup[c]
        return result
```

remove-duplicates-from-sorted-array.py

```
# remove-duplicates-from-sorted-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param a list of integers
    # @return an integer
    def removeDuplicates(self, A):
        if not A:
            return 0

        last = 0
        for i in xrange(len(A)):
            if A[last] != A[i]:
                last += 1
                A[last] = A[i]
        return last + 1
```

valid-word-abbreviation.py

```python
# valid-word-abbreviation is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def validWordAbbreviation(self, word, abbr):
        """
        :type word: str
        :type abbr: str
        :rtype: bool
        """
        i , digit = 0, 0
        for c in abbr:
            if c.isdigit():
                if digit == 0 and c == '0':
                    return False
                digit *= 10
                digit += int(c)
            else:
                if digit:
                    i += digit
                    digit = 0
                if i >= len(word) or word[i] != c:
                    return False
                i += 1
        if digit:
            i += digit

        return i == len(word)
```

shortest-path-with-alternating-colors.py

```
# DESC
# [i, j]
# Example 2:
# Example 4:
# Example 5:
# Example 3:
# Example 1:
# Each [i, j] in red_edges denotes a red directed edge from node i to node j.  Sim
# ilarly, each [i, j] in blue_edges denotes a blue directed edge from node i to no
# de j.
# Return an array answer of length n, where each answer[X] is the length of the sh
# ortest 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).
# Constraints:
# Consider a directed graph, with nodes labelled 0, 1, ..., n-1.  In this graph, e
# ach edge is either red or blue, and there could be self-edges or parallel edges.

# NOTE
# red_edges[i].length == blue_edges[i].length == 2
# red_edges.length <= 400
# 1 <= n <= 100
# blue_edges.length <= 400
# 0 <= red_edges[i][j], blue_edges[i][j] < n

# EXAMPLE
# Input: n = 3, red_edges = [[0,1]], blue_edges = [[2,1]]
# Output: [0,1,-1]
# Input: n = 3, red_edges = [[1,0]], blue_edges = [[2,1]]
# Output: [0,-1,-1]
# Input: n = 3, red_edges = [[0,1]], blue_edges = [[1,2]]
# Output: [0,1,2]
# Input: n = 3, red_edges = [[0,1],[1,2]], blue_edges = []
# Output: [0,1,-1]
# Input: n = 3, red_edges = [[0,1],[0,2]], blue_edges = [[1,0]]
# Output: [0,1,1]

# 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)]
        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
# maximum-width-ram is not found.
# 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
```

verbal-arithmetic-puzzle.py

```
# verbal-arithmetic-puzzle is not found.
# Time:  O(10! * n * l)
# Space: O(n * l)

import collections


class Solution(object):
    def isSolvable(self, words, result):
        """
        :type words: List[str]
        :type result: str
        :rtype: bool
        """
        def backtracking(words, result, i, j, carry, lookup, used):
            if j == len(result):
                return carry == 0

            if i != len(words):
                if j >= len(words[i]) or words[i][j] in lookup:
                    return backtracking(words, result, i+1, j, carry, lookup, used)
                for val in xrange(10):
                    if val in used or (val == 0 and j == len(words[i])-1):
                        continue
                    lookup[words[i][j]] = val
                    used.add(val)
                    if backtracking(words, result, i+1, j, carry, lookup, used):
                        return True
                    used.remove(val)
                    del lookup[words[i][j]]
                return False

            carry, val = divmod(carry + sum(lookup[w[j]] for w in words if j < len(w)), 10)
            if result[j] in lookup:
                return val == lookup[result[j]] and \
                       backtracking(words, result, 0, j+1, carry, lookup, used)
            if val in used or (val == 0 and j == len(result)-1):
                return False
            lookup[result[j]] = val
            used.add(val)
            if backtracking(words, result, 0, j+1, carry, lookup, used):
                return True
            used.remove(val)
            del lookup[result[j]]
            return False

        return backtracking([w[::-1] for w in words], result[::-1], 0, 0, 0, {}, set())
```

maximum-sum-bst-in-binary-tree.py

```
# maximum-sum-bst-in-binary-tree is not found.
# 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 maxSumBST(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result = 0
        stk = [[root, None, []]]
        while stk:
            node, tmp, ret = stk.pop()
            if tmp:
                lvalid, lsum, lmin, lmax = tmp[0]
                rvalid, rsum, rmin, rmax = tmp[1]
                if lvalid and rvalid and lmax < node.val < rmin:
                    total = lsum + node.val + rsum
                    result = max(result, total)
                    ret[:] = [True, total, min(lmin, node.val), max(node.val, rmax)]
                    continue
                ret[:] = [False, 0, 0, 0]
                continue
            if not node:
                ret[:] = [True, 0, float("inf"), float("-inf")]
                continue
            new_tmp = [[], []]
            stk.append([node, new_tmp, ret])
            stk.append([node.right, None, new_tmp[1]])
            stk.append([node.left, None, new_tmp[0]])
        return result


# Time:  O(n)
# Space: O(h)
# dfs solution with recursion
class Solution2(object):
    def maxSumBST(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, result):
            if not node:
                return True, 0, float("inf"), float("-inf")
            lvalid, lsum, lmin, lmax = dfs(node.left, result)
            rvalid, rsum, rmin, rmax = dfs(node.right, result)
            if lvalid and rvalid and lmax < node.val < rmin:
```

```
            total = lsum + node.val + rsum
            result[0] = max(result[0], total)
            return True, total, min(lmin, node.val), max(node.val, rmax)
        return False, 0, 0, 0

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

valid-permutations-for-di-sequence.py

```
# DESC
# Example 1:
# We are given S, a length n string of characters from the set {'D', 'I'}. (These
# letters stand for "decreasing" and "increasing".)
# A valid permutation is a permutation P[0], P[1], ..., P[n] of integers {0, 1, ..
# ., n}, such that for all i:
# Note:
# How many valid permutations are there?  Since the answer may be large, return yo
# ur answer modulo 10^9 + 7.
# P[0], P[1], ..., P[n]

# NOTE
# 1 <= S.length <= 200
# If S[i] == 'I', then P[i] < P[i+1].
# S consists only of characters from the set {'D', 'I'}.
# If S[i] == 'D', then P[i] > P[i+1], and;

# EXAMPLE
# Input: "DID"
# Output: 5
# Explanation:
# The 5 valid permutations of (0, 1, 2, 3) ar
# e:
# (1, 0, 3, 2)
# (2, 0, 3, 1)
# (2, 1, 3, 0)
# (3, 0, 2, 1)
# (3, 1, 2, 0)

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

class Solution(object):
    def numPermsDISequence(self, S):
        """
        :type S: str
        :rtype: int
        """
        dp = [1]*(len(S)+1)
        for c in S:
            if c == "I":
                dp = dp[:-1]
                for i in xrange(1, len(dp)):
                    dp[i] += dp[i-1]
            else:
                dp = dp[1:]
                for i in reversed(xrange(len(dp)-1)):
                    dp[i] += dp[i+1]
        return dp[0] % (10**9+7)
```

maximum-number-of-darts-inside-of-a-circular-dartboard.py

```python
# maximum-number-of-darts-inside-of-a-circular-dartboard is not found.
# Time:  O(n^2 * logn)
# Space: O(n)

import math


# angle sweep solution
# great explanation:
# https://leetcode.com/problems/maximum-number-of-darts-inside-of-a-circular-dartboard/discuss/636345/Python-O
class Solution(object):
    def numPoints(self, points, r):
        """
        :type points: List[List[int]]
        :type r: int
        :rtype: int
        """
        def count_points(points, r, i):
            angles = []
            for j in xrange(len(points)):
                if i == j:
                    continue
                dx, dy = points[i][0]-points[j][0], points[i][1]-points[j][1]
                d = math.sqrt(dx**2 + dy**2)
                if d > 2*r:
                    continue
                delta, angle = math.acos(d/(2*r)), math.atan2(dy, dx)
                angles.append((angle-delta, 0)), angles.append((angle+delta, 1))
            angles.sort()
            result, count = 1, 1
            for _, is_closed in angles:  # angle sweep
                if not is_closed:
                    count += 1
                else:
                    count -= 1
                result = max(result, count)
            return result

        return max(count_points(points, r, i) for i in xrange(len(points)))
```

sentence-similarity.py

```
# sentence-similarit is not found.
# Time:  O(n + p)
# Space: O(p)

import itertools


class Solution(object):
    def areSentencesSimilar(self, words1, words2, pairs):
        """
        :type words1: List[str]
        :type words2: List[str]
        :type pairs: List[List[str]]
        :rtype: bool
        """
        if len(words1) != len(words2): return False
        lookup = set(map(tuple, pairs))
        return all(w1 == w2 or (w1, w2) in lookup or (w2, w1) in lookup \
                   for w1, w2 in itertools.izip(words1, words2))
```

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

```
# DESC
# As a reminder, a binary search tree is a tree that satisfies these constraints:
# Constraints:
# Given the root of a binary search tree with distinct values, modify it so that e
# very node has a new value equal to the sum of the values of the original tree th
# at are greater than or equal to node.val.
# Example 1:

# NOTE
# The left subtree of a node contains only nodes with keys less than the node's key.
# The given tree is a binary search tree.
# Each node will have value between 0 and 100.
# The number of nodes in the tree is between 1 and 100.
# Both the left and right subtrees must also be binary search trees.
# The right subtree of a node contains only nodes with keys greater than the node'
# s key.

# EXAMPLE
# Input: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
# Output: [30,36,21,36,35
# ,26,15,null,null,null,33,null,null,null,8]

# 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)
```

smallest-rotation-with-highest-score.py

```
# DESC
# For example, if we have [2, 4, 1, 3, 0], and we rotate by K = 2, it becomes [1,
# 3, 0, 2, 4].  This is worth 3 points because 1 > 0 [no points], 3 > 1 [no points
# ], 0 <= 2 [one point], 2 <= 3 [one point], 4 <= 4 [one point].
# Given an array A, we may rotate it by a non-negative integer K so that the array
#  becomes A[K], A[K+1], A{K+2], ... A[A.length - 1], A[0], A[1], ..., A[K-1].  Af
# terward, any entries that are less than or equal to their index are worth 1 poin
# t.
# So we should choose K = 3, which has the highest score.
# Over all possible rotations, return the rotation index K that corresponds to the
#  highest score we could receive.  If there are multiple answers, return the smal
# lest such index K.
# [2, 4, 1, 3, 0]
# Note:

# NOTE
# A[i] will be in the range [0, A.length].
# A will have length at most 20000.

# EXAMPLE
# Example 1:
# Input: [2, 3, 1, 4, 0]
# Output: 3
# Explanation:
# Scores for each K are
#  listed below:
# K = 0,  A = [2,3,1,4,0],    score 2
# K = 1,  A = [3,1,4,0,2],
# score 3
# K = 2,  A = [1,4,0,2,3],    score 3
# K = 3,  A = [4,0,2,3,1],    score 4
#
# K = 4,  A = [0,2,3,1,4],    score 3
# Example 2:
# Input: [1, 3, 0, 2, 4]
# Output: 0
# Explanation:  A will always have 3 p
# oints no matter how it shifts.
# So we will choose the smallest K, which is 0.

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

class Solution(object):
    def bestRotation(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        N = len(A)
        change = [1] * N
        for i in xrange(N):
            change[(i-A[i]+1)%N] -= 1
        for i in xrange(1, N):
            change[i] += change[i-1]
        return change.index(max(change))
```

goat-latin.py

```
# DESC
# Example 1:
# Example 2:
# We would like to convert the sentence to "Goat Latin" (a made-up language simila
# r to Pig Latin.)
# A sentence S is given, composed of words separated by spaces. Each word consists
#  of lowercase and uppercase letters only.
# Notes:
# Return the final sentence representing the conversion from S to Goat Latin.
# The rules of Goat Latin are as follows:

# NOTE
# If a word begins with a consot (i.e. not a vowel), remove the first letter and a
# ppend it to the end, then add "ma".
#
#         For example, the word "goat" becomes "oatg
# ma".
# Add one letter 'a' to the end of each word per its word index in the sentence, s
# tarting with 1.
#
#         For example, the first word gets "a" added to the end, the sec
# ond word gets "aa" added to the end and so on.
# 1 <= S.length <= 150.
# S contains only uppercase, lowercase and spaces. Exactly one space between each word.
# If a word begins with a vowel (a, e, i, o, or u), append "ma" to the end of the
# word.
#
#         For example, the word 'apple' becomes 'applema'.

# EXAMPLE
# Input: "I speak Goat Latin"
# Output: "Imaa peaksmaaa oatGmaaaa atinLmaaaaa"
# Input: "The quick brown fox jumped over the lazy dog"
# Output: "heTmaa uickqmaaa
# rownbmaaaa oxfmaaaaa umpedjmaaaaaa overmaaaaaaa hetmaaaaaaaa azylmaaaaaaaaa ogdm
# aaaaaaaaaa"

# Time:  O(n + w^2), n = w * l,
#                    n is the length of S,
#                    w is the number of word,
#                    l is the average length of word
# Space: O(n)

class Solution(object):
    def toGoatLatin(self, S):
        """
        :type S: str
        :rtype: str
        """
        def convert(S):
            vowel = set('aeiouAEIOU')
            for i, word in enumerate(S.split(), 1):
                if word[0] not in vowel:
                    word = word[1:] + word[:1]
                yield word + 'ma' + 'a'*i
        return " ".join(convert(S))
```

android-unlock-patterns.py

```
# android-unlock-patterns is not found.
# Time:  O(9^2 * 2^9)
# Space: O(9 * 2^9)


# DP solution.
class Solution(object):
    def numberOfPatterns(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        def merge(used, i):
            return used | (1 << i)

        def number_of_keys(i):
            number = 0
            while i > 0:
                i &= i - 1
                number += 1
            return number

        def contain(used, i):
            return bool(used & (1 << i))

        def convert(i, j):
            return 3 * i + j

        # dp[i][j]: i is the set of the numbers in binary representation,
        #           dp[i][j] is the number of ways ending with the number j.
        dp = [[0] * 9 for _ in xrange(1 << 9)]
        for i in xrange(9):
            dp[merge(0, i)][i] = 1

        res = 0
        for used in xrange(len(dp)):
            number = number_of_keys(used)
            if number > n:
                continue

            for i in xrange(9):
                if not contain(used, i):
                    continue

                if m <= number <= n:
                    res += dp[used][i]

                x1, y1 = divmod(i, 3)
                for j in xrange(9):
                    if contain(used, j):
                        continue

                    x2, y2 = divmod(j, 3)
                    if ((x1 == x2 and abs(y1 - y2) == 2) or
                        (y1 == y2 and abs(x1 - x2) == 2) or
                        (abs(x1 - x2) == 2 and abs(y1 - y2) == 2)) and \
                       not contain(used,
```

```
                                    convert((x1 + x2) // 2, (y1 + y2) // 2)):
                            continue

                    dp[merge(used, j)][j] += dp[used][i]

        return res


# Time:  O(9^2 * 2^9)
# Space: O(9 * 2^9)
# DP solution.
class Solution2(object):
    def numberOfPatterns(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        def merge(used, i):
            return used | (1 << i)

        def number_of_keys(i):
            number = 0
            while i > 0:
                i &= i - 1
                number += 1
            return number

        def exclude(used, i):
            return used & ~(1 << i)

        def contain(used, i):
            return bool(used & (1 << i))

        def convert(i, j):
            return 3 * i + j

        # dp[i][j]: i is the set of the numbers in binary representation,
        #           d[i][j] is the number of ways ending with the number j.
        dp = [[0] * 9 for _ in xrange(1 << 9)]
        for i in xrange(9):
            dp[merge(0, i)][i] = 1

        res = 0
        for used in xrange(len(dp)):
            number = number_of_keys(used)
            if number > n:
                continue

            for i in xrange(9):
                if not contain(used, i):
                    continue

                x1, y1 = divmod(i, 3)
                for j in xrange(9):
                    if i == j or not contain(used, j):
                        continue

                    x2, y2 = divmod(j, 3)
                    if ((x1 == x2 and abs(y1 - y2) == 2) or
```

```
                        (y1 == y2 and abs(x1 - x2) == 2) or
                        (abs(x1 - x2) == 2 and abs(y1 - y2) == 2)) and \
                       not contain(used,
                                   convert((x1 + x2) // 2, (y1 + y2) // 2)):
                           continue

                    dp[used][i] += dp[exclude(used, i)][j]

                if m <= number <= n:
                    res += dp[used][i]

        return res


# Time:  O(9!)
# Space: O(9)
# Backtracking solution. (TLE)
class Solution_TLE(object):
    def numberOfPatterns(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        def merge(used, i):
            return used | (1 << i)

        def contain(used, i):
            return bool(used & (1 << i))

        def convert(i, j):
            return 3 * i + j

        def numberOfPatternsHelper(m, n, level, used, i):
            number = 0
            if level > n:
                return number

            if m <= level <= n:
                number += 1

            x1, y1 = divmod(i, 3)
            for j in xrange(9):
                if contain(used, j):
                    continue

                x2, y2 = divmod(j, 3)
                if ((x1 == x2 and abs(y1 - y2) == 2) or
                    (y1 == y2 and abs(x1 - x2) == 2) or
                    (abs(x1 - x2) == 2 and abs(y1 - y2) == 2)) and \
                   not contain(used,
                               convert((x1 + x2) // 2, (y1 + y2) // 2)):
                        continue

                number += numberOfPatternsHelper(m, n, level + 1, merge(used, j), j)

            return number

        number = 0
        # 1, 3, 7, 9
```

```
number += 4 * numberOfPatternsHelper(m, n, 1, merge(0, 0), 0)
# 2, 4, 6, 8
number += 4 * numberOfPatternsHelper(m, n, 1, merge(0, 1), 1)
# 5
number += numberOfPatternsHelper(m, n, 1, merge(0, 4), 4)
return number
```

binary-tree-maximum-path-sum.py

```
# DESC
# For this problem, a path is defined as any sequence of nodes from some starting
# node to any node in the tree along the parent-child connections. The path must c
# ontain at least one node and does not need to go through the root.
# Example 2:
# Example 1:
# Given a non-empty binary tree, find the maximum path sum.

# NOTE
#

# EXAMPLE
# Input: [1,2,3]
#
#        1
#       / \
#      2   3
#
# Output: 6
# Input: [-10,9,20,null,null,15,7]
#
#    -10
#    / \
#   9  20
#     /  \
#    15   7
#
# Outp
# ut: 42

# 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):
    maxSum = float("-inf")

    # @param root, a tree node
    # @return an integer
    def maxPathSum(self, root):
        self.maxPathSumRecu(root)
        return self.maxSum

    def maxPathSumRecu(self, root):
        if root is None:
            return 0
        left = max(0, self.maxPathSumRecu(root.left))
        right = max(0, self.maxPathSumRecu(root.right))
        self.maxSum = max(self.maxSum, root.val + left + right)
        return root.val + max(left, right)
```

nth-digit.py

```
# DESC
# Note:
#
# n is positive and will fit within the range of a 32-bit signed integer (n
#  < 231).
# Example 2:
# Example 1:
# Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 1
# 0, 11, ...

# NOTE
#

# EXAMPLE
# Input:
# 3
#
# Output:
# 3
# 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

```
# DESC
# There is a list of sorted integers from 1 to n. Starting from left to right, rem
# ove the first number and every other number afterward until you reach the end of
#  the list.
# 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.
# Repeat the previous step again, but this time from right to left, remove the rig
# ht most number and every other number from the remaining numbers.
# Example:

# NOTE
#

# 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
```

find-a-value-of-a-mysterious-function-closest-to-target.py

```python
# find-a-value-of-a-mysterious-function-closest-to-target is not found.
# Time:  O(nlogm), m is the max value of arr
# Space: O(logm)

class BitCount(object):
    def __init__(self, n):
        self.__l = 0
        self.__n = n
        self.__count = [0]*n

    def __iadd__(self, num):
        self.__l += 1
        base = 1
        for i in xrange(self.__n):
            if num&base:
                self.__count[i] += 1
            base <<= 1
        return self

    def __isub__(self, num):
        self.__l -= 1
        base = 1
        for i in xrange(self.__n):
            if num&base:
                self.__count[i] -= 1
            base <<= 1
        return self

    def bit_and(self):
        num, base = 0, 1
        for i in xrange(self.__n):
            if self.__count[i] == self.__l:
                num |= base
            base <<= 1
        return num


class Solution(object):
    def closestToTarget(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        count = BitCount(max(arr).bit_length())
        result, left = float("inf"), 0
        for right in xrange(len(arr)):
            count += arr[right]
            while left <= right:
                f = count.bit_and()
                result = min(result, abs(f-target))
                if f >= target:
                    break
                count -= arr[left]
                left += 1
        return result
```

```python
# Time:  O(nlogm), m is the max value of arr
# Space: O(logm)
class Solution2(object):
    def closestToTarget(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        result, dp = float("inf"), set()  # at most O(logm) dp states
        for x in arr:
            dp = {x}|{f&x for f in dp}
            for f in dp:
                result = min(result, abs(f-target))
        return result
```

move-zeroes.py

```
# DESC
# Given an array nums, write a function to move all 0's to the end of it while mai
# ntaining the relative order of the non-zero elements.
# Example:
# Note:

# NOTE
# Minimize the total number of operations.
# You must do this in-place without making a copy of the array.

# EXAMPLE
# Input: [0,1,0,3,12]
# Output: [1,3,12,0,0]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def moveZeroes(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        pos = 0
        for i in xrange(len(nums)):
            if nums[i]:
                nums[i], nums[pos] = nums[pos], nums[i]
                pos += 1

    def moveZeroes2(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        nums.sort(cmp=lambda a, b: 0 if b else -1)


class Solution2(object):
    def moveZeroes(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        pos = 0
        for i in xrange(len(nums)):
            if nums[i]:
                nums[pos] = nums[i]
                pos += 1

        for i in xrange(pos, len(nums)):
            nums[i] = 0
```

consecutive-characters.py

```
# consecutive-characters is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxPower(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, count = 1, 1
        for i in xrange(1, len(s)):
            if s[i] == s[i-1]:
                count += 1
            else:
                count = 1
            result = max(result, count)
        return result


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


class Solution2(object):
    def maxPower(self, s):
        return max(len(list(v)) for _, v in itertools.groupby(s))
```

valid-palindrome-iii.py

```python
# valid-palindrome-iii is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def isValidPalindrome(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: bool
        """
        if s == s[::-1]:  # optional, to optimize special case
            return True

        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 len(s) <= k + dp[0][-1]
```

serialize-and-deserialize-n-ary-tree.py

```
# serialize-and-deserialize-n-ary-tree is not found.
# Time:  O(n)
# Space: O(h)

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


class Codec(object):

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

        :type root: Node
        :rtype: str
        """
        def dfs(node, vals):
            if not node:
                return
            vals.append(str(node.val))
            for child in node.children:
                dfs(child, vals)
            vals.append("#")

        vals = []
        dfs(root, vals)
        return " ".join(vals)


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

        :type data: str
        :rtype: Node
        """
        def isplit(source, sep):
            sepsize = len(sep)
            start = 0
            while True:
                idx = source.find(sep, start)
                if idx == -1:
                    yield source[start:]
                    return
                yield source[start:idx]
                start = idx + sepsize

        def dfs(vals):
            val = next(vals)
            if val == "#":
                return None
            root = Node(int(val), [])
            child = dfs(vals)
            while child:
                root.children.append(child)
                child = dfs(vals)
            return root
```

```
if not data:
    return None

return dfs(iter(isplit(data, ' ')))
```

build-an-array-with-stack-operations.py

```python
# build-an-array-with-stack-operations is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def buildArray(self, target, n):
        """
        :type target: List[int]
        :type n: int
        :rtype: List[str]
        """
        result, curr = [], 1
        for t in target:
            result.extend(["Push", "Pop"]*(t-curr))
            result.append("Push")
            curr = t+1
        return result
```

missing-ranges.py

```
# missing-ranges is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMissingRanges(self, nums, lower, upper):
        """
        :type nums: List[int]
        :type lower: int
        :type upper: int
        :rtype: List[str]
        """
        def getRange(lower, upper):
            if lower == upper:
                return "{}".format(lower)
            else:
                return "{}->{}".format(lower, upper)
        ranges = []
        pre = lower - 1

        for i in xrange(len(nums) + 1):
            if i == len(nums):
                cur = upper + 1
            else:
                cur = nums[i]
            if cur - pre >= 2:
                ranges.append(getRange(pre + 1, cur - 1))

            pre = cur

        return ranges
```

get-the-maximum-score.py

```python
# get-the-maximum-score is not found.
# Time:  O(m + n)
# Space: O(1)

class Solution(object):
    def maxSum(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: int
        """
        MOD = 10**9+7
        i, j = 0, 0
        result, sum1, sum2 = 0, 0, 0,
        while i != len(nums1) or j != len(nums2):
            if i != len(nums1) and (j == len(nums2) or nums1[i] < nums2[j]):
                sum1 += nums1[i]
                i += 1
            elif j != len(nums2) and (i == len(nums1) or nums1[i] > nums2[j]):
                sum2 += nums2[j]
                j += 1
            else:
                result = (result + (max(sum1, sum2) + nums1[i])) % MOD
                sum1, sum2 = 0, 0
                i += 1
                j += 1
        return (result + max(sum1, sum2)) % MOD
```

design-log-storage-system.py

```
# design-log-storage-system is not found.
# Time:  put:      O(1)
#        retrieve: O(n + dlogd), n is the size of the total logs
#                              , d is the size of the found logs
# Space: O(n)

class LogSystem(object):

    def __init__(self):
        self.__logs = []
        self.__granularity = {'Year': 4, 'Month': 7, 'Day': 10, \
                              'Hour': 13, 'Minute': 16, 'Second': 19}


    def put(self, id, timestamp):
        """
        :type id: int
        :type timestamp: str
        :rtype: void
        """
        self.__logs.append((id, timestamp))


    def retrieve(self, s, e, gra):
        """
        :type s: str
        :type e: str
        :type gra: str
        :rtype: List[int]
        """
        i = self.__granularity[gra]
        begin = s[:i]
        end = e[:i]
        return sorted(id for id, timestamp in self.__logs \
                      if begin <= timestamp[:i] <= end)
```

can-make-palindrome-from-substring.py

```python
# can-make-palindrome-from-substring is not found.
# Time:  O(m + n), m is the number of queries, n is the length of s
# 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[:])
        return [sum((b-a)%2 for a, b in itertools.izip(count[left], count[right+1]))//2 <= k
                for left, right, k in queries]
```

min-cost-climbing-stairs.py

```
# DESC
# Note:
# On a staircase, the i-th step has some non-negative cost cost[i] assigned (0 ind
# exed).
# Once you pay the cost, you can either climb one or two steps. You need to find m
# inimum cost to reach the top of the floor, and you can either start from the ste
# p with index 0, or the step with index 1.
# Example 2:
# Example 1:

# NOTE
# cost will have a length in the range [2, 1000].
# Every cost[i] will be an integer in the range [0, 999].

# EXAMPLE
# Input: cost = [10, 15, 20]
# Output: 15
# Explanation: Cheapest is start on cost[1],
#  pay that cost and go to the top.
# Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
# Output: 6
# Explanation: Cheape
# st is start on cost[0], and only step on 1s, skipping cost[3].

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minCostClimbingStairs(self, cost):
        """
        :type cost: List[int]
        :rtype: int
        """
        dp = [0] * 3
        for i in reversed(xrange(len(cost))):
            dp[i%3] = cost[i] + min(dp[(i+1)%3], dp[(i+2)%3])
        return min(dp[0], dp[1])
```

reverse-subarray-to-maximize-array-value.py

```python
# reverse-subarray-to-maximize-array-value is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxValueAfterReverse(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, add, max_pair, min_pair = 0, 0, float("-inf"), float("inf")
        for i in xrange(1, len(nums)):
            result += abs(nums[i-1]-nums[i])
            add = max(add,
                      abs(nums[0]-nums[i]) - abs(nums[i-1]-nums[i]),
                      abs(nums[-1]-nums[i-1]) - abs(nums[i-1]-nums[i]))
            min_pair = min(min_pair, max(nums[i-1], nums[i]))
            max_pair = max(max_pair, min(nums[i-1], nums[i]))
        return result + max(add, (max_pair-min_pair)*2)
```

campus-bikes-ii.py

```
# campus-bikes-ii is not found.
# Time:  O(w * b * 2^b)
# Space: O(b * 2^b)

# if w = b, we can even apply Hungarian algorithm (see https://en.wikipedia.org/wiki/Hungarian_algorithm),
# it can be improved to O(w^3), see https://github.com/t3nsor/codebook/blob/master/bipartite-mincost.cpp
class Solution(object):  # this is slower than Solution2 in python
    def assignBikes(self, workers, bikes):
        """
        :type workers: List[List[int]]
        :type bikes: List[List[int]]
        :rtype: int
        """
        def manhattan(p1, p2):
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        dp = [[float("inf")]*((1<<len(bikes))) for _ in xrange(2)]
        dp[0][0] = 0
        for i in xrange(len(workers)):
            dp[(i+1)%2] = [float("inf")] * ((1<<len(bikes)))
            for j in xrange(len(bikes)):
                for taken in xrange((1<<len(bikes))):
                    if taken & (1<<j):
                        continue
                    dp[(i+1)%2][taken|(1<<j)] = \
                        min(dp[(i+1)%2][taken|(1<<j)],
                            dp[i%2][taken] +
                            manhattan(workers[i], bikes[j]))
        return min(dp[len(workers)%2])


# Time:  O((w * b * 2^b) * log(w * b * 2^b))
# Space: O(w * b * 2^b)
import heapq


class Solution2(object):
    def assignBikes(self, workers, bikes):
        """
        :type workers: List[List[int]]
        :type bikes: List[List[int]]
        :rtype: int
        """
        def manhattan(p1, p2):
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        min_heap = [(0, 0, 0)]
        lookup = set()
        while min_heap:
            cost, i, taken = heapq.heappop(min_heap)
            if (i, taken) in lookup:
                continue
            lookup.add((i, taken))
            if i == len(workers):
                return cost
            for j in xrange(len(bikes)):
                if taken & (1<<j):
                    continue
```

```
                heapq.heappush(min_heap, (cost+manhattan(workers[i], bikes[j]),  # O(b)
                                          i+1,            # O(w)
                                          taken|(1<<j)))  # O(2^b)
```

max-increase-to-keep-city-skyline.py

```
# DESC
# What is the maximum total sum that the height of the buildings can be increased?
# In a 2 dimensional array grid, each value grid[i][j] represents the height of a
# building located there. We are allowed to increase the height of any number of b
# uildings, by any amount (the amounts can be different for different buildings).
# Height 0 is considered to be a building as well.
# Notes:
# At the end, the "skyline" when viewed from all four directions of the grid, i.e.
#  top, bottom, left, and right, must be the same as the skyline of the original g
# rid. A city's skyline is the outer contour of the rectangles formed by all the b
# uildings when viewed from a distance. See the following example.

# NOTE
# All buildings in grid[i][j] occupy the entire grid cell: that is, they are a 1 x
#  1 x grid[i][j] rectangular prism.
# All heights grid[i][j] are in the range [0, 100].
# 1 < grid.length = grid[0].length <= 50.

# EXAMPLE
# Example:
# Input: grid = [[3,0,8,4],[2,4,5,7],[9,2,6,3],[0,3,1,0]]
# Output: 35
# Expl
# anation:
# The grid is:
# [ [3, 0, 8, 4],
#   [2, 4, 5, 7],
#   [9, 2, 6, 3],
#   [0, 3,
#  1, 0] ]
#
# The skyline viewed from top or bottom is: [9, 4, 8, 7]
# The skyline vie
# wed from left or right is: [8, 7, 9, 3]
#
# The grid after increasing the height of
#  buildings without affecting skylines is:
#
# gridNew = [ [8, 4, 8, 7],
#
#  [7, 4, 7, 7],
#             [9, 4, 8, 7],
#             [3, 3, 3, 3] ]

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

import itertools


class Solution(object):
    def maxIncreaseKeepingSkyline(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        row_maxes = [max(row) for row in grid]
        col_maxes = [max(col) for col in itertools.izip(*grid)]
```

```
    return sum(min(row_maxes[r], col_maxes[c])-val \
               for r, row in enumerate(grid) \
               for c, val in enumerate(row))
```

number-of-good-leaf-nodes-pairs.py

```python
# number-of-good-leaf-nodes-pairs is not found.
# Time:  O(n)
# Space: O(h)

import collections


# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution(object):
    def countPairs(self, root, distance):
        """
        :type root: TreeNode
        :type distance: int
        :rtype: int
        """
        def iter_dfs(distance, root):
            result = 0
            stk = [(1, (root, [collections.Counter()]))]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node, ret = params
                    if not node:
                        continue
                    if not node.left and not node.right:
                        ret[0][0] = 1
                        continue
                    left, right = [collections.Counter()], [collections.Counter()]
                    stk.append((2, (left, right, ret)))
                    stk.append((1, (node.right, right)))
                    stk.append((1, (node.left, left)))
                else:
                    left, right, ret = params
                    for left_d, left_c in left[0].iteritems():
                        for right_d,right_c in right[0].iteritems():
                            if left_d+right_d+2 <= distance:
                                result += left_c*right_c
                    ret[0] = collections.Counter({k+1:v for k,v in (left[0]+right[0]).iteritems()})
            return result

        return iter_dfs(distance, root)


# Time:  O(n)
# Space: O(h)
import collections


class Solution2(object):
    def countPairs(self, root, distance):
        """
```

```
        :type root: TreeNode
        :type distance: int
        :rtype: int
        """
        def dfs(distance, node):
            if not node:
                return 0, collections.Counter()
            if not node.left and not node.right:
                return 0, collections.Counter([0])
            left, right = dfs(distance, node.left), dfs(distance, node.right)
            result = left[0]+right[0]
            for left_d, left_c in left[1].iteritems():
                for right_d,right_c in right[1].iteritems():
                    if left_d+right_d+2 <= distance:
                        result += left_c*right_c
            return result, collections.Counter({k+1:v for k,v in (left[1]+right[1]).iteritems()})

        return dfs(distance, root)[0]
```

design-twitter.py

```
# DESC
# Example:
# Design a simplified version of Twitter where users can post tweets, follow/unfol
# low 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:

# NOTE
# follow(followerId, followeeId): Follower follows a followee.
# postTweet(userId, tweetId): Compose a new tweet.
# unfollow(followerId, followeeId): Follower unfollows a followee.
# getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's news fe
# ed. 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.

# 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 prece
# de 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 use
# r 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

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)
```

maximum-product-of-three-numbers.py

```
# DESC
# Example 2:
# Example 1:
# Note:
# Given an integer array, find three numbers whose product is maximum and output t
# he maximum product.

# NOTE
# Multiplication of any three numbers in the input won't exceed the range of 32-bi
# t signed integer.
# The length of the given array will be in range [3,104] and all elements are in t
# he range [-1000, 1000].

# EXAMPLE
# Input: [1,2,3,4]
# Output: 24
# Input: [1,2,3]
# Output: 6

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maximumProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        min1, min2 = float("inf"), float("inf")
        max1, max2, max3 = float("-inf"), float("-inf"), float("-inf")

        for n in nums:
            if n <= min1:
                min2 = min1
                min1 = n
            elif n <= min2:
                min2 = n

            if n >= max1:
                max3 = max2
                max2 = max1
                max1 = n
            elif n >= max2:
                max3 = max2
                max2 = n
            elif n >= max3:
                max3 = n

        return max(min1 * min2 * max1, max1 * max2 * max3)
```

maximum-average-subarray-ii.py

```python
# maximum-average-subarray-ii is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def findMaxAverage(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: float
        """
        def getDelta(avg, nums, k):
            accu = [0.0] * (len(nums) + 1)
            minval_pos = None
            delta = 0.0
            for i in xrange(len(nums)):
                accu[i+1] = nums[i] + accu[i] - avg
                if i >= (k-1):
                    if minval_pos == None or accu[i-k+1] < accu[minval_pos]:
                        minval_pos = i-k+1
                    if accu[i+1] - accu[minval_pos] >= 0:
                        delta = max(delta, (accu[i+1] - accu[minval_pos]) / (i+1 - minval_pos))
            return delta

        left, delta = min(nums), float("inf")
        while delta > 1e-5:
            delta = getDelta(left, nums, k)
            left += delta
        return left
```

edit-distance.py

```python
# DESC
# You have the following 3 operations permitted on a word:
# Example 1:
# Example 2:
# Given two words word1 and word2, find the minimum number of operations required
# to convert word1 to word2.

# NOTE
# Insert a character
# Delete a character
# Replace a character

# EXAMPLE
# Input: word1 = "intention", word2 = "execution"
# Output: 5
# Explanation:
# intentio
# n -> inention (remove 't')
# inention -> enention (replace 'i' with 'e')
# enention
# -> exention (replace 'n' with 'x')
# exention -> exection (replace 'n' with 'c')
# e
# xection -> execution (insert 'u')
# Input: word1 = "horse", word2 = "ros"
# Output: 3
# Explanation:
# horse -> rorse (re
# place 'h' with 'r')
# rorse -> rose (remove 'r')
# rose -> ros (remove 'e')

# Time:  O(n * m)
# Space: O(n + m)

class Solution(object):
    # @return an integer
    def minDistance(self, word1, word2):
        if len(word1) < len(word2):
            return self.minDistance(word2, word1)

        distance = [i for i in xrange(len(word2) + 1)]

        for i in xrange(1, len(word1) + 1):
            pre_distance_i_j = distance[0]
            distance[0] = i
            for j in xrange(1, len(word2) + 1):
                insert = distance[j - 1] + 1
                delete = distance[j] + 1
                replace = pre_distance_i_j
                if word1[i - 1] != word2[j - 1]:
                    replace += 1
                pre_distance_i_j = distance[j]
                distance[j] = min(insert, delete, replace)

        return distance[-1]

# Time:  O(n * m)
```

```
# Space: O(n * m)
class Solution2(object):
    # @return an integer
    def minDistance(self, word1, word2):
        distance = [[i] for i in xrange(len(word1) + 1)]
        distance[0] = [j for j in xrange(len(word2) + 1)]

        for i in xrange(1, len(word1) + 1):
            for j in xrange(1, len(word2) + 1):
                insert = distance[i][j - 1] + 1
                delete = distance[i - 1][j] + 1
                replace = distance[i - 1][j - 1]
                if word1[i - 1] != word2[j - 1]:
                    replace += 1
                distance[i].append(min(insert, delete, replace))

        return distance[-1][-1]
```

contains-duplicate-iii.py

```
# DESC
# Given an array of integers, find out whether there are two distinct indices i an
# d j in the array such that the absolute difference between nums[i] and nums[j] i
# s at most t and the absolute difference between i and j is at most k.
# Example 2:
# Example 1:
# Example 3:

# NOTE
#

# EXAMPLE
# Input: nums = [1,2,3,1], k = 3, t = 0
# Output: true
# Input: nums = [1,0,1,1], k = 1, t = 2
# Output: true
# Input: nums = [1,5,9,1,5,9], k = 2, t = 3
# Output: false

# Time:  O(n * t)
# Space: O(max(k, t))

import collections


class Solution(object):
    # @param {integer[]} nums
    # @param {integer} k
    # @param {integer} t
    # @return {boolean}
    def containsNearbyAlmostDuplicate(self, nums, k, t):
        if k < 0 or t < 0:
            return False
        window = collections.OrderedDict()
        for n in nums:
            # Make sure window size
            if len(window) > k:
                window.popitem(False)

            bucket = n if not t else n // t
            # At most 2t items.
            for m in (window.get(bucket - 1), window.get(bucket), window.get(bucket + 1)):
                if m is not None and abs(n - m) <= t:
                    return True
            window[bucket] = n
        return False
```

palindrome-partitioning.py

```
# DESC
# Given a string s, partition s such that every substring of the partition is a pa
# lindrome.
# Example:
# Return all possible palindrome partitioning of s.

# NOTE
#

# 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()

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

k-th-smallest-prime-fraction.py

```
# DESC
# A sorted list A contains 1, plus some number of primes.  Then, for every p < q i
# n the list, we consider the fraction p/q.
# What is the K-th smallest fraction considered?  Return your answer as an array o
# f ints, where answer[0] = p and answer[1] = q.
# Note:

# NOTE
# Each A[i] will be between 1 and 30000.
# K will be between 1 and A.length * (A.length - 1) / 2.
# A will have length between 2 and 2000.

# EXAMPLE
# Examples:
# Input: A = [1, 2, 3, 5], K = 3
# Output: [2, 5]
# Explanation:
# The fractio
# ns to be considered in sorted order are:
# 1/5, 1/3, 2/5, 1/2, 3/5, 2/3.
# The third
#  fraction is 2/5.
#
# Input: A = [1, 7], K = 1
# Output: [1, 7]

# Time:  O(nlogr)
# Space: O(1)

class Solution(object):
    def kthSmallestPrimeFraction(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: List[int]
        """
        def check(mid, A, K, result):
            tmp = [0]*2
            count = 0
            j = 0
            for i in xrange(len(A)):
                while j < len(A):
                    if i < j and A[i] < A[j]*mid:
                        if tmp[0] == 0 or \
                           tmp[0]*A[j] < tmp[1]*A[i]:
                            tmp[0] = A[i]
                            tmp[1] = A[j]
                        break
                    j += 1
                count += len(A)-j
            if count == K:
                result[:] = tmp
            return count >= K

        result = []
        left, right = 0.0, 1.0
        while right-left > 1e-8:
            mid = left + (right-left) / 2.0
```

```
        if check(mid, A, K, result):
            right = mid
        else:
            left = mid
        if result:
            break
    return result
```

avoid-flood-in-the-city.py

```python
# avoid-flood-in-the-cit is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections
import heapq


class Solution(object):
    def avoidFlood(self, rains):
        """
        :type rains: List[int]
        :rtype: List[int]
        """
        lookup = collections.defaultdict(list)
        i = len(rains)-1
        for lake in reversed(rains):
            lookup[lake].append(i)
            i -= 1
        result, min_heap = [], []
        for i, lake in enumerate(rains):
            if lake:
                if len(lookup[lake]) >= 2:
                    lookup[lake].pop()
                    heapq.heappush(min_heap, lookup[lake][-1])
                result.append(-1)
            elif min_heap:
                j = heapq.heappop(min_heap)
                if j < i:
                    return []
                result.append(rains[j])
            else:
                result.append(1)
        return result if not min_heap else []
```

majority-element.py

```
# DESC
# Example 1:
# You may assume that the array is non-empty and the majority element always exist
#  in the array.
# Given an array of size n, find the majority element. The majority element is the
#  element that appears more than  n/2  times.
# Example 2:

# NOTE
#

# EXAMPLE
# Input: [2,2,1,1,1,2,2]
# Output: 2
# Input: [3,2,3]
# Output: 3

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def majorityElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        idx, cnt = 0, 1

        for i in xrange(1, len(nums)):
            if nums[idx] == nums[i]:
                cnt += 1
            else:
                cnt -= 1
                if cnt == 0:
                    idx = i
                    cnt = 1

        return nums[idx]

    def majorityElement2(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return sorted(collections.Counter(nums).items(), key=lambda a: a[1], reverse=True)[0][0]

    def majorityElement3(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return collections.Counter(nums).most_common(1)[0][0]
```

minimum-swaps-to-arrange-a-binary-grid.py

```
# minimum-swaps-to-arrange-a-binary-grid is not found.
# Time:  O(n^2)
# Space: O(1)

import itertools


class Solution(object):
    def minSwaps(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        result = 0
        for target in reversed(xrange(1, len(grid))):
            row_idx = len(grid)-1-target
            while row_idx < len(grid):
                row = grid[row_idx]
                if not sum(itertools.islice(row, len(row)-target, len(row))):
                    break
                row_idx += 1
            else:
                return -1
            while row_idx != len(grid)-1-target:
                grid[row_idx], grid[row_idx-1] = grid[row_idx-1], grid[row_idx]
                result += 1
                row_idx -= 1
        return result
```

rectangle-area.py

```
# DESC
# Example:
# Each rectangle is defined by its bottom left corner and top right corner as show
# n in the figure.
# Note:
# Assume that the total area is never beyond the maximum possible value of int.
# Find the total area covered by two rectilinear rectangles in a 2D plane.

# NOTE
#

# EXAMPLE
# Input: A = -3, B = 0, C = 3, D = 4, E = 0, F = -1, G = 9, H = 2
# Output: 45

# 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)))
```

cells-with-odd-values-in-a-matrix.py

```
# cells-with-odd-values-in-a-matrix is not found.
# Time:  O(n + m)
# Space: O(n + m)

class Solution(object):
    def oddCells(self, n, m, indices):
        """
        :type n: int
        :type m: int
        :type indices: List[List[int]]
        :rtype: int
        """
        row, col = [0]*n, [0]*m
        for r, c in indices:
            row[r] ^= 1
            col[c] ^= 1
        row_sum, col_sum = sum(row), sum(col)
        return row_sum*m+col_sum*n-2*row_sum*col_sum


# Time:  O(n + m)
# Space: O(n + m)
import collections
import itertools


class Solution2(object):
    def oddCells(self, n, m, indices):
        """
        :type n: int
        :type m: int
        :type indices: List[List[int]]
        :rtype: int
        """
        fn = lambda x: sum(count&1 for count in collections.Counter(x).itervalues())
        row_sum, col_sum = map(fn, itertools.izip(*indices))
        return row_sum*m+col_sum*n-2*row_sum*col_sum
```

course-schedule-ii.py

```
# DESC
# Example 1:
# Given the total number of courses and a list of prerequisite pairs, return the o
# rdering of courses you should take to finish all courses.
# [0,1]
# Note:
# There are a total of n courses you have to take, labeled from 0 to n-1.
# There may be multiple correct orders, you just need to return one of them. If it
#  is impossible to finish all courses, return an empty array.
# Some courses may have prerequisites, for example to take course 0 you have to fi
# rst take course 1, which is expressed as a pair: [0,1]
# Example 2:

# NOTE
# You may assume that there are no duplicate edges in the input prerequisites.
# The input prerequisites is a graph represented by a list of edges, not adjacency
#  matrices. Read more about how a graph is represented.

# EXAMPLE
# Input: 4, [[1,0],[2,0],[3,1],[3,2]]
# Output: [0,1,2,3] or [0,2,1,3]
# Explanation:
# There are a total of 4 courses to take. To take course 3 you should have finishe
# d both
#              courses 1 and 2. Both courses 1 and 2 should be taken a
# fter you finished course 0.
#              So one correct course order is [0,1,2,
# 3]. Another correct ordering is [0,2,1,3] .
# Input: 2, [[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] .

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)

        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
```

degree-of-an-array.py

```
# degree-of-an-arra is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findShortestSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        counts = collections.Counter(nums)
        left, right = {}, {}
        for i, num in enumerate(nums):
            left.setdefault(num, i)
            right[num] = i
        degree = max(counts.values())
        return min(right[num]-left[num]+1 \
                   for num in counts.keys() \
                   if counts[num] == degree)
```

binary-tree-right-side-view.py

```
# DESC
# Example:
# Given a binary tree, imagine yourself standing on the right side of it, return t
# he values of the nodes you can see ordered from top to bottom.

# NOTE
#

# EXAMPLE
# Input: [1,2,3,null,5,null,4]
# Output: [1, 3, 4]
# Explanation:
#
#    1            <--
# -
#  /   \
# 2     3         <---
#  \     \
#   5     4       <---

# 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):
    # @param root, a tree node
    # @return a list of integers
    def rightSideView(self, root):
        result = []
        self.rightSideViewDFS(root, 1, result)
        return result

    def rightSideViewDFS(self, node, depth, result):
        if not node:
            return

        if depth > len(result):
            result.append(node.val)

        self.rightSideViewDFS(node.right, depth+1, result)
        self.rightSideViewDFS(node.left, depth+1, result)


# BFS solution
# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param root, a tree node
    # @return a list of integers
    def rightSideView(self, root):
        if root is None:
            return []
```

```
result, current = [], [root]
while current:
    next_level = []
    for node in current:
        if node.left:
            next_level.append(node.left)
        if node.right:
            next_level.append(node.right)
    result.append(node.val)
    current = next_level

return result
```

car-pooling.py

```
# DESC
# Constraints:
# Example 2:
# Return true if and only if it is possible to pick up and drop off all passengers
#  for all the given trips.
# You are driving a vehicle that has capacity empty seats initially available for
# passengers.  The vehicle only drives east (ie. it cannot turn around and drive w
# est.)
# Example 4:
# Example 1:
# Example 3:
# Given a list of trips, trip[i] = [num_passengers, start_location, end_location]
# contains information about the i-th trip: the number of passengers that must be
# picked up, and the locations to pick them up and drop them off.  The locations a
# re given as the number of kilometers due east from your vehicle's initial locati
# on.
# trips

# NOTE
# 1 <= capacity <= 100000
# trips.length <= 1000
# 0 <= trips[i][1] < trips[i][2] <= 1000
# 1 <= trips[i][0] <= 100
# trips[i].length == 3

# EXAMPLE
# Input: trips = [[2,1,5],[3,3,7]], capacity = 5
# Output: true
# Input: trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11
# Output: true
# Input: trips = [[2,1,5],[3,5,7]], capacity = 3
# Output: true
# Input: trips = [[2,1,5],[3,3,7]], capacity = 4
# Output: false

# 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
```

erect-the-fence.py

```
# DESC
# Example 1:
# Note:
# Example 2:
# There are some trees, where each tree is represented by (x,y) coordinate in a tw
# o-dimensional garden. Your job is to fence the entire garden using the minimum l
# ength of rope as it is expensive. The garden is well fenced only if all the tree
# s are enclosed. Your task is to help find the coordinates of trees which are exa
# ctly located on the fence perimeter.

# NOTE
# input types have been changed on April 15, 2019. Please reset to default code de
# finition to get new method signature.
# All coordinates are distinct.
# The garden has at least one tree.
# Input points have NO order. No order required for output.
# All input integers will range from 0 to 100.
# All trees should be enclosed together. You cannot cut the rope to enclose trees
# that will separate them in more than one group.

# EXAMPLE
# Input: [[1,1],[2,2],[2,0],[2,4],[3,3],[4,2]]
# Output: [[1,1],[2,0],[4,2],[3,3],[2
# ,4]]
# Explanation:
# Input: [[1,2],[2,2],[4,2]]
# Output: [[1,2],[2,2],[4,2]]
# Explanation:
#
# Even you on
# ly have trees in a line, you need to use rope to enclose them.

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

import itertools


# Monotone Chain Algorithm
class Solution(object):
    def outerTrees(self, points):
        """
        :type points: List[List[int]]
        :rtype: List[List[int]]
        """
        def ccw(A, B, C):
            return (B[0]-A[0])*(C[1]-A[1]) - (B[1]-A[1])*(C[0]-A[0])

        if len(points) <= 1:
           return points

        hull = []
        points.sort()
        for i in itertools.chain(xrange(len(points)), reversed(xrange(len(points)-1))):
            while len(hull) >= 2 and ccw(hull[-2], hull[-1], points[i]) < 0:
                hull.pop()
            hull.append(points[i])
        hull.pop()
```

```
for i in xrange(1, (len(hull)+1)//2):
    if hull[i] != hull[-1]:
        break
    hull.pop()
return hull
```

find-pivot-index.py

```
# DESC
# Given an array of integers nums, write a method that returns the "pivot" index o
# f this array.
# If no such index exists, we should return -1. If there are multiple pivot indexe
# s, you should return the left-most pivot index.
# We define the pivot index as the index where the sum of all the numbers to the l
# eft of the index is equal to the sum of all the numbers to the right of the inde
# x.
# Example 1:
# Constraints:
# Example 2:

# NOTE
# Each element nums[i] will be an integer in the range [-1000, 1000].
# The length of nums will be in the range [0, 10000].

# EXAMPLE
# Input: nums = [1,7,3,6,5,6]
# Output: 3
# Explanation:
# The sum of the numbers to the
#  left of index 3 (nums[3] = 6) is equal to the sum of numbers to the right of in
# dex 3.
# Also, 3 is the first index where this occurs.
# Input: nums = [1,2,3]
# Output: -1
# Explanation:
# There is no index that satisfies t
# he conditions in the problem statement.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def pivotIndex(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        total = sum(nums)
        left_sum = 0
        for i, num in enumerate(nums):
            if left_sum == (total-left_sum-num):
                return i
            left_sum += num
        return -1
```

shortest-subarray-with-sum-at-least-k.py

```
# DESC
# If there is no non-empty subarray with sum at least K, return -1.
# Example 3:
# Example 2:
# Note:
# Example 1:
# Return the length of the shortest, non-empty, contiguous subarray of A with sum
# at least K.

# NOTE
# 1 <= K <= 10 ^ 9
# -10 ^ 5 <= A[i] <= 10 ^ 5
# 1 <= A.length <= 50000

# EXAMPLE
# Input: A = [1], K = 1
# Output: 1
# Input: A = [2,-1,2], K = 3
# Output: 3
# Input: A = [1,2], K = 4
# Output: -1

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

import collections


class Solution(object):
    def shortestSubarray(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        accumulated_sum = [0]*(len(A)+1)
        for i in xrange(len(A)):
            accumulated_sum[i+1] = accumulated_sum[i]+A[i]

        result = float("inf")
        mono_increasing_q = collections.deque()
        for i, curr in enumerate(accumulated_sum):
            while mono_increasing_q and curr <= \
                    accumulated_sum[mono_increasing_q[-1]]:
                mono_increasing_q.pop()
            while mono_increasing_q and \
                    curr-accumulated_sum[mono_increasing_q[0]] >= K:
                result = min(result, i-mono_increasing_q.popleft())
            mono_increasing_q.append(i)
        return result if result != float("inf") else -1
```

create-maximum-number.py

```
# DESC
# Example 2:
# Example 3:
# Example 1:
# Note: You should try to optimize your time and space complexity.
# Given two arrays of length m and n with digits 0-9 representing two numbers. Cre
# ate the maximum number of length k <= m + n from digits of the two. The relative
#  order of the digits from the same array must be preserved. Return an array of t
# he k digits.

# NOTE
#

# EXAMPLE
# Input:
# nums1 = [6, 7]
# nums2 = [6, 0, 4]
# k = 5
# Output:
# [6, 7, 6, 0, 4]
# Input:
# nums1 = [3, 4, 6, 5]
# nums2 = [9, 1, 2, 5, 8, 3]
# k = 5
# Output:
# [9, 8, 6, 5, 3]
# Input:
# nums1 = [3, 9]
# nums2 = [8, 9]
# k = 3
# Output:
# [9, 8, 9]

# Time:  O(k * (m + n + k)) ~ O(k * (m + n + k^2))
# Space: O(m + n + k^2)

class Solution(object):
    def maxNumber(self, nums1, nums2, k):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :type k: int
        :rtype: List[int]
        """
        def get_max_digits(nums, start, end, max_digits):
            max_digits[end] = max_digit(nums, end)
            for i in reversed(xrange(start, end)):
                max_digits[i] = delete_digit(max_digits[i + 1])

        def max_digit(nums, k):
            drop = len(nums) - k
            res = []
            for num in nums:
                while drop and res and res[-1] < num:
                    res.pop()
                    drop -= 1
                res.append(num)
            return res[:k]
```

```
def delete_digit(nums):
    res = list(nums)
    for i in xrange(len(res)):
        if i == len(res) - 1 or res[i] < res[i + 1]:
            res = res[:i] + res[i+1:]
            break
    return res

def merge(a, b):
    return [max(a, b).pop(0) for _ in xrange(len(a)+len(b))]

m, n = len(nums1), len(nums2)

max_digits1, max_digits2 = [[] for _ in xrange(k + 1)], [[] for _ in xrange(k + 1)]
get_max_digits(nums1, max(0, k - n), min(k, m), max_digits1)
get_max_digits(nums2, max(0, k - m), min(k, n), max_digits2)

return max(merge(max_digits1[i], max_digits2[k-i]) \
           for i in xrange(max(0, k - n), min(k, m) + 1))
```

design-bounded-blocking-queue.py

```python
# design-bounded-blocking-queue is not found.
# Time:  O(n)
# Space: O(1)

import threading
import collections


class BoundedBlockingQueue(object):
    def __init__(self, capacity):
        """
        :type capacity: int
        """
        self.__cv = threading.Condition()
        self.__q = collections.deque()
        self.__cap = capacity

    def enqueue(self, element):
        """
        :type element: int
        :rtype: void
        """
        with self.__cv:
            while len(self.__q) == self.__cap:
                self.__cv.wait()
            self.__q.append(element)
            self.__cv.notifyAll()

    def dequeue(self):
        """
        :rtype: int
        """
        with self.__cv:
            while not self.__q:
                self.__cv.wait()
            self.__cv.notifyAll()
            return self.__q.popleft()

    def size(self):
        """
        :rtype: int
        """
        with self.__cv:
            return len(self.__q)
```

array-partition-i.py

```
# DESC
# Given an array of 2n integers, your task is to group these integers into n pairs
#  of integer, say (a1, b1), (a2, b2), ..., (an, bn) which makes sum of min(ai, bi
# ) for all i from 1 to n as large as possible.
# Example 1:
# Note:

# NOTE
# n is a positive integer, which is in the range of [1, 10000].
# All the integers in the array will be in the range of [-10000, 10000].

# EXAMPLE
# Input: [1,4,3,2]
#
# Output: 4
# Explanation: n is 2, and the maximum sum of pairs is
#  4 = min(1, 2) + min(3, 4).

# Time:  O(r), r is the range size of the integers
# Space: O(r)


class Solution(object):
    def arrayPairSum(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        LEFT, RIGHT = -10000, 10000
        lookup = [0] * (RIGHT-LEFT+1)
        for num in nums:
            lookup[num-LEFT] += 1
        r, result = 0, 0
        for i in xrange(LEFT, RIGHT+1):
            result += (lookup[i-LEFT] + 1 - r) / 2 * i
            r = (lookup[i-LEFT] + r) % 2
        return result


# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def arrayPairSum(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        nums.sort()
        result = 0
        for i in xrange(0, len(nums), 2):
            result += nums[i]
        return result


# Time:  O(nlogn)
# Space: O(n)
class Solution3(object):
    def arrayPairSum(self, nums):
```

```
        """
        :type nums: List[int]
        :rtype: int
        """
        nums = sorted(nums)
        return sum([nums[i] for i in range(0, len(nums), 2)])
```

traffic-light-controlled-intersection.py

```python
# traffic-light-controlled-intersection is not found.
# Time:  O(n)
# Space: O(1)

import threading


class TrafficLight(object):

    def __init__(self):
        self.__l = threading.Lock()
        self.__light = 1

    def carArrived(self, carId, roadId, direction, turnGreen, crossCar):
        """
        :type roadId: int --> // ID of the car
        :type carId: int --> // ID of the road the car travels on. Can be 1 (road A) or 2 (road B)
        :type direction: int --> // Direction of the car
        :type turnGreen: method --> // Use turnGreen() to turn light to green on current road
        :type crossCar: method --> // Use crossCar() to make car cross the intersection
        :rtype: void
        """
        with self.__l:
            if self.__light != roadId:
                self.__light = roadId
                turnGreen()
            crossCar()
```

find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance.py

```python
# find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance is not found.
# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def findTheCity(self, n, edges, distanceThreshold):
        """
        :type n: int
        :type edges: List[List[int]]
        :type distanceThreshold: int
        :rtype: int
        """
        dist = [[float("inf")]*n for _ in xrange(n)]
        for i, j, w in edges:
            dist[i][j] = dist[j][i] = w
        for i in xrange(n):
            dist[i][i] = 0
        for k in xrange(n):
            for i in xrange(n):
                for j in xrange(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j])
        result = {sum(d <= distanceThreshold for d in dist[i]): i for i in xrange(n)}
        return result[min(result.iterkeys())]
```

reverse-pairs.py

```
# DESC
# Example1:
# Given an array nums, we call (i, j) an important reverse pair if i < j and nums[
# i] > 2*nums[j].
# You need to return the number of important reverse pairs in the given array.
# Example2:
# Note:

# NOTE
# All the numbers in the input array are in the range of 32-bit integer.
# The length of the given array will not exceed 50,000.

# EXAMPLE
# Input: [2,4,3,5,1]
# Output: 3
# Input: [1,3,2,3,1]
# Output: 2

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

class Solution(object):
    def reversePairs(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def merge(nums, start, mid, end):
            r = mid + 1
            tmp = []
            for i in xrange(start, mid + 1):
                while r <= end and nums[i] > nums[r]:
                    tmp.append(nums[r])
                    r += 1
                tmp.append(nums[i])
            nums[start:start+len(tmp)] = tmp

        def countAndMergeSort(nums, start, end):
            if end - start <= 0:
                return 0

            mid = start + (end - start) / 2
            count = countAndMergeSort(nums, start, mid) + countAndMergeSort(nums, mid + 1, end)
            r = mid + 1
            for i in xrange(start, mid + 1):
                while r <= end and nums[i] > nums[r] * 2:
                    r += 1
                count += r - (mid + 1)
            merge(nums, start, mid, end)
            return count

        return countAndMergeSort(nums, 0, len(nums) - 1)
```

shortest-word-distance-iii.py

```python
# shortest-word-distance-iii is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {string[]} words
    # @param {string} word1
    # @param {string} word2
    # @return {integer}
    def shortestWordDistance(self, words, word1, word2):
        dist = float("inf")
        is_same = (word1 == word2)
        i, index1, index2 = 0, None, None
        while i < len(words):
            if words[i] == word1:
                if is_same and index1 is not None:
                    dist = min(dist, abs(index1 - i))
                index1 = i
            elif words[i] == word2:
                index2 = i

            if index1 is not None and index2 is not None:
                dist = min(dist, abs(index1 - index2))
            i += 1

        return dist
```

smallest-range.py

```python
# smallest-range is not found.
# Time:  O(nlogk)
# Space: O(k)

import heapq


class Solution(object):
    def smallestRange(self, nums):
        """
        :type nums: List[List[int]]
        :rtype: List[int]
        """
        left, right = float("inf"), float("-inf")
        min_heap = []
        for row in nums:
            left = min(left, row[0])
            right = max(right, row[0])
            it = iter(row)
            heapq.heappush(min_heap, (next(it, None), it))

        result = (left, right)
        while min_heap:
            (val, it) = heapq.heappop(min_heap)
            val = next(it, None)
            if val is None:
                break
            heapq.heappush(min_heap, (val, it))
            left, right = min_heap[0][0], max(right, val)
            if right - left < result[1] - result[0]:
                result = (left, right)
        return result
```

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

```python
# DESC
# Given a VPS seq, split it into two disjoint subsequences A and B, such that A an
# d B are VPS's (and A.length + B.length = seq.length).
# Return an answer array (of length seq.length) that encodes such a choice of A an
# d 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:
# For example,  "", "()()", and "(()())" are VPS's (with nesting depths 0, 1, an
# d 2), and ")(" and "(()" are not VPS's.
# Example 2:
# Now choose any such A and B such that max(depth(A), depth(B)) is the minimum pos
# sible value.
# A string is a valid parentheses string (denoted VPS) if and only if it consists
# of "(" and ")" characters only, and:
# We can similarly define the nesting depth depth(S) of any VPS S as follows:
# Constraints:
# depth(S)

# NOTE
# 1 <= seq.size <= 10000
# depth("") = 0
# depth("(" + A + ")") = 1 + depth(A), where A is a VPS.
# It is the empty string, or
# depth(A + B) = max(depth(A), depth(B)), where A and B are VPS's
# 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.

# EXAMPLE
# Input: seq = "()(())()"
# Output: [0,0,0,1,1,0,1,1]
# Input: seq = "((()))"
# Output: [0,1,1,1,1,0]

# Time:  O(n)
# Space: O(1)

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

zigzag-iterator.py

```
# zigzag-iterator is not found.
# Time:  O(n)
# Space: O(k)

import collections


class ZigzagIterator(object):

    def __init__(self, v1, v2):
        """
        Initialize your q structure here.
        :type v1: List[int]
        :type v2: List[int]
        """
        self.q = collections.deque([(len(v), iter(v)) for v in (v1, v2) if v])

    def next(self):
        """
        :rtype: int
        """
        len, iter = self.q.popleft()
        if len > 1:
            self.q.append((len-1, iter))
        return next(iter)

    def hasNext(self):
        """
        :rtype: bool
        """
        return bool(self.q)
```

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

```
# DESC
# Example 1:
# Given the root of a binary tree, the level of its root is 1, the level of its ch
# ildren is 2, and so on.
# Return the smallest level X such that the sum of all the values of nodes at leve
# l X is maximal.
# Note:

# NOTE
# -10^5 <= node.val <= 10^5
# The number of nodes in the given tree is between 1 and 10^4.

# EXAMPLE
# Input: [1,7,0,7,-8,null,null]
# Output: 2
# Explanation:
# Level 1 sum = 1.
# Level 2 s
# um = 7 + 0 = 7.
# Level 3 sum = 7 + -8 = -1.
# So we return the level with the maxim
# um sum which is level 2.

# 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 = []
        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

```
# DESC
# Constraints:
# Example 1:
# Example 2:
# 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.
# Note:
# candidates
# The same repeated number may be chosen from candidates unlimited number of times.

# NOTE
# The solution set must not contain duplicate combinations.
# 1 <= target <= 500
# 1 <= candidates[i] <= 200
# 1 <= candidates.length <= 30
# All numbers (including target) will be positive integers.
# Each element of candidate is unique.

# EXAMPLE
# Input: candidates = [2,3,5], target = 8,
# A solution set is:
# [
#   [2,2,2,2],
#   [2,
# 3,3],
#   [3,5]
# ]
# Input: candidates = [2,3,6,7], target = 7,
# A solution set is:
# [
#   [7],
#   [2,2,3]
# ]

# 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
```

kth-largest-element-in-an-array.py

```
# kth-largest-element-in-an-arra is not found.
# Time:  O(n) ~ O(n^2)
# Space: O(1)

from random import randint

class Solution(object):
    # @param {integer[]} nums
    # @param {integer} k
    # @return {integer}
    def findKthLargest(self, nums, k):
        left, right = 0, len(nums) - 1
        while left <= right:
            pivot_idx = randint(left, right)
            new_pivot_idx = self.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(self, 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
```

the-maze-iii.py

```
# the-maze-iii is not found.
# Time:  O(max(r, c) * wlogw)
# Space: O(w^2)

import heapq


class Solution(object):
    def findShortestWay(self, maze, ball, hole):
        """
        :type maze: List[List[int]]
        :type ball: List[int]
        :type hole: List[int]
        :rtype: str
        """
        ball, hole = tuple(ball), tuple(hole)
        dirs = {'u' : (-1, 0), 'r' : (0, 1), 'l' : (0, -1), 'd': (1, 0)}

        def neighbors(maze, node):
            for dir, vec in dirs.iteritems():
                cur_node, dist = list(node), 0
                while 0 <= cur_node[0]+vec[0] < len(maze) and \
                      0 <= cur_node[1]+vec[1] < len(maze[0]) and \
                      not maze[cur_node[0]+vec[0]][cur_node[1]+vec[1]]:
                    cur_node[0] += vec[0]
                    cur_node[1] += vec[1]
                    dist += 1
                    if tuple(cur_node) == hole:
                        break
                yield tuple(cur_node), dir, dist

        heap = [(0, '', ball)]
        visited = set()
        while heap:
            dist, path, node = heapq.heappop(heap)
            if node in visited: continue
            if node == hole: return path
            visited.add(node)
            for neighbor, dir, neighbor_dist in neighbors(maze, node):
                heapq.heappush(heap, (dist+neighbor_dist, path+dir, neighbor))

        return "impossible"
```

super-washing-machines.py

```
# DESC
# Note:
# Example1
# Example2
# You have n super washing machines on a line. Initially, each washing machine has
#  some dresses or is empty.
# Given an integer array representing the number of dresses in each washing machin
# e from left to right on the line, you should find the minimum number of moves to
#  make all the washing machines have the same number of dresses. If it is not pos
# sible to do it, return -1.
# Example3
# For each move, you could choose any m (1  m  n) washing machines, and pass one
#  dress of each washing machine to one of its adjacent washing machines  at the s
# ame time .

# NOTE
# The range of n is [1, 10000].
# The range of dresses number in a super washing machine is [0, 1e5].

# EXAMPLE
# Input: [1,0,5]
#
# Output: 3
#
# Explanation:
# 1st move:    1     0 <-- 5    =>    1
#    1     4
# 2nd move:    1 <-- 1 <-- 4    =>    2     1     3
# 3rd move:    2
#     1 <-- 3    =>    2     2     2
# Input: [0,3,0]
#
# Output: 2
#
# Explanation:
# 1st move:    0 <-- 3     0    =>    1
#    2     0
# 2nd move:    1     2 --> 0    =>    1     1     1
# Input: [0,2,0]
#
# Output: -1
#
# Explanation:
# It's impossible to make all the three
# washing machines have the same number of dresses.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMinMoves(self, machines):
        """
        :type machines: List[int]
        :rtype: int
        """
        total = sum(machines)
        if total % len(machines): return -1
```

```
result, target, curr = 0, total / len(machines), 0
for n in machines:
    curr += n - target
    result = max(result, max(n - target, abs(curr)))
return result
```

prefix-and-suffix-search.py

```
# DESC
# Given many words, words[i] has weight i.
# Examples:
# Note:
# Design a class WordFilter that supports one function, WordFilter.f(String prefix
# , String suffix). It will return the word with given prefix and suffix with maxi
# mum weight. If no word exists, return -1.
# WordFilter

# NOTE
# words[i] and prefix, suffix queries consist of lowercase letters only.
# words[i] has length in range [1, 10].
# words has length in range [1, 15000].
# For each test case, up to words.length queries WordFilter.f may be made.
# prefix, suffix have lengths in range [0, 10].

# EXAMPLE
# Input:
# WordFilter(["apple"])
# WordFilter.f("a", "e") // returns 0
# WordFilter.f("b
# ", "") // returns -1

# Time:  ctor:   O(w * l^2), w is the number of words, l is the word length on average
#        search: O(p + s)  , p is the length of the prefix, s is the length of the suffix,
# Space: O(t), t is the number of trie nodes

import collections


class WordFilter(object):

    def __init__(self, words):
        """
        :type words: List[str]
        """
        _trie = lambda: collections.defaultdict(_trie)
        self.__trie = _trie()

        for weight, word in enumerate(words):
            word += '#'
            for i in xrange(len(word)):
                cur = self.__trie
                cur["_weight"] = weight
                for j in xrange(i, 2*len(word)-1):
                    cur = cur[word[j%len(word)]]
                    cur["_weight"] = weight

    def f(self, prefix, suffix):
        """
        :type prefix: str
        :type suffix: str
        :rtype: int
        """
        cur = self.__trie
        for letter in suffix + '#' + prefix:
            if letter not in cur:
                return -1
```

```
            cur = cur[letter]
        return cur["_weight"]


# Time:  ctor:   O(w * l), w is the number of words, l is the word length on average
#        search: O(p + s + max(m, n)), p is the length of the prefix, s is the length of the suffix,
#                                      m is the number of the prefix match, n is the number of the suffix matc
# Space: O(w * l)
class Trie(object):

    def __init__(self):
        _trie = lambda: collections.defaultdict(_trie)
        self.__trie = _trie()

    def insert(self, word, i):
        def add_word(cur, i):
            if "_words" not in cur:
                cur["_words"] = []
            cur["_words"].append(i)

        cur = self.__trie
        add_word(cur, i)
        for c in word:
            cur = cur[c]
            add_word(cur, i)

    def find(self, word):
        cur = self.__trie
        for c in word:
            if c not in cur:
                return []
            cur = cur[c]
        return cur["_words"]


class WordFilter2(object):

    def __init__(self, words):
        """
        :type words: List[str]
        """
        self.__prefix_trie = Trie()
        self.__suffix_trie = Trie()
        for i in reversed(xrange(len(words))):
            self.__prefix_trie.insert(words[i], i)
            self.__suffix_trie.insert(words[i][::-1], i)

    def f(self, prefix, suffix):
        """
        :type prefix: str
        :type suffix: str
        :rtype: int
        """
        prefix_match = self.__prefix_trie.find(prefix)
        suffix_match = self.__suffix_trie.find(suffix[::-1])
        i, j = 0, 0
        while i != len(prefix_match) and j != len(suffix_match):
            if prefix_match[i] == suffix_match[j]:
                return prefix_match[i]
            elif prefix_match[i] > suffix_match[j]:
```

```
            i += 1
        else:
            j += 1
    return -1
```

flip-string-to-monotone-increasing.py

```
# DESC
# Example 2:
# Example 1:
# Example 3:
# Return the minimum number of flips to make S monotone increasing.
# Note:
# A string of '0's and '1's is monotone increasing if it consists of some number o
# f '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'.

# NOTE
# S only consists of '0' and '1' characters.
# 1 <= S.length <= 20000

# EXAMPLE
# Input: "00110"
# Output: 1
# Explanation: We flip the last digit to get 00111.
# Input: "010110"
# Output: 2
# Explanation: We flip to get 011111, or alternatively 0
# 00111.
# Input: "00011000"
# Output: 2
# Explanation: We flip to get 00000000.

# 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
```

max-consecutive-ones.py

```
# DESC
# Given a binary array, find the maximum number of consecutive 1s in this array.
# Example 1:
# Note:

# NOTE
# The input array will only contain 0 and 1.
# The length of input array is a positive integer and will not exceed 10,000

# EXAMPLE
# Input: [1,1,0,1,1,1]
# Output: 3
# Explanation: The first two digits or the last thr
# ee digits are consecutive 1s.
#     The maximum number of consecutive 1s is 3.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMaxConsecutiveOnes(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, local_max = 0, 0
        for n in nums:
            local_max = (local_max + 1 if n else 0)
            result = max(result, local_max)
        return result
```

count-triplets-that-can-form-two-arrays-of-equal-xor.py

```python
# count-triplets-that-can-form-two-arrays-of-equal-xor is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def countTriplets(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        count_sum = collections.defaultdict(lambda: [0, 0])
        count_sum[0] = [1, 0]
        result, prefix = 0, 0
        for i, x in enumerate(arr):
            prefix ^= x
            c, t = count_sum[prefix]
            # sum(i-(j+1) for j in index[prefix])
            # = len(index[prefix])*i - sum((j+1) for j in index[prefix])
            result += c*i - t
            count_sum[prefix] = [c+1, t+i+1]
        return result
```

grumpy-bookstore-owner.py

```
# DESC
# 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 grump
# y on the i-th minute, grumpy[i] = 1, otherwise grumpy[i] = 0.  When the bookstor
# e owner is grumpy, the customers of that minute are not satisfied, otherwise the
# y are satisfied.
# Example 1:
# grumpy[i] = 1
# 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.
# Note:

# NOTE
# 0 <= grumpy[i] <= 1
# 0 <= customers[i] <= 1000
# 1 <= X <= customers.length == grumpy.length <= 20000

# EXAMPLE
# 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 m
# inutes.
# The maximum number of customers that can be satisfied = 1 + 1 + 1 + 1 +
#  7 + 5 = 16.

# 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
```

two-sum.py

```python
# DESC
# Example:
# Given an array of integers, return indices of the two numbers such that they add
#  up to a specific target.
# You may assume that each input would have exactly one solution, and you may not
# use the same element twice.

# NOTE
#

# EXAMPLE
# Given nums = [2, 7, 11, 15], target = 9,
#
# Because nums[0] + nums[1] = 2 + 7 = 9,
#
# return [0, 1].

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

class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        lookup = {}
        for i, num in enumerate(nums):
            if target - num in lookup:
                return [lookup[target - num], i]
            lookup[num] = i

    def twoSum2(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        for i in nums:
            j = target - i
            tmp_nums_start_index = nums.index(i) + 1
            tmp_nums = nums[tmp_nums_start_index:]
            if j in tmp_nums:
                return [nums.index(i), tmp_nums_start_index + tmp_nums.index(j)]
```

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

```
# DESC
# Note:
# Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in
#  the tree.
# Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]
# Example 2:
# 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)."
# Example 1:

# NOTE
# All of the nodes' values will be unique.
# p and q are different and both values will exist in the binary tree.

# EXAMPLE
# 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.
# 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 accor
# ding to the LCA definition.

# 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

```
# DESC
# Example 2:
# Eventually, we reach all R * C spaces of the grid.
# Example 1:
# On a 2 dimensional grid with R rows and C columns, we start at (r0, c0) facing east.
# Now, we walk in a clockwise spiral shape to visit every position in this grid.
# Return a list of coordinates representing the positions of the grid in the order
#  they were visited.
# Note:
# Here, the north-west corner of the grid is at the first row and column, and the
# south-east corner of the grid is at the last row and column.
# Whenever we would move outside the boundary of the grid, we continue our walk ou
# tside the grid (but may return to the grid boundary later.)

# NOTE
# 1 <= R <= 100
# 0 <= c0 < C
# 0 <= r0 < R
# 1 <= C <= 100

# EXAMPLE
# Input: R = 5, C = 6, r0 = 1, c0 = 4
# Output: [[1,4],[1,5],[2,5],[2,4],[2,3],[1,3]
# ,[0,3],[0,4],[0,5],[3,5],[3,4],[3,3],[3,2],[2,2],[1,2],[0,2],[4,5],[4,4],[4,3],[
# 4,2],[4,1],[3,1],[2,1],[1,1],[0,1],[4,0],[3,0],[2,0],[1,0],[0,0]]
# Input: R = 1, C = 4, r0 = 0, c0 = 0
# Output: [[0,0],[0,1],[0,2],[0,3]]

# 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
```

monotonic-array.py

```
# monotonic-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isMonotonic(self, A):
        """
        :type A: List[int]
        :rtype: bool
        """
        inc, dec = False, False
        for i in xrange(len(A)-1):
            if A[i] < A[i+1]:
                inc = True
            elif A[i] > A[i+1]:
                dec = True
        return not inc or not dec
```

length-of-last-word.py

```python
# DESC
# Example:
# Given a string s consists of upper/lower-case alphabets and empty space characte
# rs ' ', return the length of last word (last word means the last appearing word
# if we loop from left to right) in the string.
# If the last word does not exist, return 0.
# Note: A word is defined as a maximal substring consisting of non-space character
# s only.

# NOTE
#

# EXAMPLE
# Input: "Hello World"
# Output: 5

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param s, a string
    # @return an integer
    def lengthOfLastWord(self, s):
        length = 0
        for i in reversed(s):
            if i == ' ':
                if length:
                    break
            else:
                length += 1
        return length

# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param s, a string
    # @return an integer
    def lengthOfLastWord(self, s):
        return len(s.strip().split(" ")[-1])
```

summary-ranges.py

```
# DESC
# Given a sorted integer array without duplicates, return the summary of its ranges.
# Example 2:
# Example 1:

# NOTE
#

# EXAMPLE
# Input:  [0,1,2,4,5,7]
# Output: ["0->2","4->5","7"]
# Explanation: 0,1,2 form a cont
# inuous range; 4,5 form a continuous range.
# Input:  [0,2,3,4,6,8,9]
# Output: ["0","2->4","6","8->9"]
# Explanation: 2,3,4 form
# a continuous range; 8,9 form a continuous range.

# Time:  O(n)
# Space: O(1)

import itertools
import re


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

        start, end = nums[0], nums[0]
        for i in xrange(1, len(nums) + 1):
            if i < len(nums) and nums[i] == end + 1:
                end = nums[i]
            else:
                interval = str(start)
                if start != end:
                    interval += "->" + str(end)
                ranges.append(interval)
                if i < len(nums):
                    start = end = nums[i]

        return ranges

# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param {integer[]} nums
    # @return {string[]}
    def summaryRanges(self, nums):
        return [re.sub('->.*>', '->', '->'.join(repr(n) for _, n in g))
            for _, g in itertools.groupby(enumerate(nums), lambda i_n: i_n[1]-i_n[0])]
```

guess-the-majority-in-a-hidden-array.py

```python
# guess-the-majority-in-a-hidden-arra is not found.
# Time:  O(n), n queries
# Space: O(1)

class ArrayReader(object):
    def query(self, a, b, c, d):
        """
        :type a, b, c, d: int
        :rtype int
        """
        pass

    def length(self):
        """
        :rtype int
        """
        pass


class Solution(object):
    def guessMajority(self, reader):
        """
        :type reader: ArrayReader
        :rtype: integer
        """
        count_a, count_b, idx_b = 1, 0, None
        value_0_1_2_3 = reader.query(0, 1, 2, 3)
        for i in reversed(xrange(4, reader.length())):
            value_0_1_2_i = reader.query(0, 1, 2, i)
            if value_0_1_2_i == value_0_1_2_3:  # nums[i] == nums[3]
                count_a = count_a+1
            else:
                count_b, idx_b = count_b+1, i
        value_0_1_2_4 = value_0_1_2_i
        for i in xrange(3):
            value_a_b_3_4 = reader.query(*[v for v in [0, 1, 2, 3, 4] if v != i])
            if value_a_b_3_4 == value_0_1_2_4:  # nums[i] == nums[3]
                count_a = count_a+1
            else:
                count_b, idx_b = count_b+1, i
        if count_a == count_b:
            return -1
        return 3 if count_a > count_b else idx_b
```

middle-of-the-linked-list.py

```
# DESC
# If there are two middle nodes, return the second middle node.
# Given a non-empty, singly linked list with head node head, return a middle node
# of linked list.
# Example 2:
# Note:
# Example 1:

# NOTE
# The number of nodes in the given list will be between 1 and 100.

# EXAMPLE
# Input: [1,2,3,4,5]
# Output: Node 3 from this list (Serialization: [3,4,5])
# The re
# turned node has value 3.  (The judge's serialization of this node is [3,4,5]).
# N
# ote that we returned a ListNode object ans, such that:
# ans.val = 3, ans.next.val
#  = 4, ans.next.next.val = 5, and ans.next.next.next = NULL.
# Input: [1,2,3,4,5,6]
# Output: Node 4 from this list (Serialization: [4,5,6])
# Sinc
# e the list has two middle nodes with values 3 and 4, we return the second one.

# Time:  O(n)
# Space: O(1)

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

class Solution(object):
    def middleNode(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        slow, fast = head, head
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        return slow
```

robot-bounded-in-circle.py

```python
# DESC
# instructions
# Example 3:
# Example 2:
# Return true if and only if there exists a circle in the plane such that the robo
# t never leaves the circle.
# Note:
# The robot performs the instructions given in order, and repeats them forever.
# Example 1:
# On an infinite plane, a robot initially stands at (0, 0) and faces north.  The r
# obot can receive one of three instructions:

# NOTE
# "R": turn 90 degress to the right.
# "L": turn 90 degrees to the left;
# instructions[i] is in {'G', 'L', 'R'}
# 1 <= instructions.length <= 100
# "G": go straight 1 unit;

# EXAMPLE
# 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.
# Input: "GL"
# Output: true
# Explanation:
# The robot moves from (0, 0) -> (0, 1) ->
# (-1, 1) -> (-1, 0) -> (0, 0) -> ...
# Input: "GG"
# Output: false
# Explanation:
# The robot moves north indefinitely.

# Time:  O(n)
# Space: O(1)

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

valid-number.py

```
# DESC
# Some examples:
#
# "0" => true
#
# " 0.1 " => true
#
# "abc" => false
#
# "1 a" => false
#
# "2
# e10" => true
#
# " -90e3   " => true
#
# " 1e" => false
#
# "e3" => false
#
# " 6e-1" => tru
# e
#
# " 99e2.5 " => false
#
# "53.5e93" => true
#
# " --6 " => false
#
# "-+3" => false
#
# "95
# a54e53" => false
# Note: It is intended for the problem statement to be ambiguous. You should gathe
# r all requirements up front before implementing one. However, here is a list of
# characters that can be in a valid decimal number:
# Validate if a given string can be interpreted as a decimal number.
# Of course, the context of these characters also matters in the input.
# Update (2015-02-10):
#
# The signature of the C++ function had been updated. If you
#  still see your function signature accepts a const char * argument, please click
#  the reload button to reset your code definition.

# NOTE
# Exponent - "e"
# Positive/negative sign - "+"/"-"
# Numbers 0-9
# Decimal point - "."

# EXAMPLE
#

# Time:  O(n)
# Space: O(1)

class InputType(object):
    INVALID    = 0
```

```
    SPACE      = 1
    SIGN       = 2
    DIGIT      = 3
    DOT        = 4
    EXPONENT   = 5


# regular expression: "^\s*[\+-]?((\d+(\.\d*)?)|\.\d+)([eE][\+-]?\d+)?\s*$"
# automata: http://images.cnitblog.com/i/627993/201405/012016243309923.png
class Solution(object):
    def isNumber(self, s):
        """
        :type s: str
        :rtype: bool
        """
        transition_table = [[-1,  0,  3,  1,  2, -1],     # next states for state 0
                            [-1,  8, -1,  1,  4,  5],     # next states for state 1
                            [-1, -1, -1,  4, -1, -1],     # next states for state 2
                            [-1, -1, -1,  1,  2, -1],     # next states for state 3
                            [-1,  8, -1,  4, -1,  5],     # next states for state 4
                            [-1, -1,  6,  7, -1, -1],     # next states for state 5
                            [-1, -1, -1,  7, -1, -1],     # next states for state 6
                            [-1,  8, -1,  7, -1, -1],     # next states for state 7
                            [-1,  8, -1, -1, -1, -1]]     # next states for state 8

        state = 0
        for char in s:
            inputType = InputType.INVALID
            if char.isspace():
                inputType = InputType.SPACE
            elif char == '+' or char == '-':
                inputType = InputType.SIGN
            elif char.isdigit():
                inputType = InputType.DIGIT
            elif char == '.':
                inputType = InputType.DOT
            elif char == 'e' or char == 'E':
                inputType = InputType.EXPONENT

            state = transition_table[state][inputType]

            if state == -1:
                return False

        return state == 1 or state == 4 or state == 7 or state == 8


class Solution2(object):
    def isNumber(self, s):
        """
        :type s: str
        :rtype: bool
        """
        import re
        return bool(re.match("^\s*[\+-]?((\d+(\.\d*)?)|\.\d+)([eE][\+-]?\d+)?\s*$", s))
```

daily-temperatures.py

```
# DESC
# Given a list of daily temperatures T, return a list such that, for each day in t
# he input, tells you how many days you would have to wait until a warmer temperat
# ure.  If there is no future day for which this is possible, put 0 instead.
# For example, given the list of temperatures T = [73, 74, 75, 71, 69, 72, 76, 73]
# , your output should be [1, 1, 4, 2, 1, 1, 0, 0].
# T = [73, 74, 75, 71, 69, 72, 76, 73]
# Note:
# The length of temperatures will be in the range [1, 30000].
# Each temperatu
# re will be an integer in the range [30, 100].

# NOTE
#

# EXAMPLE
#

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

class Solution(object):
    def dailyTemperatures(self, temperatures):
        """
        :type temperatures: List[int]
        :rtype: List[int]
        """
        result = [0] * len(temperatures)
        stk = []
        for i in xrange(len(temperatures)):
            while stk and \
                  temperatures[stk[-1]] < temperatures[i]:
                idx = stk.pop()
                result[idx] = i-idx
            stk.append(i)
        return result
```

minimum-insertions-to-balance-a-parentheses-string.py

```python
# minimum-insertions-to-balance-a-parentheses-string is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minInsertions(self, s):
        """
        :type s: str
        :rtype: int
        """
        add, bal = 0, 0
        for c in s:
            if c == '(':
                if bal > 0 and bal%2:
                    add += 1
                    bal -= 1
                bal += 2
            else:
                bal -= 1
                if bal < 0:
                    add += 1
                    bal += 2
        return add + bal
```

longest-duplicate-substring.py

```
# DESC
# Return any duplicated substring that has the longest possible length.  (If S doe
# s not have a duplicated substring, the answer is "".)
# Example 2:
# Example 1:
# Given a string S, consider all duplicated substrings: (contiguous) substrings of
#  S that occur 2 or more times.  (The occurrences may overlap.)
# Note:

# NOTE
# S consists of lowercase English letters.
# 2 <= S.length <= 10^5

# EXAMPLE
# Input: "abcd"
# Output: ""
# Input: "baa"
# Output: "ana"

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

# 1. other solution is to apply kasai's algorithm, refer to the link below:
# - https://leetcode.com/problems/longest-duplicate-substring/discuss/290852/Suffix-array-clear-solution
# 2. the best solution is to apply ukkonen's algorithm, refer to the link below:
# - https://leetcode.com/problems/longest-duplicate-substring/discuss/312999/best-java-on-complexity-and-on-sp

import collections


class Solution(object):
    def longestDupSubstring(self, S):
        """
        :type S: str
        :rtype: str
        """
        M = 10**9+7
        D = 26

        def check(S, L):
            p = pow(D, L, M)
            curr = reduce(lambda x, y: (D*x+ord(y)-ord('a')) % M, S[:L], 0)
            lookup = collections.defaultdict(list)
            lookup[curr].append(L-1)
            for i in xrange(L, len(S)):
                curr = ((D*curr) % M + ord(S[i])-ord('a') -
                        ((ord(S[i-L])-ord('a'))*p) % M) % M
                if curr in lookup:
                    for j in lookup[curr]:  # check if string is the same when hash is the same
                        if S[j-L+1:j+1] == S[i-L+1:i+1]:
                            return i-L+1
                lookup[curr].append(i)
            return 0

        left, right = 1, len(S)-1
        while left <= right:
            mid = left + (right-left)//2
            if not check(S, mid):
```

```
            right = mid-1
        else:
            left = mid+1
    result = check(S, right)
    return S[result:result + right]
```

escape-the-ghosts.py

```
# DESC
# Return True if and only if it is possible to escape.
# You escape if and only if you can reach the target before any ghost reaches you
# (for any given moves the ghosts may take.)  If you reach any square (including t
# he target) at the same time as a ghost, it doesn't count as an escape.
# You are playing a simplified Pacman game. You start at the point (0, 0), and you
# r destination is (target[0], target[1]). There are several ghosts on the map, th
# e i-th ghost starts at (ghosts[i][0], ghosts[i][1]).
# Each turn, you and all ghosts simultaneously *may* move in one of 4 cardinal dir
# ections: north, east, west, or south, going from the previous point to a new poi
# nt 1 unit of distance away.
# Note:

# NOTE
# The number of ghosts will not exceed 100.
# All points have coordinates with absolute value <= 10000.

# EXAMPLE
# Example 1:
# Input:
# ghosts = [[1, 0], [0, 3]]
# target = [0, 1]
# Output: true
# Explan
# ation:
# You can directly reach the destination (0, 1) at time 1, while the ghost
# s located at (1, 0) or (0, 3) have no way to catch up with you.
# Example 3:
# Input:
# ghosts = [[2, 0]]
# target = [1, 0]
# Output: false
# Explanation:
#
# The ghost can reach the target at the same time as you.
# Example 2:
# Input:
# ghosts = [[1, 0]]
# target = [2, 0]
# Output: false
# Explanation:
#
# You need to reach the destination (2, 0), but the ghost at (1, 0) lies between
# you and the destination.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def escapeGhosts(self, ghosts, target):
        """
        :type ghosts: List[List[int]]
        :type target: List[int]
        :rtype: bool
        """
        total = abs(target[0])+abs(target[1])
        return all(total < abs(target[0]-i)+abs(target[1]-j) for i, j in ghosts)
```

repeated-substring-pattern.py

```
# DESC
# Given a non-empty string check if it can be constructed by taking a substring of
#  it and appending multiple copies of the substring together. You may assume the
# given string consists of lowercase English letters only and its length will not
# exceed 10000.
# Example 2:
# Example 3:
# Example 1:

# NOTE
#

# EXAMPLE
# Input: "abcabcabcabc"
# Output: True
# Explanation: It's the substring "abc" four ti
# mes. (And the substring "abcabc" twice.)
# Input: "abab"
# Output: True
# Explanation: It's the substring "ab" twice.
# Input: "aba"
# Output: False

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

class Solution(object):
    def repeatedSubstringPattern(self, str):
        """
        :type str: str
        :rtype: bool
        """
        def getPrefix(pattern):
            prefix = [-1] * len(pattern)
            j = -1
            for i in xrange(1, len(pattern)):
                while j > -1 and pattern[j + 1] != pattern[i]:
                    j = prefix[j]
                if pattern[j + 1] == pattern[i]:
                    j += 1
                prefix[i] = j
            return prefix

        prefix = getPrefix(str)
        return prefix[-1] != -1 and \
               (prefix[-1] + 1) % (len(str) - prefix[-1] - 1) == 0

    def repeatedSubstringPattern2(self, str):
        """
        :type str: str
        :rtype: bool
        """
        if not str:
            return False

        ss = (str + str)[1:-1]
        return ss.find(str) != -1
```

integer-replacement.py

```
# DESC
# Given a positive integer n and you can do operations as follow:
# Example 1:
# What is the minimum number of replacements needed for n to become 1?
# Example 2:

# NOTE
# If n is odd, you can replace n with either n + 1 or n - 1.
# If n is even, replace n with n/2.

# EXAMPLE
# Input:
# 7
#
# Output:
# 4
#
# Explanation:
# 7 -> 8 -> 4 -> 2 -> 1
# or
# 7 -> 6 -> 3 -> 2 -> 1
# Input:
# 8
#
# Output:
# 3
#
# Explanation:
# 8 -> 4 -> 2 -> 1

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def integerReplacement(self, n):
        """
        :type n: int
        :rtype: int
        """
        result = 0
        while n != 1:
            b = n & 3
            if n == 3:
                n -= 1
            elif b == 3:
                n += 1
            elif b == 1:
                n -= 1
            else:
                n /= 2
            result += 1

        return result


# Time:  O(logn)
# Space: O(logn)
# Recursive solution.
```

```python
class Solution2(object):
    def integerReplacement(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n < 4:
            return [0, 0, 1, 2][n]
        if n % 4 in (0, 2):
            return self.integerReplacement(n / 2) + 1
        elif n % 4 == 1:
            return self.integerReplacement((n - 1) / 4) + 3
        else:
            return self.integerReplacement((n + 1) / 4) + 3
```

super-pow.py

```
# DESC
# Example 1:
# 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 2:

# NOTE
#

# EXAMPLE
# Input: a = 2, b = [1,0]
# Output: 1024
# Input: a = 2, b = [3]
# Output: 8

# 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
```

minimize-max-distance-to-gas-station.py

```
# minimize-max-distance-to-gas-station is not found.
# Time:  O(nlogr)
# Space: O(1)

class Solution(object):
    def minmaxGasDist(self, stations, K):
        """
        :type stations: List[int]
        :type K: int
        :rtype: float
        """
        def possible(stations, K, guess):
            return sum(int((stations[i+1]-stations[i]) / guess)
                       for i in xrange(len(stations)-1)) <= K

        left, right = 0, 10**8
        while right-left > 1e-6:
            mid = left + (right-left)/2.0
            if possible(mid):
                right = mid
            else:
                left = mid
        return left
```

how-many-numbers-are-smaller-than-the-current-number.py

```python
# how-many-numbers-are-smaller-than-the-current-number is not found.
# Time:  O(n + m), m is the max number of nums
# Space: O(m)

import collections


class Solution(object):
    def smallerNumbersThanCurrent(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        count = collections.Counter(nums)
        for i in xrange(max(nums)+1):
            count[i] += count[i-1]
        return [count[i-1] for i in nums]


# Time:  O(nlogn)
# Space: O(n)
import bisect


class Solution2(object):
    def smallerNumbersThanCurrent(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        sorted_nums = sorted(nums)
        return [bisect.bisect_left(sorted_nums, i) for i in nums]
```

insert-delete-getrandom-o1.py

```python
# insert-delete-getrandom-o1 is not found.
# Time:  O(1)
# Space: O(n)

from random import randint

class RandomizedSet(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__set = []
        self.__used = {}


    def insert(self, val):
        """
        Inserts a value to the set. Returns true if the set did not already contain the specified element.
        :type val: int
        :rtype: bool
        """
        if val in self.__used:
            return False

        self.__set += val,
        self.__used[val] = len(self.__set)-1

        return True


    def remove(self, val):
        """
        Removes a value from the set. Returns true if the set contained the specified element.
        :type val: int
        :rtype: bool
        """
        if val not in self.__used:
            return False

        self.__used[self.__set[-1]] = self.__used[val]
        self.__set[self.__used[val]], self.__set[-1] = self.__set[-1], self.__set[self.__used[val]]

        self.__used.pop(val)
        self.__set.pop()

        return True

    def getRandom(self):
        """
        Get a random element from the set.
        :rtype: int
        """
        return self.__set[randint(0, len(self.__set)-1)]
```

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

```
# DESC
# Constraints:
# Example 1:
# Given a binary search tree (BST), find the lowest common ancestor (LCA) of two g
# iven nodes in the BST.
# Example 2:
# Given binary search tree:  root = [6,2,8,0,4,7,9,null,null,3,5]
# 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)."

# NOTE
# p and q are different and both values will exist in the BST.
# All of the nodes' values will be unique.

# EXAMPLE
# Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
# Output: 2
# Explanation:
#  The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself accor
# ding to the LCA definition.
# Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
# Output: 6
# Explanation:
#  The LCA of nodes 2 and 8 is 6.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {TreeNode} root
    # @param {TreeNode} p
    # @param {TreeNode} q
    # @return {TreeNode}
    def lowestCommonAncestor(self, root, p, q):
        s, b = sorted([p.val, q.val])
        while not s <= root.val <= b:
            # Keep searching since root is outside of [s, b].
            root = root.left if s <= root.val else root.right
        # s <= root.val <= b.
        return root
```

stepping-numbers.py

```
# stepping-numbers is not found.
# Time:  O(logk + r), r is the size of result
# Space: O(k), k is the size of stepping numbers in [0, high]

import bisect


MAX_HIGH = int(2e9)
result = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in xrange(1, MAX_HIGH):
    if result[-1] >= MAX_HIGH:
        break
    d1 = result[i]%10 - 1
    if d1 >= 0:
        result.append(result[i]*10 + d1)
    d2 = result[i]%10 + 1
    if d2 <= 9:
        result.append(result[i]*10 + d2)
result.append(float("inf"))


class Solution(object):
    def countSteppingNumbers(self, low, high):
        """
        :type low: int
        :type high: int
        :rtype: List[int]
        """
        lit = bisect.bisect_left(result, low);
        rit = bisect.bisect_right(result, high);
        return result[lit:rit]


# Time:  O(k + r), r is the size of result
# Space: O(k), k is the size of stepping numbers in [0, high]
class Solution2(object):
    def countSteppingNumbers(self, low, high):
        """
        :type low: int
        :type high: int
        :rtype: List[int]
        """
        result = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        for i in xrange(1, high):
            if result[-1] >= high:
                break
            d1 = result[i]%10 - 1
            if d1 >= 0:
                result.append(result[i]*10 + d1)
            d2 = result[i]%10 + 1
            if d2 <= 9:
                result.append(result[i]*10 + d2)
        result.append(float("inf"))
        lit = bisect.bisect_left(result, low);
        rit = bisect.bisect_right(result, high);
        return result[lit:rit]
```

first-unique-number.py

```
# first-unique-number is not found.
# Time:  ctor: O(k)
#        add: O(1)
#        showFirstUnique: O(1)
# Space: O(n)

import collections


class FirstUnique(object):

    def __init__(self, nums):
        """
        :type nums: List[int]
        """
        self.__q = collections.OrderedDict()
        self.__dup = set()
        for num in nums:
            self.add(num)

    def showFirstUnique(self):
        """
        :rtype: int
        """
        if self.__q:
            return next(iter(self.__q))
        return -1

    def add(self, value):
        """
        :type value: int
        :rtype: None
        """
        if value not in self.__dup and value not in self.__q:
            self.__q[value] = None
            return
        if value in self.__q:
            self.__q.pop(value)
            self.__dup.add(value)
```

the-skyline-problem.py

```
# DESC
# The geometric information of each building is represented by a triplet of intege
# rs [Li, Ri, Hi], where Li and Ri are the x coordinates of the left and right edg
# e of the ith building, respectively, and Hi is its height. It is guaranteed that
#  0  Li, Ri  INT_MAX, 0 < Hi  INT_MAX, and Ri - Li > 0. You may assume all bui
# ldings are perfect rectangles grounded on an absolutely flat surface at height 0
# .
# The output is a list of "key points" (red dots in Figure B) in the format of [ [
# x1,y1], [x2, y2], [x3, y3], ... ] that uniquely defines a skyline. A key point i
# s the left endpoint of a horizontal line segment. Note that the last key point,
# where the rightmost building ends, is merely used to mark the termination of the
#  skyline, and always has zero height. Also, the ground in between any two adjace
# nt buildings should be considered part of the skyline contour.
# For instance, the dimensions of all buildings in Figure A are recorded as: [ [2
# 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ] .
# A city's skyline is the outer contour of the silhouette formed by all the buildi
# ngs in that city when viewed from a distance. Now suppose you are given the loca
# tions and height of all the buildings as shown on a cityscape photo (Figure A),
# write a program to output the skyline formed by these buildings collectively (Fi
# gure B).
# [Li, Ri, Hi]
# Notes:
# For instance, the skyline in Figure B should be represented as:[ [2 10], [3 15],
#  [7 12], [12 0], [15 10], [20 8], [24, 0] ].

# NOTE
# There must be no consecutive horizontal lines of equal height in the output skyl
# ine. For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not acceptable
# ; the three lines of height 5 should be merged into one in the final output as s
# uch: [...[2 3], [4 5], [12 7], ...]
# The number of buildings in any input list is guaranteed to be in the range [0, 1
# 0000].
# The input list is already sorted in ascending order by the left x position Li.
# The output list must be sorted by the x position.

# EXAMPLE
#

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

start, end, height = 0, 1, 2
class Solution(object):
    # @param {integer[][]} buildings
    # @return {integer[][]}
    def getSkyline(self, buildings):
        intervals = self.ComputeSkylineInInterval(buildings, 0, len(buildings))

        res = []
        last_end = -1
        for interval in intervals:
            if last_end != -1 and last_end < interval[start]:
                res.append([last_end, 0])
            res.append([interval[start], interval[height]])
            last_end = interval[end]
        if last_end != -1:
            res.append([last_end, 0])
```

```
        return res

    # Divide and Conquer.
    def ComputeSkylineInInterval(self, buildings, left_endpoint, right_endpoint):
        if right_endpoint - left_endpoint <= 1:
            return buildings[left_endpoint:right_endpoint]
        mid = left_endpoint + ((right_endpoint - left_endpoint) / 2)
        left_skyline = self.ComputeSkylineInInterval(buildings, left_endpoint, mid)
        right_skyline = self.ComputeSkylineInInterval(buildings, mid, right_endpoint)
        return self.MergeSkylines(left_skyline, right_skyline)

    # Merge Sort.
    def MergeSkylines(self, left_skyline, right_skyline):
        i, j = 0, 0
        merged = []

        while i < len(left_skyline) and j < len(right_skyline):
            if left_skyline[i][end] < right_skyline[j][start]:
                merged.append(left_skyline[i])
                i += 1
            elif right_skyline[j][end] < left_skyline[i][start]:
                merged.append(right_skyline[j])
                j += 1
            elif left_skyline[i][start] <= right_skyline[j][start]:
                i, j = self.MergeIntersectSkylines(merged, left_skyline[i], i,\
                                                   right_skyline[j], j)
            else: # left_skyline[i][start] > right_skyline[j][start].
                j, i = self.MergeIntersectSkylines(merged, right_skyline[j], j, \
                                                   left_skyline[i], i)

        # Insert the remaining skylines.
        merged += left_skyline[i:]
        merged += right_skyline[j:]
        return merged

    # a[start] <= b[start]
    def MergeIntersectSkylines(self, merged, a, a_idx, b, b_idx):
        if a[end] <= b[end]:
            if a[height] > b[height]:   # |aaa|
                if b[end] != a[end]:    # |abb|b
                    b[start] = a[end]
                    merged.append(a)
                    a_idx += 1
                else:               # aaa
                    b_idx += 1      # abb
            elif a[height] == b[height]:  # abb
                b[start] = a[start]       # abb
                a_idx += 1
            else:  # a[height] < b[height].
                if a[start] != b[start]:                              #    bb
                    merged.append([a[start], b[start], a[height]])  # |a|bb
                a_idx += 1
        else:  # a[end] > b[end].
            if a[height] >= b[height]:  # aaaa
                b_idx += 1              # abba
            else:
                #    |bb|
                # |a||bb|a
                if a[start] != b[start]:
                    merged.append([a[start], b[start], a[height]])
```

```
            a[start] = b[end]
            merged.append(b)
            b_idx += 1
    return a_idx, b_idx
```

minimum-window-subsequence.py

```
# minimum-window-subsequence is not found.
# Time:  O(s * t)
# Space: O(s)

class Solution(object):
    def minWindow(self, S, T):
        """
        :type S: str
        :type T: str
        :rtype: str
        """
        lookup = [[None for _ in xrange(26)] for _ in xrange(len(S)+1)]
        find_char_next_pos = [None]*26
        for i in reversed(xrange(len(S))):
            find_char_next_pos[ord(S[i])-ord('a')] = i+1
            lookup[i] = list(find_char_next_pos)

        min_i, min_len = None, float("inf")
        for i in xrange(len(S)):
            if S[i] != T[0]:
                continue
            start = i
            for c in T:
                start = lookup[start][ord(c)-ord('a')]
                if start == None:
                    break
            else:
                if start-i < min_len:
                    min_i, min_len = i, start-i
        return S[min_i:min_i+min_len] if min_i is not None else ""


# Time:  O(s * t)
# Space: O(s)
class Solution2(object):
    def minWindow(self, S, T):
        """
        :type S: str
        :type T: str
        :rtype: str
        """
        dp = [[None for _ in xrange(len(S))] for _ in xrange(2)]
        for j, c in enumerate(S):
            if c == T[0]:
                dp[0][j] = j

        for i in xrange(1, len(T)):
            prev = None
            dp[i%2] = [None] * len(S)
            for j, c in enumerate(S):
                if prev is not None and c == T[i]:
                    dp[i%2][j] = prev
                if dp[(i-1)%2][j] is not None:
                    prev = dp[(i-1)%2][j]

        start, end = 0, len(S)
        for j, i in enumerate(dp[(len(T)-1)%2]):
            if i >= 0 and j-i < end-start:
```

```
        start, end = i, j
    return S[start:end+1] if end < len(S) else ""
```

find-duplicate-subtrees.py

```
# DESC
# Two trees are duplicate if they have the same structure with same node values.
# Example 1:
# Given a binary tree, return all duplicate subtrees. For each kind of duplicate s
# ubtrees, you only need to return the root node of any one of them.
# The following are two duplicate subtrees:

# NOTE
#

# EXAMPLE
# 2
#      /
#     4
# 1
#        / \
#       2   3
#      /   / \
#     4   2   4
#        /
#       4

# 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]


# 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
```

number-of-ways-to-paint-n-3-grid.py

```
# number-of-ways-to-paint-n-3-grid is not found.
# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def numOfWays(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 % MOD for a, b in itertools.izip(row, col)) % MOD
                     for col in ZB] for row in A]

        MOD = 10**9 + 7
        T = [[3, 2],
             [2, 2]]
        return sum(matrix_mult([[6, 6]], matrix_expo(T, n-1))[0]) % MOD  # [a1, a0] * T^(n-1)


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def numOfWays(self, n):
        """
        :type n: int
        :rtype: int
        """
        MOD = 10**9 + 7
        aba, abc = 6, 6
        for _ in xrange(n-1):
            aba, abc = (3*aba%MOD + 2*abc%MOD)%MOD, \
                       (2*abc%MOD + 2*aba%MOD)%MOD
        return (aba+abc)%MOD
```

insertion-sort-list.py

```
# DESC
# 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
# Sort a linked list using insertion sort.
# Example 2:
# Example 1:
# Algorithm of Insertion Sort:

# NOTE
# 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.
# Insertion sort iterates, consuming one input element each repetition, and growin
# g a sorted output list.
# It repeats until no input elements remain.

# EXAMPLE
# Input: -1->5->3->4->0
# Output: -1->0->3->4->5
# Input: 4->2->1->3
# Output: 1->2->3->4

# 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
# DESC
# There is a special square room with mirrors on each of the four walls.  Except f
# or the southwest corner, there are receptors on each of the remaining corners, n
# umbered 0, 1, and 2.
# Return the number of the receptor that the ray meets first.  (It is guaranteed t
# hat the ray will meet a receptor eventually.)
# Note:
# Example 1:
# The square room has walls of length p, and a laser ray from the southwest corner
#  first meets the east wall at a distance q from the 0th receptor.

# NOTE
# 1 <= p <= 1000
# 0 <= q <= p

# EXAMPLE
# Input: p = 2, q = 1
# Output: 2
# Explanation: The ray meets receptor 2 the first ti
# me it gets reflected back to the left wall.

# 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)
```

design-snake-game.py

```
# design-snake-game is not found.
# Time:  O(1) per move
# Space: O(s), s is the current length of the snake.

from collections import defaultdict, deque

class SnakeGame(object):

    def __init__(self, width,height,food):
        """
        Initialize your data structure here.
        @param width - screen width
        @param height - screen height
        @param food - A list of food positions
        E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second is at [1,0].
        :type width: int
        :type height: int
        :type food: List[List[int]]
        """
        self.__width = width
        self.__height = height
        self.__score = 0
        self.__food = deque(food)
        self.__snake = deque([(0, 0)])
        self.__direction = {"U": (-1, 0), "L": (0, -1), "R": (0, 1), "D": (1, 0)}
        self.__lookup = defaultdict(int)
        self.__lookup[(0, 0)] += 1

    def move(self, direction):
        """
        Moves the snake.
        @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
        @return The game's score after the move. Return -1 if game over.
        Game over when snake crosses the screen boundary or bites its body.
        :type direction: str
        :rtype: int
        """
        def valid(x, y):
            return 0 <= x < self.__height and \
                   0 <= y < self.__width and \
                   (x, y) not in self.__lookup
        d = self.__direction[direction]
        x, y = self.__snake[-1][0] + d[0], self.__snake[-1][1] + d[1]

        tail = self.__snake[-1]
        self.__lookup[self.__snake[0]] -= 1
        if self.__lookup[self.__snake[0]] == 0:
            self.__lookup.pop(self.__snake[0])
        self.__snake.popleft()
        if not valid(x, y):
            return -1
        elif self.__food and (self.__food[0][0], self.__food[0][1]) == (x, y):
            self.__score += 1
            self.__food.popleft()
            self.__snake.appendleft(tail)
            self.__lookup[tail] += 1
        self.__snake += (x, y),
        self.__lookup[(x, y)] += 1
```

```
        return self.__score
```

quad-tree-intersection.py

```
# quad-tree-intersection is not found.
# 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


class Solution(object):
    def intersect(self, quadTree1, quadTree2):
        """
        :type quadTree1: Node
        :type quadTree2: Node
        :rtype: Node
        """
        if quadTree1.isLeaf:
            return quadTree1 if quadTree1.val else quadTree2
        elif quadTree2.isLeaf:
            return quadTree2 if quadTree2.val else quadTree1
        topLeftNode = self.intersect(quadTree1.topLeft, quadTree2.topLeft)
        topRightNode = self.intersect(quadTree1.topRight, quadTree2.topRight)
        bottomLeftNode = self.intersect(quadTree1.bottomLeft, quadTree2.bottomLeft)
        bottomRightNode = self.intersect(quadTree1.bottomRight, quadTree2.bottomRight)
        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)
```

allocate-mailboxes.py

```python
# allocate-mailboxes is not found.
# Time:  O(m * n^2)
# Space: O(n)

class Solution(object):
    def minDistance(self, houses, k):
        """
        :type houses: List[int]
        :type k: int
        :rtype: int
        """
        def cost(prefix, i, j):
            return (prefix[j+1]-prefix[(i+j+1)//2])-(prefix[(i+j)//2+1]-prefix[i])

        houses.sort()
        prefix = [0]*(len(houses)+1)
        for i, h in enumerate(houses):
            prefix[i+1] = prefix[i]+h
        dp = [cost(prefix, 0, j) for j in xrange(len(houses))]
        for m in xrange(1, k):
            for j in reversed(xrange(m, len(houses))):
                for i in xrange(m, j+1):
                    dp[j] = min(dp[j], dp[i-1]+cost(prefix, i, j))
        return dp[-1]
```

minimum-cost-to-merge-stones.py

```
# DESC
# Example 1:
# Example 2:
# A move consists of merging exactly K consecutive piles into one pile, and the co
# st of this move is equal to the total number of stones in these K piles.
# Example 3:
# There are N piles of stones arranged in a row.  The i-th pile has stones[i] stones.
# Find the minimum cost to merge all piles of stones into one pile.  If it is impo
# ssible, return -1.
# Note:

# NOTE
# 2 <= K <= 30
# 1 <= stones[i] <= 100
# 1 <= stones.length <= 30

# EXAMPLE
# Input: stones = [3,5,1,2,6], K = 3
# Output: 25
# Explanation:
# We start with [3, 5,
#  1, 2, 6].
# We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6].
# W
# e merge [3, 8, 6] for a cost of 17, and we are left with [17].
# The total cost wa
# s 25, and this is the minimum possible.
# Input: stones = [3,2,4,1], K = 2
# Output: 20
# Explanation:
# We start with [3, 2, 4
# , 1].
# We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1].
# We merge
# [4, 1] for a cost of 5, and we are left with [5, 5].
# We merge [5, 5] for a cost
# of 10, and we are left with [10].
# The total cost was 20, and this is the minimum
#  possible.
# Input: stones = [3,2,4,1], K = 3
# Output: -1
# Explanation: After any merge operati
# on, there are 2 piles left, and we can't merge anymore.  So the task is impossib
# le.

# Time:  O(n^3 / k)
# Space: O(n^2)

class Solution(object):
    def mergeStones(self, stones, K):
        """
        :type stones: List[int]
        :type K: int
        :rtype: int
        """
        if (len(stones)-1) % (K-1):
            return -1
        prefix = [0]
```

```
for x in stones:
    prefix.append(prefix[-1]+x)
dp = [[0]*len(stones) for _ in xrange(len(stones))]
for l in xrange(K-1, len(stones)):
    for i in xrange(len(stones)-l):
        dp[i][i+l] = min(dp[i][j]+dp[j+1][i+l] for j in xrange(i, i+l, K-1))
        if l % (K-1) == 0:
            dp[i][i+l] += prefix[i+l+1] - prefix[i]
return dp[0][len(stones)-1]
```

circular-permutation-in-binary-representation.py

```python
# circular-permutation-in-binary-representation is not found.
# Time:  O(2^n)
# Space: O(1)

class Solution(object):
    def circularPermutation(self, n, start):
        """
        :type n: int
        :type start: int
        :rtype: List[int]
        """
        return [start ^ (i>>1) ^ i for i in xrange(1<<n)]
```

best-time-to-buy-and-sell-stock-with-transaction-fee.py

```
# DESC
# Return the maximum profit you can make.
# Your are given an array of integers prices, for which the i-th element is the pr
# ice of a given stock on day i; and a non-negative integer fee representing a tra
# nsaction fee.
# Note:
# Example 1:
# You may complete as many transactions as you like, but you need to pay the trans
# action fee for each transaction.  You may not buy more than 1 share of a stock a
# t a time (ie. you must sell the stock share before you buy again.)

# NOTE
# 0 < prices[i] < 50000.
# Selling at prices[3] = 8
# 0 < prices.length <= 50000.
# Selling at prices[5] = 9
# 0 <= fee < 50000.
# Buying at prices[0] = 1
# Buying at prices[4] = 4

# EXAMPLE
# Input: prices = [1, 3, 2, 8, 4, 9], fee = 2
# Output: 8
# Explanation: The maximum p
# rofit can be achieved by:
# Buying at prices[0] = 1Selling at prices[3] = 8Buying
# at prices[4] = 4Selling at prices[5] = 9The total profit is ((8 - 1) - 2) + ((9
# - 4) - 2) = 8.

# Time:  O(n)
# Space: O(1)


class Solution(object):
    def maxProfit(self, prices, fee):
        """
        :type prices: List[int]
        :type fee: int
        :rtype: int
        """
        cash, hold = 0, -prices[0]
        for i in xrange(1, len(prices)):
            cash = max(cash, hold+prices[i]-fee)
            hold = max(hold, cash-prices[i])
        return cash
```

loud-and-rich.py

```
# DESC
# Also, we'll say quiet[x] = q if person x has quietness q.
# In a group of N people (labelled 0, 1, 2, ..., N-1), each person has different a
# mounts of money, and different levels of quietness.
# Note:
# For convenience, we'll call the person with label x, simply "person x".
# We'll say that richer[i] = [x, y] if person x definitely has more money than per
# son y.  Note that richer may only be a subset of valid observations.
# Now, return answer, where answer[x] = y if y is the least quiet person (that is,
#  the person y with the smallest value of quiet[y]), among all people who definit
# ely have equal to or more money than person x.
# Example 1:

# NOTE
# richer[i][0] != richer[i][1]
# 0 <= quiet[i] < N, all quiet[i] are different.
# The observations in richer are all logically consistent.
# richer[i]'s are all different.
# 1 <= quiet.length = N <= 500
# 0 <= richer[i][j] < N
# 0 <= richer.length <= N * (N-1) / 2

# EXAMPLE
# Input: richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]], quiet = [3,2,5,4,6,
# 1,7,0]
# Output: [5,5,2,5,4,5,6,7]
# Explanation:
# answer[0] = 5.
# Person 5 has more
# money than 3, which has more money than 1, which has more money than 0.
# The only
#  person who is quieter (has lower quiet[x]) is person 7, but
# it isn't clear if t
# hey have more money than person 0.
#
# answer[7] = 7.
# Among all people that definit
# ely have equal to or more money than person 7
# (which could be persons 3, 4, 5, 6
# , or 7), the person who is the quietest (has lower quiet[x])
# is person 7.
#
# The o
# ther answers can be filled out with similar reasoning.

# Time:  O(q + r)
# Space: O(q + r)


class Solution(object):
    def loudAndRich(self, richer, quiet):
        """
        :type richer: List[List[int]]
        :type quiet: List[int]
        :rtype: List[int]
        """
        def dfs(graph, quiet, node, result):
            if result[node] is None:
```

```
        result[node] = node
        for nei in graph[node]:
            smallest_person = dfs(graph, quiet, nei, result)
            if quiet[smallest_person] < quiet[result[node]]:
                result[node] = smallest_person
    return result[node]

graph = [[] for _ in xrange(len(quiet))]
for u, v in richer:
    graph[v].append(u)
result = [None]*len(quiet)
return map(lambda x: dfs(graph, quiet, x, result), xrange(len(quiet)))
```

reformat-date.py

```
# reformat-date is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reformatDate(self, date):
        """
        :type date: str
        :rtype: str
        """
        lookup = {"Jan":1, "Feb":2, "Mar":3, "Apr":4,
                  "May":5, "Jun":6, "Jul":7, "Aug":8,
                  "Sep":9, "Oct":10, "Nov":11, "Dec":12}
        return "{:04d}-{:02d}-{:02d}".format(int(date[-4:]), lookup[date[-8:-5]], int(date[:date.index(' ')-2]
```

flower-planting-with-no-adjacent.py

```python
# DESC
# Your task is to choose a flower type for each garden such that, for any two gard
# ens connected by a path, they have different types of flowers.
# Example 2:
# Also, there is no garden that has more than 3 paths coming into or leaving it.
# Return any such a choice as an array answer, where answer[i] is the type of flow
# er planted in the (i+1)-th garden.  The flower types are denoted 1, 2, 3, or 4.
#  It is guaranteed an answer exists.
# Example 3:
# You have N gardens, labelled 1 to N.  In each garden, you want to plant one of 4
#  types of flowers.
# Example 1:
# paths[i] = [x, y] describes the existence of a bidirectional path from garden x
# to garden y.
# paths[i] = [x, y]
# Note:

# NOTE
# 0 <= paths.size <= 20000
# No garden has 4 or more paths coming into or leaving it.
# It is guaranteed an answer exists.
# 1 <= N <= 10000

# EXAMPLE
# Input: N = 4, paths = [[1,2],[3,4]]
# Output: [1,2,1,2]
# Input: N = 3, paths = [[1,2],[2,3],[3,1]]
# Output: [1,2,3]
# Input: N = 4, paths = [[1,2],[2,3],[3,4],[4,1],[1,3],[2,4]]
# Output: [1,2,3,4]

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

class Solution(object):
    def gardenNoAdj(self, N, paths):
        """
        :type N: int
        :type paths: List[List[int]]
        :rtype: List[int]
        """
        result = [0]*N
        G = [[] for i in xrange(N)]
        for x, y in paths:
            G[x-1].append(y-1)
            G[y-1].append(x-1)
        for i in xrange(N):
            result[i] = ({1, 2, 3, 4} - {result[j] for j in G[i]}).pop()
        return result
```

bulb-switcher-iv.py

```python
# bulb-switcher-iv is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minFlips(self, target):
        """
        :type target: str
        :rtype: int
        """
        result, curr = 0, '0'
        for c in target:
            if c == curr:
                continue
            curr = c
            result += 1
        return result
```

similar-rgb-color.py

```python
# similar-rgb-color is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def similarRGB(self, color):
        """
        :type color: str
        :rtype: str
        """
        def rounding(color):
            q, r = divmod(int(color, 16), 17)
            if r > 8: q += 1
            return '{:02x}'.format(17*q)

        return '#' + \
                rounding(color[1:3]) + \
                rounding(color[3:5]) + \
                rounding(color[5:7])
```

can-convert-string-in-k-moves.py

```python
# can-convert-string-in-k-moves is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def canConvertString(self, s, t, k):
        """
        :type s: str
        :type t: str
        :type k: int
        :rtype: bool
        """
        if len(s) != len(t):
            return False
        cnt = [0]*26
        for a, b in itertools.izip(s, t):
            diff = (ord(b)-ord(a)) % len(cnt)
            if diff != 0 and cnt[diff]*len(cnt) + diff > k:
                return False
            cnt[diff] += 1
        return True
```

remove-all-adjacent-duplicates-in-string.py

```
# DESC
# Return the final string after all such duplicate removals have been made.  It is
#  guaranteed the answer is unique.
# Given a string S of lowercase letters, a duplicate removal consists of choosing
# two adjacent and equal letters, and removing them.
# Example 1:
# We repeatedly make duplicate removals on S until we no longer can.
# Note:

# NOTE
# S consists only of English lowercase letters.
# 1 <= S.length <= 20000

# EXAMPLE
# Input: "abbaca"
# Output: "ca"
# Explanation:
# For example, in "abbaca" we could rem
# ove "bb" since the letters are adjacent and equal, and this is the only possible
#  move.  The result of this move is that the string is "aaca", of which only "aa"
#  is possible, so the final string is "ca".

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

class Solution(object):
    def removeDuplicates(self, S):
        """
        :type S: str
        :rtype: str
        """
        result = []
        for c in S:
            if result and result[-1] == c:
                result.pop()
            else:
                result.append(c)
        return "".join(result)
```

before-and-after-puzzle.py

```
# before-and-after-puzzle is not found.
# Time:  O(l * rlogr)  , l is the max length of phrases
#                      , r is the number of result, could be up to O(n^2)
# Space: O(l * (n + r)), n is the number of phrases

import collections


class Solution(object):
    def beforeAndAfterPuzzles(self, phrases):
        """
        :type phrases: List[str]
        :rtype: List[str]
        """
        lookup = collections.defaultdict(list)
        for i, phrase in enumerate(phrases):
            right = phrase.rfind(' ')
            word = phrase if right == -1 else phrase[right+1:]
            lookup[word].append(i)

        result_set = set()
        for i, phrase in enumerate(phrases):
            left = phrase.find(' ')
            word = phrase if left == -1 else phrase[:left]
            if word not in lookup:
                continue
            for j in lookup[word]:
                if j == i:
                    continue
                result_set.add(phrases[j] + phrase[len(word):])
        return sorted(result_set)
```

reverse-linked-list-ii.py

```
# DESC
# Example:
# Note: 1  m  n  length of list.
# Reverse a linked list from position m to n. Do it in one-pass.

# NOTE
#

# 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
```

unique-morse-code-words.py

```
# DESC
# Return the number of different transformations among all words we have.
# For convenience, the full table for the 26 letters of the English alphabet is gi
# ven below:
# International Morse Code defines a standard encoding where each letter is mapped
#  to a series of dots and dashes, as follows: "a" maps to ".-", "b" maps to "-...
# ", "c" maps to "-.-.", and so on.
# Now, given a list of words, each word can be written as a concatenation of the M
# orse code of each letter. For example, "cab" can be written as "-.-..--...", (wh
# ich is the concatenation "-.-." + ".-" + "-..."). We'll call such a concatenatio
# n, the transformation of a word.
# Note:

# NOTE
# Each words[i] will have length in range [1, 12].
# words[i] will only consist of lowercase letters.
# The length of words will be at most 100.

# EXAMPLE
# [".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..","--",
# "-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.--","--.."
# ]
# Example:
# Input: words = ["gin", "zen", "gig", "msg"]
# Output: 2
# Explanation:
# The
#  transformation of each word is:
# "gin" -> "--...-."
# "zen" -> "--...-."
# "gig" ->
# "--...--."
# "msg" -> "--...--."
#
# There are 2 different transformations, "--...-."
#  and "--...--.".

# Time:  O(n), n is the sume of all word lengths
# Space: O(n)

class Solution(object):
    def uniqueMorseRepresentations(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        MORSE = [".-", "-...", "-.-.", "-..", ".", "..-.", "--.",
                 "....", "..", ".---", "-.-", ".-..", "--", "-.",
                 "---", ".--.", "--.-", ".-.", "...", "-", "..-",
                 "...-", ".--", "-..-", "-.--", "--.."]

        lookup = {"".join(MORSE[ord(c) - ord('a')] for c in word) \
                  for word in words}
        return len(lookup)
```

sum-of-subsequence-widths.py

```
# DESC
# Return the sum of the widths of all subsequences of A.
# Note:
# Example 1:
# Given an array of integers A, consider all non-empty subsequences of A.
# As the answer may be very large, return the answer modulo 10^9 + 7.
# For any sequence S, let the width of S be the difference between the maximum and
#  minimum element of S.

# NOTE
# 1 <= A[i] <= 20000
# 1 <= A.length <= 20000

# EXAMPLE
# Input: [2,1,3]
# Output: 6
# Explanation:
# Subsequences are [1], [2], [3], [2,1], [2,
# 3], [1,3], [2,1,3].
# The corresponding widths are 0, 0, 0, 1, 1, 2, 2.
# The sum of
#  these widths is 6.

# Time:  O(n)
# Spce:  O(1)

class Solution(object):
    def sumSubseqWidths(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        M = 10**9+7
        # sum(A[i] * (2^i - 2^(len(A)-1-i))), i = 0..len(A)-1
        # <=>
        # sum(((A[i] - A[len(A)-1-i]) * 2^i), i = 0..len(A)-1
        result, c = 0, 1
        A.sort()
        for i in xrange(len(A)):
            result = (result + (A[i]-A[len(A)-1-i])*c % M) % M
            c = (c<<1) % M
        return result
```

validate-binary-tree-nodes.py

```python
# validate-binary-tree-nodes is not found.
# 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

```
# DESC
# A robot is located at the top-left corner of a m x n grid (marked 'Start' in the
#  diagram below).
# Note: m and n will be at most 100.
# An obstacle and empty space is marked as 1 and 0 respectively in the grid.
# Example 1:
# Now consider if some obstacles are added to the grids. How many unique paths wou
# ld there be?
# 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 diag
# ram below).

# NOTE
#

# EXAMPLE
# Input:
# [
#   [0,0,0],
#   [0,1,0],
#   [0,0,0]
# ]
# Output: 2
# Explanation:
# There is one o
# bstacle in the middle of the 3x3 grid above.
# There are two ways to reach the bot
# tom-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]
```

reverse-words-in-a-string-iii.py

```
# DESC
# Note:
# In the string, each word is separated by single space and there will not b
# e any extra space in the string.
# Given a string, you need to reverse the order of characters in each word within
# a sentence while still preserving whitespace and initial word order.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: "Let's take LeetCode contest"
# Output: "s'teL ekat edoCteeL tsetnoc"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reverseWords(self, s):
        """
        :type s: str
        :rtype: str
        """
        def reverse(s, begin, end):
            for i in xrange((end - begin) // 2):
                s[begin + i], s[end - 1 - i] = s[end - 1 - i], s[begin + i]

        s, i = list(s), 0
        for j in xrange(len(s) + 1):
            if j == len(s) or s[j] == ' ':
                reverse(s, i, j)
                i = j + 1
        return "".join(s)


class Solution2(object):
    def reverseWords(self, s):
        reversed_words = [word[::-1] for word in s.split(' ')]
        return ' '.join(reversed_words)
```

subrectangle-queries.py

```
# subrectangle-queries is not found.
# Time:  ctor:   O(1)
#        update: O(1)
#        get:    O(u), u is the number of updates
# Space: O(u)

class SubrectangleQueries(object):

    def __init__(self, rectangle):
        """
        :type rectangle: List[List[int]]
        """
        self.__rectangle = rectangle
        self.__updates = []


    def updateSubrectangle(self, row1, col1, row2, col2, newValue):
        """
        :type row1: int
        :type col1: int
        :type row2: int
        :type col2: int
        :type newValue: int
        :rtype: None
        """
        self.__updates.append((row1, col1, row2, col2, newValue))

    def getValue(self, row, col):
        """
        :type row: int
        :type col: int
        :rtype: int
        """
        for (row1, col1, row2, col2, newValue) in reversed(self.__updates):
            if row1 <= row <= row2 and col1 <= col <= col2:
                return newValue
        return self.__rectangle[row][col]


# Time:  ctor:   O(1)
#        update: O(m * n)
#        get:    O(1)
# Space: O(1)
class SubrectangleQueries2(object):

    def __init__(self, rectangle):
        """
        :type rectangle: List[List[int]]
        """
        self.__rectangle = rectangle


    def updateSubrectangle(self, row1, col1, row2, col2, newValue):
        """
        :type row1: int
        :type col1: int
        :type row2: int
        :type col2: int
```

```
        :type newValue: int
        :rtype: None
        """
        for r in xrange(row1, row2+1):
            for c in xrange(col1, col2+1):
                self.__rectangle[r][c] = newValue

    def getValue(self, row, col):
        """
        :type row: int
        :type col: int
        :rtype: int
        """
        return self.__rectangle[row][col]
```

longest-absolute-file-path.py

```
# DESC
# The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:
# Suppose we abstract our file system by a string in the following manner:
# Given a string representing the file system in the above format, return the leng
# th of the longest absolute path to file in the abstracted file system. If there
# is no file in the system, return 0.
# The directory dir contains an empty sub-directory subdir1 and a sub-directory su
# bdir2 containing a file file.ext.
# Time complexity required: O(n) where n is the size of the input string.
# Note:
# "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext"
# The directory dir contains two sub-directories subdir1 and subdir2. subdir1 cont
# ains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir
# 2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.
# The string "dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsub
# dir2\n\t\t\tfile2.ext" represents:
# We are interested in finding the longest (number of characters) absolute path to
#  a file within our file system. For example, in the second example above, the lo
# ngest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32
# (not including the double quotes).
# Notice that a/aa/aaa/file1.txt is not the longest file path, if there is another
#  path aaaaaaaaaaaaaaaaaaaaa/sth.png.

# NOTE
# The name of a file contains at least a . and an extension.
# The name of a directory or sub-directory will not contain a ..

# EXAMPLE
# dir
#     subdir1
#     subdir2
#         file.ext
# dir
#     subdir1
#         file1.ext
#         subsubdir1
#     subdir2
#         subsubd
# ir2
#             file2.ext

# 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

```
# DESC
# Example 1:
# For a given query word, the spell checker handles two categories of spelling mis
# takes:
# In addition, the spell checker operates under the following precedence rules:
# Note:
# Given a wordlist, we want to implement a spellchecker that converts a query word
#  into a correct word.
# Given some queries, return a list of words answer, where answer[i] is the correc
# t word for query = queries[i].
# query

# NOTE
# If the query has no matches in the wordlist, you should return the empty string.
# Example: wordlist = ["YellOw"], query = "yllw": correct = "" (no match)
# 1 <= queries.length <= 5000
# Example: wordlist = ["Yellow"], query = "yellow": correct = "Yellow"
# Vowel Errors: If after replacing the vowels ('a', 'e', 'i', 'o', 'u') of the que
# ry word with any vowel individually, it matches a word in the wordlist (case-ins
# ensitive), then the query word is returned with the same case as the match in th
# e wordlist.
#
#                 Example: wordlist = ["YellOw"], query = "yollow": correct = "Yel
# lOw"
#                 Example: wordlist = ["YellOw"], query = "yeellow": correct = "" (no match
# )
#                 Example: wordlist = ["YellOw"], query = "yllw": correct = "" (no match)
# 1 <= wordlist.length <= 5000
# When the query matches a word up to capitlization, you should return the first s
# uch match in the wordlist.
# Example: wordlist = ["yellow"], query = "YellOw": correct = "yellow"
# Example: wordlist = ["yellow"], query = "yellow": correct = "yellow"
# Example: wordlist = ["YellOw"], query = "yeellow": correct = "" (no match)
# 1 <= queries[i].length <= 7
# 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"
#                 Exam
# ple: wordlist = ["Yellow"], query = "yellow": correct = "Yellow"
#                 Example: word
# list = ["yellow"], query = "yellow": correct = "yellow"
# 1 <= wordlist[i].length <= 7
# All strings in wordlist and queries consist only of english letters.
# When the query matches a word up to vowel errors, you should return the first su
# ch match in the wordlist.
# Example: wordlist = ["YellOw"], query = "yollow": correct = "YellOw"
# When the query exactly matches a word in the wordlist (case-sensitive), you shou
# ld return the same word back.

# EXAMPLE
# 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"]
```

```
# 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)
```

reverse-integer.py

```
# DESC
# Example 2:
# Given a 32-bit signed integer, reverse digits of an integer.
# Example 3:
# Example 1:
# Note:
#
# 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 th
# is problem, assume that your function returns 0 when the reversed integer overfl
# ows.

# NOTE
#

# EXAMPLE
# Input: 120
# Output: 21
# Input: 123
# Output: 321
# Input: -123
# Output: -321

# Time:  O(logn) = O(1)
# Space: O(1)

class Solution(object):
    def reverse(self, x):
        """
        :type x: int
        :rtype: int
        """
        if x < 0:
            return -self.reverse(-x)

        result = 0
        while x:
            result = result * 10 + x % 10
            x //= 10
        return result if result <= 0x7fffffff else 0  # Handle overflow.

    def reverse2(self, x):
        """
        :type x: int
        :rtype: int
        """
        if x < 0:
            x = int(str(x)[::-1][-1] + str(x)[::-1][:-1])
        else:
            x = int(str(x)[::-1])
        x = 0 if abs(x) > 0x7FFFFFFF else x
        return x

    def reverse3(self, x):
        """
        :type x: int
        :rtype: int
        """
```

```
s = cmp(x, 0)
r = int(repr(s * x)[::-1])
return s * r * (r < 2 ** 31)
```

remove-comments.py

```
# DESC
# In C++, there are two types of comments, line comments, and block comments.
# Example 1:
# After removing the comments from the source code, return the source code in the
# same format.
# Example 2:
# Given a C++ program, remove comments from it. The program source is an array whe
# re source[i] is the i-th line of the source code.  This represents the result of
#  splitting the original source code string by the newline character \n.
# There will be no control characters, single quote, or double quote characters.
# For example, source = "string s = "/* Not a comment. */";" will not be a test ca
# se.  (Also, nothing else such as defines or macros will interfere with the comme
# nts.)
# It is guaranteed that every open block comment will eventually be closed, so /*
# outside of a line or block comment always starts a new comment.
# Finally, implicit newline characters can be deleted by block comments.  Please s
# ee the examples below for details.
# The string // denotes a line comment, which represents that it and rest of the c
# haracters to the right of it in the same line should be ignored.
# If a certain line of code is empty after removing comments, you must not output
# that line: each string in the answer list will be non-empty.
# Note:
# The string /* denotes a block comment, which represents that all characters unti
# l the next (non-overlapping) occurrence of */ should be ignored.  (Here, occurre
# nces happen in reading order: line by line from left to right.)  To be clear, th
# e string /*/ does not yet end the block comment, as the ending would be overlapp
# ing the beginning.
# The first effective comment takes precedence over others: if the string // occur
# s in a block comment, it is ignored. Similarly, if the string /* occurs in a lin
# e or block comment, it is also ignored.

# NOTE
# The length of source is in the range [1, 100].
# The length of source[i] is in the range [0, 80].
# There are no single-quote, double-quote, or control characters in the source code.
# Every open block comment is eventually closed.

# EXAMPLE
# Input:
# source = ["a/*comment", "line", "more_comment*/b"]
# Output: ["ab"]
# Explan
# ation: The original source string is "a/*comment\nline\nmore_comment*/b", where
# we have bolded the newline characters.  After deletion, the implicit newline cha
# racters are deleted, leaving the string "ab", which when delimited by newline ch
# aracters becomes ["ab"].
# Input:
# source = ["/*Test program */", "int main()", "{ ", "  // variable declar
# ation ", "int a, b, c;", "/* This is a test", "   multiline  ", "   comment for
# ", "   testing */", "a = b + c;", "}"]
#
# The line by line code is visualized as b
# elow:
# /*Test program */
# int main()
# {
#   // variable declaration
# int a, b, c;
```

```
# /*
#  This is a test
#    multiline
#    comment for
#    testing */
# a = b + c;
# }
#
# Outpu
# t: ["int main()","{ ","  ","int a, b, c;","a = b + c;","}"]
#
# The line by line co
# de is visualized as below:
# int main()
# {
#
# int a, b, c;
# a = b + c;
# }
#
# Explanati
# on:
# The string /* denotes a block comment, including line 1 and lines 6-9. The
# string // denotes line 4 as comments.

# Time:  O(n), n is the length of the source
# Space: O(k), k is the max length of a line

class Solution(object):
    def removeComments(self, source):
        """
        :type source: List[str]
        :rtype: List[str]
        """
        in_block = False
        result, newline = [], []
        for line in source:
            i = 0
            while i < len(line):
                if not in_block and i+1 < len(line) and line[i:i+2] == '/*':
                    in_block = True
                    i += 1
                elif in_block and i+1 < len(line) and line[i:i+2] == '*/':
                    in_block = False
                    i += 1
                elif not in_block and i+1 < len(line) and line[i:i+2] == '//':
                    break
                elif not in_block:
                    newline.append(line[i])
                i += 1
            if newline and not in_block:
                result.append("".join(newline))
                newline = []
        return result
```

shuffle-an-array.py

```
# shuffle-an-arra is not found.
# 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
```

sliding-puzzle.py

```
# DESC
# Given a puzzle board, return the least number of moves required so that the stat
# e of the board is solved. If it is impossible for the state of the board to be s
# olved, return -1.
# Examples:
# On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and a
# n empty square represented by 0.
# The state of the board is solved if and only if the board is [[1,2,3],[4,5,0]].
# Note:
# A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

# NOTE
# board will be a 2 x 3 array as described above.
# board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

# EXAMPLE
# Input: board = [[1,2,3],[4,0,5]]
# Output: 1
# Explanation: Swap the 0 and the 5 in
# one move.
# Input: board = [[3,2,4],[1,5,0]]
# Output: 14
# Input: board = [[1,2,3],[5,4,0]]
# Output: -1
# Explanation: No number of moves will
#  make the board solved.
# Input: board = [[4,1,2],[5,0,3]]
# Output: 5
# Explanation: 5 is the smallest number
#  of moves that solves the board.
# An example path:
# After move 0: [[4,1,2],[5,0,3]
# ]
# After move 1: [[4,1,2],[0,5,3]]
# After move 2: [[0,1,2],[4,5,3]]
# After move 3:
# [[1,0,2],[4,5,3]]
# After move 4: [[1,2,0],[4,5,3]]
# After move 5: [[1,2,3],[4,5,0]
# ]

# Time:  O((m * n) * (m * n)!)
# Space: O((m * n) * (m * n)!)

import heapq
import itertools


# A* Search Algorithm
class Solution(object):
    def slidingPuzzle(self, board):
        """
        :type board: List[List[int]]
        :rtype: int
        """
        def dot(p1, p2):
            return p1[0]*p2[0]+p1[1]*p2[1]
```

```
        def heuristic_estimate(board, R, C, expected):
            result = 0
            for i in xrange(R):
                for j in xrange(C):
                    val = board[C*i + j]
                    if val == 0: continue
                    r, c = expected[val]
                    result += abs(r-i) + abs(c-j)
            return result

        R, C = len(board), len(board[0])
        begin = tuple(itertools.chain(*board))
        end = tuple(range(1, R*C) + [0])
        expected = {(C*i+j+1) % (R*C) : (i, j)
                    for i in xrange(R) for j in xrange(C)}

        min_steps = heuristic_estimate(begin, R, C, expected)
        closer, detour = [(begin.index(0), begin)], []
        lookup = set()
        while True:
            if not closer:
                if not detour:
                    return -1
                min_steps += 2
                closer, detour = detour, closer
            zero, board = closer.pop()
            if board == end:
                return min_steps
            if board not in lookup:
                lookup.add(board)
                r, c = divmod(zero, C)
                for direction in ((-1, 0), (1, 0), (0, -1), (0, 1)):
                    i, j = r+direction[0], c+direction[1]
                    if 0 <= i < R and 0 <= j < C:
                        new_zero = i*C+j
                        tmp = list(board)
                        tmp[zero], tmp[new_zero] = tmp[new_zero], tmp[zero]
                        new_board = tuple(tmp)
                        r2, c2 = expected[board[new_zero]]
                        r1, c1 = divmod(zero, C)
                        r0, c0 = divmod(new_zero, C)
                        is_closer = dot((r1-r0, c1-c0), (r2-r0, c2-c0)) > 0
                        (closer if is_closer else detour).append((new_zero, new_board))
        return min_steps


# Time:  O((m * n) * (m * n)! * log((m * n)!))
# Space: O((m * n) * (m * n)!)
# A* Search Algorithm
class Solution2(object):
    def slidingPuzzle(self, board):
        """
        :type board: List[List[int]]
        :rtype: int
        """
        def heuristic_estimate(board, R, C, expected):
            result = 0
            for i in xrange(R):
                for j in xrange(C):
                    val = board[C*i + j]
```

```
            if val == 0: continue
            r, c = expected[val]
            result += abs(r-i) + abs(c-j)
    return result

R, C = len(board), len(board[0])
begin = tuple(itertools.chain(*board))
end = tuple(range(1, R*C) + [0])
end_wrong = tuple(range(1, R*C-2) + [R*C-1, R*C-2, 0])
expected = {(C*i+j+1) % (R*C) : (i, j)
            for i in xrange(R) for j in xrange(C)}

min_heap = [(0, 0, begin.index(0), begin)]
lookup = {begin: 0}
while min_heap:
    f, g, zero, board = heapq.heappop(min_heap)
    if board == end: return g
    if board == end_wrong: return -1
    if f > lookup[board]: continue

    r, c = divmod(zero, C)
    for direction in ((-1, 0), (1, 0), (0, -1), (0, 1)):
        i, j = r+direction[0], c+direction[1]
        if 0 <= i < R and 0 <= j < C:
            new_zero = C*i+j
            tmp = list(board)
            tmp[zero], tmp[new_zero] = tmp[new_zero], tmp[zero]
            new_board = tuple(tmp)
            f = g+1+heuristic_estimate(new_board, R, C, expected)
            if f < lookup.get(new_board, float("inf")):
                lookup[new_board] = f
                heapq.heappush(min_heap, (f, g+1, new_zero, new_board))
return -1
```

cat-and-mouse.py

```
# DESC
# Then, the game can end in 3 ways:
# Example 1:
# Given a graph, and assuming both players play optimally, return 1 if the game is
#  won by Mouse, 2 if the game is won by Cat, and 0 if the game is a draw.
# A game on an undirected graph is played by two players, Mouse and Cat, who alter
# nate turns.
# Mouse starts at node 1 and goes first, Cat starts at node 2 and goes second, and
#  there is a Hole at node 0.
# During each player's turn, they must travel along one edge of the graph that mee
# ts where they are.  For example, if the Mouse is at node 1, it must travel to an
# y node in graph[1].
# Additionally, it is not allowed for the Cat to travel to the Hole (node 0.)
# graph[a]
# The graph is given as follows: graph[a] is a list of all nodes b such that ab is
#  an edge of the graph.
# Note:

# NOTE
# If ever a position is repeated (ie. the players are in the same position as a pr
# evious turn, and it is the same player's turn to move), the game is a draw.
# If ever the Cat occupies the same node as the Mouse, the Cat wins.
# 3 <= graph.length <= 50
# It is guaranteed that graph[2] contains a non-zero element.
# If ever the Mouse reaches the Hole, the Mouse wins.
# It is guaranteed that graph[1] is non-empty.

# EXAMPLE
# Input: [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
# Output: 0
# Explanation:
# 4---3---
# 1
# |   |
# 2---5
#  \ /
#   0

# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def catMouseGame(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: int
        """
        HOLE, MOUSE_START, CAT_START = range(3)
        DRAW, MOUSE, CAT = range(3)

        def move(graph, lookup, i, other_i, is_mouse_turn):
            key = (i, other_i, is_mouse_turn)
            if key in lookup:
                return lookup[key]

            lookup[key] = DRAW
            if is_mouse_turn:
                skip, target, win, lose = other_i, HOLE, MOUSE, CAT
```

```
        else:
            skip, target, win, lose = HOLE, other_i, CAT, MOUSE
        for nei in graph[i]:
            if nei == target:
                result = win
                break
        else:
            result = lose
            for nei in graph[i]:
                if nei == skip:
                    continue
                tmp = move(graph, lookup, other_i, nei, not is_mouse_turn)
                if tmp == win:
                    result = win
                    break
                if tmp == DRAW:
                    result = DRAW
        lookup[key] = result
        return result

    return move(graph, {}, MOUSE_START, CAT_START, True)
```

palindrome-permutation-ii.py

```
# palindrome-permutation-ii is not found.
# Time:  O(n * n!)
# Space: O(n)

import collections
import itertools


class Solution(object):
    def generatePalindromes(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        cnt = collections.Counter(s)
        mid = ''.join(k for k, v in cnt.iteritems() if v % 2)
        chars = ''.join(k * (v / 2) for k, v in cnt.iteritems())
        return self.permuteUnique(mid, chars) if len(mid) < 2 else []

    def permuteUnique(self, mid, nums):
        result = []
        used = [False] * len(nums)
        self.permuteUniqueRecu(mid, result, used, [], nums)
        return result

    def permuteUniqueRecu(self, mid, result, used, cur, nums):
        if len(cur) == len(nums):
            half_palindrome = ''.join(cur)
            result.append(half_palindrome + mid + half_palindrome[::-1])
            return
        for i in xrange(len(nums)):
            if not used[i] and not (i > 0 and nums[i-1] == nums[i] and used[i-1]):
                used[i] = True
                cur.append(nums[i])
                self.permuteUniqueRecu(mid, result, used, cur, nums)
                cur.pop()
                used[i] = False

class Solution2(object):
    def generatePalindromes(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        cnt = collections.Counter(s)
        mid = tuple(k for k, v in cnt.iteritems() if v % 2)
        chars = ''.join(k * (v / 2) for k, v in cnt.iteritems())
        return [''.join(half_palindrome + mid + half_palindrome[::-1]) \
                for half_palindrome in set(itertools.permutations(chars))] if len(mid) < 2 else []
```

isomorphic-strings.py

```
# DESC
# Given two strings s and t, determine if they are isomorphic.
# All occurrences of a character must be replaced with another character while pre
# serving the order of characters. No two characters may map to the same character
#  but a character may map to itself.
# Note:
#
# You may assume both s and t have the same length.
# Two strings are isomorphic if the characters in s can be replaced to get t.
# Example 2:
# Example 1:
# Example 3:

# NOTE
#

# EXAMPLE
# Input: s = "foo", t = "bar"
# Output: false
# Input: s = "egg", t = "add"
# Output: true
# Input: s = "paper", t = "title"
# Output: true

# Time:  O(n)
# Space: O(1)

from itertools import izip  # Generator version of zip.

class Solution(object):
    def isIsomorphic(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        if len(s) != len(t):
            return False

        s2t, t2s = {}, {}
        for p, w in izip(s, t):
            if w not in s2t and p not in t2s:
                s2t[w] = p
                t2s[p] = w
            elif w not in s2t or s2t[w] != p:
                # Contradict mapping.
                return False
        return True


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def isIsomorphic(self, s, t):
        if len(s) != len(t):
            return False

        return self.halfIsom(s, t) and self.halfIsom(t, s)
```

```
def halfIsom(self, s, t):
    lookup = {}
    for i in xrange(len(s)):
        if s[i] not in lookup:
            lookup[s[i]] = t[i]
        elif lookup[s[i]] != t[i]:
            return False
    return True
```

binary-trees-with-factors.py

```
# DESC
# Example 2:
# Given an array of unique integers, each integer is strictly greater than 1.
# Each non-leaf node's value should be equal to the product of the values of it's
# children.
# How many binary trees can we make?  Return the answer modulo 10 ** 9 + 7.
# Example 1:
# Note:
# We make a binary tree using these integers and each number may be used for any n
# umber of times.

# NOTE
# 1 <= A.length <= 1000.
# 2 <= A[i] <= 10 ^ 9.

# EXAMPLE
# Input: A = [2, 4, 5, 10]
# Output: 7
# Explanation: We can make these trees: [2], [4
# ], [5], [10], [4, 2, 2], [10, 2, 5], [10, 5, 2].
# Input: A = [2, 4]
# Output: 3
# Explanation: We can make these trees: [2], [4], [4, 2, 2]

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


class Solution(object):
    def numFactoredBinaryTrees(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        M = 10**9 + 7
        A.sort()
        dp = {}
        for i in xrange(len(A)):
            dp[A[i]] = 1
            for j in xrange(i):
                if A[i] % A[j] == 0 and A[i] // A[j] in dp:
                    dp[A[i]] += dp[A[j]] * dp[A[i] // A[j]]
                    dp[A[i]] %= M
        return sum(dp.values()) % M
```

palindrome-linked-list.py

```
# DESC
# Example 1:
# Given a singly linked list, determine if it is a palindrome.
# Follow up:
#
# Could you do it in O(n) time and O(1) space?
# Example 2:

# NOTE
#

# EXAMPLE
# Input: 1->2->2->1
# Output: true
# Input: 1->2
# Output: false

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {ListNode} head
    # @return {boolean}
    def isPalindrome(self, head):
        reverse, fast = None, head
        # Reverse the first half part of the list.
        while fast and fast.next:
            fast = fast.next.next
            head.next, reverse, head = reverse, head, head.next

        # If the number of the nodes is odd,
        # set the head of the tail list to the next of the median node.
        tail = head.next if fast else head

        # Compare the reversed first half list with the second half list.
        # And restore the reversed first half list.
        is_palindrome = True
        while reverse:
            is_palindrome = is_palindrome and reverse.val == tail.val
            reverse.next, head, reverse = head, reverse, reverse.next
            tail = tail.next

        return is_palindrome
```

contains-duplicate-ii.py

```python
# DESC
# Given an array of integers and an integer k, find out whether there are two dist
# inct indices i and j in the array such that nums[i] = nums[j] and the absolute d
# ifference between i and j is at most k.
# Example 3:
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: nums = [1,0,1,1], k = 1
# Output: true
# Input: nums = [1,2,3,1], k = 3
# Output: true
# Input: nums = [1,2,3,1,2,3], k = 2
# Output: false

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

class Solution(object):
    # @param {integer[]} nums
    # @param {integer} k
    # @return {boolean}
    def containsNearbyDuplicate(self, nums, k):
        lookup = {}
        for i, num in enumerate(nums):
            if num not in lookup:
                lookup[num] = i
            else:
                # If the value occurs before, check the difference.
                if i - lookup[num] <= k:
                    return True
                # Update the index of the value.
                lookup[num] = i
        return False
```

minimum-falling-path-sum.py

```
# DESC
# Given a square array of integers A, we want the minimum sum of a falling path th
# rough A.
# A falling path starts at any element in the first row, and chooses one element f
# rom 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:
# The falling path with the smallest sum is [1,4,7], so the answer is 12.
# Constraints:

# NOTE
# [3,5,7], [3,5,8], [3,5,9], [3,6,8], [3,6,9]
# [2,4,7], [2,4,8], [2,5,7], [2,5,8], [2,5,9], [2,6,8], [2,6,9]
# 1 <= A.length == A[0].length <= 100
# [1,4,7], [1,4,8], [1,5,7], [1,5,8], [1,5,9]
# -100 <= A[i][j] <= 100

# EXAMPLE
# Input: [[1,2,3],[4,5,6],[7,8,9]]
# Output: 12
# Explanation:
# The possible falling p
# aths are:

# 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

```
# DESC
# Example 3:
# The i-th person has weight people[i], and each boat can carry a maximum weight o
# f limit.
# Note:
# Example 1:
# limit
# Return the minimum number of boats to carry every given person.  (It is guarante
# ed each person can be carried by a boat.)
# Example 2:
# Each boat carries at most 2 people at the same time, provided the sum of the wei
# ght of those people is at most limit.

# NOTE
# 1 <= people.length <= 50000
# 1 <= people[i] <= limit <= 30000

# EXAMPLE
# Input: people = [1,2], limit = 3
# Output: 1
# Explanation: 1 boat (1, 2)
# Input: people = [3,5,3,4], limit = 5
# Output: 4
# Explanation: 4 boats (3), (3), (4
# ), (5)
# Input: people = [3,2,2,1], limit = 3
# Output: 3
# Explanation: 3 boats (1, 2), (2)
# and (3)

# 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
```

house-robber.py

```
# DESC
# You are a professional robber planning to rob houses along a street. Each house
# has a certain amount of money stashed, the only constraint stopping you from rob
# bing each of them is that adjacent houses have security system connected and it
# will automatically contact the police if two adjacent houses were broken into on
#  the same night.
# Example 2:
# Given a list of non-negative integers representing the amount of money of each h
# ouse, determine the maximum amount of money you can rob tonight without alerting
#  the police.
# Example 1:
# Constraints:

# NOTE
# 0 <= nums[i] <= 400
# 0 <= nums.length <= 100

# EXAMPLE
# Input: nums = [1,2,3,1]
# Output: 4
# Explanation: Rob house 1 (money = 1) and then
# rob house 3 (money = 3).
#              Total amount you can rob = 1 + 3 = 4.
# Input: nums = [2,7,9,3,1]
# Output: 12
# Explanation: Rob house 1 (money = 2), rob h
# ouse 3 (money = 9) and rob house 5 (money = 1).
#              Total amount you ca
# n rob = 2 + 9 + 1 = 12.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param num, a list of integer
    # @return an integer
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        last, now = 0, 0
        for i in nums:
            last, now = now, max(last + i, now)
        return now
```

time-needed-to-inform-all-employees.py

```
# time-needed-to-inform-all-employees is not found.
# Time:  O(n)
# Space: O(n)

import collections


# dfs solution with stack
class Solution(object):
    def numOfMinutes(self, n, headID, manager, informTime):
        """
        :type n: int
        :type headID: int
        :type manager: List[int]
        :type informTime: List[int]
        :rtype: int
        """
        children = collections.defaultdict(list)
        for child, parent in enumerate(manager):
            if parent != -1:
                children[parent].append(child)

        result = 0
        stk = [(headID, 0)]
        while stk:
            node, curr = stk.pop()
            curr += informTime[node]
            result = max(result, curr)
            if node not in children:
                continue
            for c in children[node]:
                stk.append((c, curr))
        return result


# Time:  O(n)
# Space: O(n)
# dfs solution with recursion
class Solution2(object):
    def numOfMinutes(self, n, headID, manager, informTime):
        """
        :type n: int
        :type headID: int
        :type manager: List[int]
        :type informTime: List[int]
        :rtype: int
        """
        def dfs(informTime, children, node):
            return (max(dfs(informTime, children, c)
                        for c in children[node])
                    if node in children
                    else 0) + informTime[node]

        children = collections.defaultdict(list)
        for child, parent in enumerate(manager):
            if parent != -1:
                children[parent].append(child)
        return dfs(informTime, children, headID)
```

sort-items-by-groups-respecting-dependencies.py

```python
# sort-items-by-groups-respecting-dependencies is not found.
# Time:  O(n + e)
# Space: O(n + e)

import collections


class Topo(object):
    def __init__(self):
        self.__nodes = set()
        self.__in_degree = collections.defaultdict(set)
        self.__out_degree = collections.defaultdict(set)

    def add_node(self, node):
        self.__nodes.add(node)

    def add_edge(self, src, dst):
        self.add_node(src), self.add_node(dst)
        self.__in_degree[dst].add(src)
        self.__out_degree[src].add(dst)

    def sort(self):
        q = collections.deque()
        result = []
        for node in self.__nodes:
            if node not in self.__in_degree:
                q.append(node)
        while q:
            node = q.popleft()
            result.append(node)
            for nei in self.__out_degree[node]:
                self.__in_degree[nei].remove(node)
                if not self.__in_degree[nei]:
                    self.__in_degree.pop(nei)
                    q.append(nei)
        if len(result) < len(self.__nodes):
            return
        return result


class Solution(object):
    def sortItems(self, n, m, group, beforeItems):
        """
        :type n: int
        :type m: int
        :type group: List[int]
        :type beforeItems: List[List[int]]
        :rtype: List[int]
        """
        for i in xrange(n):
            if group[i] == -1:
                group[i] = m
                m += 1
        global_group = Topo()
        for i in xrange(m):
            global_group.add_node(i)
        local_groups = collections.defaultdict(Topo)
        for i in xrange(n):
```

```
    local_groups[group[i]].add_node(i)
for i in xrange(n):
    for j in beforeItems[i]:
        if group[i] == group[j]:
            local_groups[group[i]].add_edge(j, i)
        else:
            global_group.add_edge(group[j], group[i]);
result = []
global_order = global_group.sort()
if global_order is None:
    return []
for i in global_order:
    local_order = local_groups[i].sort();
    if local_order is None:
        return []
    for x in local_order:
        result.append(x)
return result
```

sum-root-to-leaf-numbers.py

```
# DESC
# Given a binary tree containing digits from 0-9 only, each root-to-leaf path coul
# d represent a number.
# Note: A leaf is a node with no children.
# Find the total sum of all root-to-leaf numbers.
# Example 2:
# An example is the root-to-leaf path 1->2->3 which represents the number 123.
# 1->2->3
# Example:

# NOTE
#

# EXAMPLE
# Input: [1,2,3]
#     1
#    / \
#   2   3
# Output: 25
# Explanation:
# The root-to-leaf pat
# h 1->2 represents the number 12.
# The root-to-leaf path 1->3 represents the numbe
# r 13.
# Therefore, sum = 12 + 13 = 25.
# 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
```

delete-and-earn.py

```
# DESC
# Example 1:
# Example 2:
# Note:
# nums[i]
# In each operation, you pick any nums[i] and delete it to earn nums[i] points. Af
# ter, you must delete every element equal to nums[i] - 1 or nums[i] + 1.
# You start with 0 points. Return the maximum number of points you can earn by app
# lying such operations.
# Given an array nums of integers, you can perform operations on the array.

# NOTE
# Each element nums[i] is an integer in the range [1, 10000].
# The length of nums is at most 20000.

# EXAMPLE
# Input: nums = [2, 2, 3, 3, 3, 4]
# Output: 9
# Explanation:
# Delete 3 to earn 3 poin
# ts, deleting both 2's and the 4.
# Then, delete 3 again to earn 3 points, and 3 ag
# ain to earn 3 points.
# 9 total points are earned.
# Input: nums = [3, 4, 2]
# Output: 6
# Explanation:
# Delete 4 to earn 4 points, conse
# quently 3 is also deleted.
# Then, delete 2 to earn 2 points. 6 total points are e
# arned.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def deleteAndEarn(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        vals = [0] * 10001
        for num in nums:
            vals[num] += num
        val_i, val_i_1 = vals[0], 0
        for i in xrange(1, len(vals)):
            val_i_1, val_i_2 = val_i, val_i_1
            val_i = max(vals[i] + val_i_2, val_i_1)
        return val_i
```

first-missing-positive.py

```
# DESC
# Example 1:
# Your algorithm should run in O(n) time and uses constant extra space.
# Example 2:
# Given an unsorted integer array, find the smallest missing positive integer.
# Follow up:
# Example 3:

# NOTE
#

# EXAMPLE
# Input: [3,4,-1,1]
# Output: 2
# Input: [7,8,9,11,12]
# Output: 1
# Input: [1,2,0]
# Output: 3

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A, a list of integers
    # @return an integer
    def firstMissingPositive(self, A):
        i = 0
        while i < len(A):
            if A[i] > 0 and A[i] - 1 < len(A) and A[i] != A[A[i]-1]:
                A[A[i]-1], A[i] = A[i], A[A[i]-1]
            else:
                i += 1

        for i, integer in enumerate(A):
            if integer != i + 1:
                return i + 1
        return len(A) + 1
```

different-ways-to-add-parentheses.py

```
# DESC
# Example 2:
# Given a string of numbers and operators, return all possible results from comput
# ing all the different possible ways to group numbers and operators. The valid op
# erators are +, - and *.
# Example 1:

# NOTE
#

# EXAMPLE
# 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
# Input: "2-1-1"
# Output: [0, 2]
# Explanation:
# ((2-1)-1) = 0
# (2-(1-1)) = 2

# 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))
```

split-array-with-equal-sum.py

```
# split-array-with-equal-sum is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def splitArray(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        if len(nums) < 7:
            return False

        accumulated_sum = [0] * len(nums)
        accumulated_sum[0] = nums[0]
        for i in xrange(1, len(nums)):
            accumulated_sum[i] = accumulated_sum[i-1] + nums[i]
        for j in xrange(3, len(nums)-3):
            lookup = set()
            for i in xrange(1, j-1):
                if accumulated_sum[i-1] == accumulated_sum[j-1] - accumulated_sum[i]:
                    lookup.add(accumulated_sum[i-1])
            for k in xrange(j+2, len(nums)-1):
                if accumulated_sum[-1] - accumulated_sum[k] == accumulated_sum[k-1] - accumulated_sum[j] and \
                   accumulated_sum[k - 1] - accumulated_sum[j] in lookup:
                    return True
        return False
```

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

```
# DESC
# If no land or water exists in the grid, return -1.
# Example 2:
# (x0, y0)
# The distance used in this problem is the Manhattan distance: the distance betwee
# n two cells (x0, y0) and (x1, y1) is |x0 - x1| + |y0 - y1|.
# Note:
# 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.
# Example 1:

# NOTE
# 1 <= grid.length == grid[0].length <= 100
# grid[i][j] is 0 or 1

# EXAMPLE
# Input: [[1,0,1],[0,0,0],[1,0,1]]
# Output: 2
# Explanation:
# The cell (1, 1) is as f
# ar as possible from all the land with distance 2.
# Input: [[1,0,0],[0,0,0],[0,0,0]]
# Output: 4
# Explanation:
# The cell (2, 2) is as f
# ar as possible from all the land with distance 4.

# 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
```

cut-off-trees-for-golf-event.py

```
# DESC
# You are asked to cut off trees in a forest for a golf event. The forest is repre
# sented as a non-negative 2D map, in this map:
# In one step you can walk in any of the four directions top, bottom, left and rig
# ht also when standing in a point which is a tree you can decide whether or not t
# o cut off the tree.
# Example 3:
# You are asked to cut off all the trees in this forest in the order of tree's hei
# ght - always cut off the tree with lowest height first. And after cutting, the o
# riginal place has the tree will become a grass (value 1).
# Example 2:
# Example 1:
# You are guaranteed that no two trees have the same height and there is at least
# one tree needs to be cut off.
# Constraints:
# You will start from the point (0, 0) and you should output the minimum steps you
#  need to walk to cut off all the trees. If you can't cut off all the trees, outp
# ut -1 in that situation.

# NOTE
# 1 <= forest[i].length <= 50
# 1 <= forest.length <= 50
# 0 represents the obstacle can't be reached.
# 1 represents the ground can be walked through.
# 0 <= forest[i][j] <= 10^9
# The place with number bigger than 1 represents a tree can be walked through, and
#  this positive number represents the tree's height.

# EXAMPLE
# Input:
# [
#  [2,3,4],
#  [0,0,5],
#  [8,7,6]
# ]
# Output: 6
# Explanation: You started from
#  the point (0,0) and you can cut off the tree in (0,0) directly without walking.
# Input:
# [
#  [1,2,3],
#  [0,0,4],
#  [7,6,5]
# ]
# Output: 6
# Input:
# [
#  [1,2,3],
#  [0,0,0],
#  [7,6,5]
# ]
# Output: -1

# Time:  O(t * (logt + m * n)), t is the number of trees
# Space: O(t + m * n)

import collections
import heapq
```

```
class Solution(object):
    def cutOffTree(self, forest):
        """
        :type forest: List[List[int]]
        :rtype: int
        """
        def dot(p1, p2):
            return p1[0]*p2[0]+p1[1]*p2[1]

        def minStep(p1, p2):
            min_steps = abs(p1[0]-p2[0])+abs(p1[1]-p2[1])
            closer, detour = [p1], []
            lookup = set()
            while True:
                if not closer:  # cannot find a path in the closer expansions
                    if not detour:  # no other possible path
                        return -1
                    # try other possible paths in detour expansions with extra 2-step cost
                    min_steps += 2
                    closer, detour = detour, closer
                i, j = closer.pop()
                if (i, j) == p2:
                    return min_steps
                if (i, j) not in lookup:
                    lookup.add((i, j))
                    for I, J in (i+1, j), (i-1, j), (i, j+1), (i, j-1):
                        if 0 <= I < m and 0 <= J < n and forest[I][J] and (I, J) not in lookup:
                            is_closer = dot((I-i, J-j), (p2[0]-i, p2[1]-j)) > 0
                            (closer if is_closer else detour).append((I, J))
            return min_steps

        m, n = len(forest), len(forest[0])
        min_heap = []
        for i in xrange(m):
            for j in xrange(n):
                if forest[i][j] > 1:
                    heapq.heappush(min_heap, (forest[i][j], (i, j)))

        start = (0, 0)
        result = 0
        while min_heap:
            tree = heapq.heappop(min_heap)
            step = minStep(start, tree[1])
            if step < 0:
                return -1
            result += step
            start = tree[1]
        return result


# Time:  O(t * (logt + m * n)), t is the number of trees
# Space: O(t + m * n)
class Solution_TLE(object):
    def cutOffTree(self, forest):
        """
        :type forest: List[List[int]]
        :rtype: int
        """
```

```
def minStep(p1, p2):
    min_steps = 0
    lookup = {p1}
    q = collections.deque([p1])
    while q:
        size = len(q)
        for _ in xrange(size):
            (i, j) = q.popleft()
            if (i, j) == p2:
                return min_steps
            for i, j in (i+1, j), (i-1, j), (i, j+1), (i, j-1):
                if not (0 <= i < m and 0 <= j < n and forest[i][j] and (i, j) not in lookup):
                    continue
                q.append((i, j))
                lookup.add((i, j))
        min_steps += 1
    return -1

m, n = len(forest), len(forest[0])
min_heap = []
for i in xrange(m):
    for j in xrange(n):
        if forest[i][j] > 1:
            heapq.heappush(min_heap, (forest[i][j], (i, j)))

start = (0, 0)
result = 0
while min_heap:
    tree = heapq.heappop(min_heap)
    step = minStep(start, tree[1])
    if step < 0:
        return -1
    result += step
    start = tree[1]
return result
```

perfect-number.py

```
# DESC
# Example:
# Note:
# The input number n will not exceed 100,000,000. (1e8)
# We define the Perfect Number is a positive integer that is equal to the sum of a
# ll its positive divisors except itself.

# NOTE
#

# EXAMPLE
# Input: 28
# Output: True
# Explanation: 28 = 1 + 2 + 4 + 7 + 14

# Time:  O(sqrt(n))
# Space: O(1)

class Solution(object):
    def checkPerfectNumber(self, num):
        """
        :type num: int
        :rtype: bool
        """
        if num <= 0:
            return False

        sqrt_num = int(num ** 0.5)
        total = sum(i+num//i for i in xrange(1, sqrt_num+1) if num%i == 0)
        if sqrt_num ** 2 == num:
            total -= sqrt_num
        return total - num == num
```

broken-calculator.py

```
# DESC
# On a broken calculator that has a number showing on its display, we can perform
# two operations:
# Initially, the calculator is displaying the number X.
# Example 2:
# Example 1:
# Example 3:
# Note:
# Example 4:
# Return the minimum number of operations needed to display the number Y.

# NOTE
# Decrement: Subtract 1 from the number on the display.
# 1 <= Y <= 10^9
# 1 <= X <= 10^9
# Double: Multiply the number on the display by 2, or;

# EXAMPLE
# Input: X = 5, Y = 8
# Output: 2
# Explanation: Use decrement and then double {5 -> 4
#  -> 8}.
# Input: X = 3, Y = 10
# Output: 3
# Explanation:  Use double, decrement and double {3
#  -> 6 -> 5 -> 10}.
# Input: X = 2, Y = 3
# Output: 2
# Explanation: Use double operation and then decreme
# nt operation {2 -> 4 -> 3}.
# Input: X = 1024, Y = 1
# Output: 1023
# Explanation: Use decrement operations 1023 times.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def brokenCalc(self, X, Y):
        """
        :type X: int
        :type Y: int
        :rtype: int
        """
        result = 0
        while X < Y:
            if Y%2:
                Y += 1
            else:
                Y /= 2
            result += 1
        return result + X-Y
```

paint-house-iii.py

```
# paint-house-iii is not found.
# Time:  O(m * t * n^2)
# Space: O(t * n)

class Solution(object):
    def minCost(self, houses, cost, m, n, target):
        """
        :type houses: List[int]
        :type cost: List[List[int]]
        :type m: int
        :type n: int
        :type target: int
        :rtype: int
        """
        # dp[i][j][k]: cost of covering i+1 houses with j+1 neighbor groups and the (k+1)th color
        dp = [[[float("inf") for _ in xrange(n)] for _ in xrange(target)] for _ in xrange(2)]
        for i in xrange(m):
            dp[i%2] = [[float("inf") for _ in xrange(n)] for _ in xrange(target)]
            for j in xrange(min(target, i+1)):
                for k in xrange(n):
                    if houses[i] and houses[i]-1 != k:
                        continue
                    same = dp[(i-1)%2][j][k] if i-1 >= 0 else 0
                    diff = (min([dp[(i-1)%2][j-1][nk] for nk in xrange(n) if nk != k] or [float("inf")]) if j-
                    paint = cost[i][k] if not houses[i] else 0
                    dp[i%2][j][k] = min(same, diff)+paint
        result = min(dp[(m-1)%2][-1])
        return result if result != float("inf") else -1


# Time:  O(m * t * n^2)
# Space: O(t * n)
class Solution2(object):
    def minCost(self, houses, cost, m, n, target):
        """
        :type houses: List[int]
        :type cost: List[List[int]]
        :type m: int
        :type n: int
        :type target: int
        :rtype: int
        """
        dp = {(0, 0): 0}
        for i, p in enumerate(houses):
            new_dp = {}
            for nk in (xrange(1, n+1) if not p else [p]):
                for j, k in dp:
                    nj = j + (k != nk)
                    if nj > target:
                        continue
                    new_dp[nj, nk] = min(new_dp.get((nj, nk), float("inf")), dp[j, k] + (cost[i][nk-1] if nk !
            dp = new_dp
        return min([dp[j, k] for j, k in dp if j == target] or [-1])
```

find-the-closest-palindrome.py

```
# DESC
# Example 1:
# Given an integer n, find the closest integer (not including itself), which is a
# palindrome.
# Note:
# The 'closest' is defined as absolute difference minimized between two integers.

# NOTE
# The input n is a positive integer represented by string, whose length will not e
# xceed 18.
# If there is a tie, return the smaller one as answer.

# EXAMPLE
# Input: "123"
# Output: "121"

# Time:  O(l)
# Space: O(l)

class Solution(object):
    def nearestPalindromic(self, n):
        """
        :type n: str
        :rtype: str
        """
        l = len(n)
        candidates = set((str(10**l + 1), str(10**(l - 1) - 1)))
        prefix = int(n[:(l + 1)/2])
        for i in map(str, (prefix-1, prefix, prefix+1)):
            candidates.add(i + [i, i[:-1]][l%2][::-1])
        candidates.discard(n)
        return min(candidates, key=lambda x: (abs(int(x) - int(n)), int(x)))
```

k-th-symbol-in-grammar.py

```
# DESC
# On the first row, we write a 0. Now in every subsequent row, we look at the prev
# ious row and replace each occurrence of 0 with 01, and each occurrence of 1 with
#  10.
# Given row N and index K, return the K-th indexed symbol in row N. (The values of
#  K are 1-indexed.) (1 indexed).
# Note:

# NOTE
# K will be an integer in the range [1, 2^(N-1)].
# N will be an integer in the range [1, 30].

# EXAMPLE
# Examples:
# Input: N = 1, K = 1
# Output: 0
#
# Input: N = 2, K = 1
# Output: 0
#
# Input: N
#  = 2, K = 2
# Output: 1
#
# Input: N = 4, K = 5
# Output: 1
#
# Explanation:
# row 1: 0
# row
# 2: 01
# row 3: 0110
# row 4: 01101001

# Time:  O(logn) = O(1) because n is 32-bit integer
# Space: O(1)

class Solution(object):
    def kthGrammar(self, N, K):
        """
        :type N: int
        :type K: int
        :rtype: int
        """
        def bitCount(n):
            result = 0
            while n:
                n &= n - 1
                result += 1
            return result

        return bitCount(K-1) % 2
```

number-of-days-in-a-month.py

```python
# number-of-days-in-a-month is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def numberOfDays(self, Y, M):
        """
        :type Y: int
        :type M: int
        :rtype: int
        """
        leap = 1 if ((Y % 4 == 0) and (Y % 100 != 0)) or (Y % 400 == 0) else 0
        return (28+leap if (M == 2) else 31-(M-1)%7%2)
```

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

```
# DESC
# Return the following binary tree:
# Note:
#
# You may assume that duplicates do not exist in the tree.
# For example, given
# Given preorder and inorder traversal of a tree, construct the binary tree.

# NOTE
#

# EXAMPLE
# preorder = [3,9,20,15,7]
# inorder = [9,3,15,20,7]
# 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)}

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)
```

find-anagram-mappings.py

```python
# find-anagram-mappings is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def anagramMappings(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: List[int]
        """
        lookup = collections.defaultdict(collections.deque)
        for i, n in enumerate(B):
            lookup[n].append(i)
        result = []
        for n in A:
            result.append(lookup[n].popleft())
        return result
```

longest-palindromic-substring.py

```
# DESC
# Example 2:
# Example 1:
# Given a string s, find the longest palindromic substring in s. You may assume th
# at the maximum length of s is 1000.

# NOTE
#

# EXAMPLE
# Input: "babad"
# Output: "bab"
# Note: "aba" is also a valid answer.
# 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]]
```

subtract-the-product-and-sum-of-digits-of-an-integer.py

```python
# subtract-the-product-and-sum-of-digits-of-an-integer is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def subtractProductAndSum(self, n):
        """
        :type n: int
        :rtype: int
        """
        product, total = 1, 0
        while n:
            n, r = divmod(n, 10)
            product *= r
            total += r
        return product-total


# Time:  O(logn)
# Space: O(logn)
import operator


class Solution2(object):
    def subtractProductAndSum(self, n):
        """
        :type n: int
        :rtype: int
        """
        A = map(int, str(n))
        return reduce(operator.mul, A) - sum(A)
```

keyboard-row.py

```
# DESC
# Given a List of words, return the words that can be typed using letters of alpha
# bet on only one row's of American keyboard like the image below.
# Example:
# Note:

# NOTE
# You may assume the input string will only contain letters of alphabet.
# You may use one character in the keyboard more than once.

# EXAMPLE
# Input: ["Hello", "Alaska", "Dad", "Peace"]
# Output: ["Alaska", "Dad"]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findWords(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        rows = [set(['q', 'w', 'e', 'r', 't', 'y','u', 'i', 'o', 'p']),
                set(['a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l']),
                set(['z', 'x', 'c', 'v', 'b' ,'n', 'm'])]

        result = []
        for word in words:
            k = 0
            for i in xrange(len(rows)):
                if word[0].lower() in rows[i]:
                    k = i
                    break
            for c in word:
                if c.lower() not in rows[k]:
                    break
            else:
                result.append(word)
        return result


class Solution2(object):
    def findWords(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        keyboard_rows = ['qwertyuiop', 'asdfghjkl', 'zxcvbnm']
        single_row_words = []
        for word in words:
            for row in keyboard_rows:
                if all(letter in row for letter in word.lower()):
                    single_row_words.append(word)
        return single_row_words
```

replace-words.py

```
# DESC
# successor
# Now, given a dictionary consisting of many roots and a sentence. You need to rep
# lace all the successor in the sentence with the root forming it. If a successor
# has many roots can form it, replace it with the root with the shortest length.
# You need to output the sentence after the replacement.
# In English, we have a concept called root, which can be followed by some other w
# ords to form another longer word - let's call this word successor. For example,
# the root an, followed by other, which can form another word another.
# Constraints:
# Example 1:

# NOTE
# 1 <= sentence words length <= 1000
# 1 <= dict.length <= 1000
# The input will only have lower-case letters.
# 1 <= sentence words number <= 1000
# 1 <= dict[i].length <= 100

# EXAMPLE
# Input: dict = ["cat","bat","rat"], sentence = "the cattle was rattled by the bat
# tery"
# Output: "the cat was rat by the bat"

# 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
        """
        _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
# k-concatenation-maximum-sum is not found.
# 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
```

trim-a-binary-search-tree.py

```
# DESC
# Example 2:
# Example 1:
# Given a binary search tree and the lowest and highest boundaries as L and R, tri
# m the tree so that all its elements lies in [L, R] (R >= L). You might need to c
# hange the root of the tree, so the result should return the new root of the trim
# med binary search tree.

# NOTE
#

# EXAMPLE
# Input:
#     1
#    / \
#   0   2
#
#   L = 1
#   R = 2
#
# Output:
#     1
#       \
#        2
# Input:
#     3
#    / \
#   0   4
#    \
#     2
#    /
#   1
#
#   L = 1
#   R = 3
#
# Output:
#
#   3
#      /
#    2
#   /
#  1

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

class Solution(object):
    def trimBST(self, root, L, R):
        """
        :type root: TreeNode
        :type L: int
        :type R: int
        :rtype: TreeNode
        """
        if not root:
            return None
        if root.val < L:
```

```
    return self.trimBST(root.right, L, R)
if root.val > R:
    return self.trimBST(root.left, L, R)
root.left, root.right = self.trimBST(root.left, L, R), self.trimBST(root.right, L, R)
return root
```

replace-the-substring-for-balanced-string.py

```python
# replace-the-substring-for-balanced-string is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def balancedString(self, s):
        """
        :type s: str
        :rtype: int
        """
        count = collections.Counter(s)
        result = len(s)
        left = 0
        for right in xrange(len(s)):
            count[s[right]] -= 1
            while left < len(s) and \
                  all(v <= len(s)//4 for v in count.itervalues()):
                result = min(result, right-left+1)
                count[s[left]] += 1
                left += 1
        return result
```

implement-stack-using-queues.py

```
# DESC
# Notes:
# Implement the following operations of a stack using queues.
# Example:

# NOTE
# pop() -- Removes the element on top of the stack.
# push(x) -- Push element x onto stack.
# empty() -- Return whether the stack is empty.
# Depending on your language, queue may not be supported natively. You may simulat
# e a queue by using a list or deque (double-ended queue), as long as you use only
#  standard operations of a queue.
# top() -- Get the top element.
# You must use only standard operations of a queue -- which means only push to bac
# k, peek/pop from front, size, and is empty operations are valid.
# You may assume that all operations are valid (for example, no pop or top operati
# ons will be called on an empty stack).

# EXAMPLE
# MyStack stack = new MyStack();
#
# stack.push(1);
# stack.push(2);
# stack.top();   /
# / returns 2
# stack.pop();   // returns 2
# stack.empty(); // returns false

# Time: push: O(n), pop: O(1), top: O(1)
# Space: O(n)

import collections


class Queue(object):
    def __init__(self):
        self.data = collections.deque()

    def push(self, x):
        self.data.append(x)

    def peek(self):
        return self.data[0]

    def pop(self):
        return self.data.popleft()

    def size(self):
        return len(self.data)

    def empty(self):
        return len(self.data) == 0


class Stack(object):
    # initialize your data structure here.
    def __init__(self):
        self.q_ = Queue()
```

```
    # @param x, an integer
    # @return nothing
    def push(self, x):
        self.q_.push(x)
        for _ in xrange(self.q_.size() - 1):
            self.q_.push(self.q_.pop())

    # @return nothing
    def pop(self):
        self.q_.pop()

    # @return an integer
    def top(self):
        return self.q_.peek()

    # @return an boolean
    def empty(self):
        return self.q_.empty()


# Time: push: O(1), pop: O(n), top: O(1)
# Space: O(n)
class Stack2(object):
    # initialize your data structure here.
    def __init__(self):
        self.q_ = Queue()
        self.top_ = None

    # @param x, an integer
    # @return nothing
    def push(self, x):
        self.q_.push(x)
        self.top_ = x

    # @return nothing
    def pop(self):
        for _ in xrange(self.q_.size() - 1):
            self.top_ = self.q_.pop()
            self.q_.push(self.top_)
        self.q_.pop()

    # @return an integer
    def top(self):
        return self.top_

    # @return an boolean
    def empty(self):
        return self.q_.empty()
```

ones-and-zeroes.py

```
# DESC
# Example 1:
# Example 2:
# Given an array, strs, with strings consisting of only 0s and 1s. Also two intege
# rs m and n.
# Now your task is to find the maximum number of strings that you can form with gi
# ven m 0s and n 1s. Each 0 and 1 can be used at most once.
# Constraints:

# NOTE
# 1 <= m, n <= 100
# 1 <= strs.length <= 600
# strs[i] consists only of digits '0' and '1'.
# 1 <= strs[i].length <= 100

# EXAMPLE
# 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".
# 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".

# 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]
```

find-two-non-overlapping-sub-arrays-each-with-target-sum.py

```python
# find-two-non-overlapping-sub-arrays-each-with-target-sum is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def minSumOfLengths(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        prefix, dp = {0: -1}, [0]*len(arr)  # dp[i], min len of target subarray until i
        result = min_len = float("inf")
        accu = 0
        for right in xrange(len(arr)):
            accu += arr[right]
            prefix[accu] = right
            if accu-target in prefix:
                left = prefix[accu-target]
                min_len = min(min_len, right-left)
                if left != -1:
                    result = min(result, dp[left] + (right-left))
            dp[right] = min_len
        return result if result != float("inf") else -1
```

backspace-string-compare.py

```
# DESC
# Note:
# Follow up:
# Example 2:
# Example 4:
# Example 3:
# Note that after backspacing an empty text, the text will continue empty.
# Given two strings S and T, return if they are equal when both are typed into emp
# ty text editors. # means a backspace character.
# Example 1:

# NOTE
# Can you solve it in O(N) time and O(1) space?
# S and T only contain lowercase letters and '#' characters.
# 1 <= T.length <= 200
# 1 <= S.length <= 200

# EXAMPLE
# Input: S = "ab#c", T = "ad#c"
# Output: true
# Explanation: Both S and T become "ac".
# Input: S = "ab##", T = "c#d#"
# Output: true
# Explanation: Both S and T become "".
# Input: S = "a#c", T = "b"
# Output: false
# Explanation: S becomes "c" while T becom
# es "b".
# Input: S = "a##c", T = "#a#c"
# Output: true
# Explanation: Both S and T become "c".

# Time:  O(m + n)
# Space: O(1)

import itertools


class Solution(object):
    def backspaceCompare(self, S, T):
        """
        :type S: str
        :type T: str
        :rtype: bool
        """
        def findNextChar(S):
            skip = 0
            for i in reversed(xrange(len(S))):
                if S[i] == '#':
                    skip += 1
                elif skip:
                    skip -= 1
                else:
                    yield S[i]

        return all(x == y for x, y in
                   itertools.izip_longest(findNextChar(S), findNextChar(T)))
```

kids-with-the-greatest-number-of-candies.py

```python
# kids-with-the-greatest-number-of-candies is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def kidsWithCandies(self, candies, extraCandies):
        """
        :type candies: List[int]
        :type extraCandies: int
        :rtype: List[bool]
        """
        max_num = max(candies)
        return [x + extraCandies >= max_num for x in candies]
```

stone-game-iv.py

```
# stone-game-iv is not found.
# Time:  O(n * sqrt(n))
# Space: O(n)

class Solution(object):
    def winnerSquareGame(self, n):
        """
        :type n: int
        :rtype: bool
        """
        dp = [False]*(n+1)
        for i in xrange(1, n+1):
            j = 1
            while j*j <= i:
                if not dp[i-j*j]:
                    dp[i] = True
                    break
                j += 1
        return dp[-1]
```

find-smallest-common-element-in-all-rows.py

```python
# find-smallest-common-element-in-all-rows is not found.
# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    def smallestCommonElement(self, mat):
        """
        :type mat: List[List[int]]
        :rtype: int
        """
        # values could be duplicated in each row
        intersections = set(mat[0])
        for i in xrange(1, len(mat)):
            intersections &= set(mat[i])
            if not intersections:
                return -1
        return min(intersections)


# Time:  O(m * n)
# Space: O(n)
import collections


class Solution2(object):
    def smallestCommonElement(self, mat):
        """
        :type mat: List[List[int]]
        :rtype: int
        """
        # assumed value is unique in each row
        counter = collections.Counter()
        for row in mat:
            for c in row:
                counter[c] += 1
                if counter[c] == len(mat):
                    return c
        return -1
```

diagonal-traverse-ii.py

```
# diagonal-traverse-ii is not found.
# Time:  O(m * n)
# Space: O(m)

import itertools
import collections


class Solution(object):
    def findDiagonalOrder(self, nums):
        """
        :type nums: List[List[int]]
        :rtype: List[int]
        """
        result, dq, col = [], collections.deque(), 0
        for i in xrange(len(nums)+max(itertools.imap(len, nums))-1):
            new_dq = collections.deque()
            if i < len(nums):
                dq.appendleft((i, 0))
            for r, c in dq:
                result.append(nums[r][c])
                if c+1 < len(nums[r]):
                    new_dq.append((r, c+1))
            dq = new_dq
        return result


# Time:  O(m * n)
# Space: O(m * n)
class Solution2(object):
    def findDiagonalOrder(self, nums):
        """
        :type nums: List[List[int]]
        :rtype: List[int]
        """
        result = []
        for r, row in enumerate(nums):
            for c, num in enumerate(row):
                if len(result) <= r+c:
                    result.append([])
                result[r+c].append(num)
        return [num for row in result for num in reversed(row)]
```

validate-binary-search-tree.py

```
# DESC
# Example 2:
# Assume a BST is defined as follows:
# Given a binary tree, determine if it is a valid binary search tree (BST).
# Example 1:

# NOTE
# The right subtree of a node contains only nodes with keys greater than the node'
# s key.
# The left subtree of a node contains only nodes with keys less than the node's key.
# Both the left and right subtrees must also be binary search trees.

# EXAMPLE
# 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.
# 2
#    / \
#   1   3
#
# Input: [2,1,3]
# Output: true

# 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:
                    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)
```

convert-to-base-2.py

```python
# convert-to-base-2 is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def baseNeg2(self, N):
        """
        :type N: int
        :rtype: str
        """
        result = []
        while N:
            result.append(str(-N & 1))  # N % -2
            N = -(N >> 1)  # N //= -2
        result.reverse()
        return "".join(result) if result else "0"


# Time:  O(logn)
# Space: O(1)
class Solution2(object):
    def baseNeg2(self, N):
        """
        :type N: int
        :rtype: str
        """
        BASE = -2
        result = []
        while N:
            N, r = divmod(N, BASE)
            if r < 0:
                r -= BASE
                N += 1
            result.append(str(r))
        result.reverse()
        return "".join(result) if result else "0"
```

snakes-and-ladders.py

```
# DESC
# Return the least number of moves required to reach square N*N.  If it is not pos
# sible, return -1.
# Note that you only take a snake or ladder at most once per move: if the destinat
# ion to a snake or ladder is the start of another snake or ladder, you do not con
# tinue moving.  (For example, if the board is `[[4,-1],[-1,3]]`, and on the first
#  move your destination square is `2`, then you finish your first move at `3`, be
# cause you do not continue moving to `4`.)
# Example 1:
# Note:
# You start on square 1 of the board (which is always in the last row and first co
# lumn).  Each move, starting from square x, consists of the following:
# A board square on row r and column c has a "snake or ladder" if board[r][c] != -
# 1.  The destination of that snake or ladder is board[r][c].
# On an N x N board, the numbers from 1 to N*N are written boustrophedonically sta
# rting from the bottom left of the board, and alternating direction each row.  Fo
# r example, for a 6 x 6 board, the numbers are written as follows:

# NOTE
# board[i][j] is between 1 and N*N or is equal to -1.
# (This choice simulates the result of a standard 6-sided die roll: ie., there are
#  always at most 6 destinations, regardless of the size of the board.)
# You choose a destination square S with number x+1, x+2, x+3, x+4, x+5, or x+6, p
# rovided this number is <= N*N.
#
#
#                 (This choice simulates the result of a stand
# ard 6-sided die roll: ie., there are always at most 6 destinations, regardless o
# f the size of the board.)
# The board square with number N*N has no snake or ladder.
# 2 <= board.length = board[0].length <= 20
# If S has a snake or ladder, you move to the destination of that snake or ladder.
#   Otherwise, you move to S.
# The board square with number 1 has no snake or ladder.

# EXAMPLE
# Input: [
# [-1,-1,-1,-1,-1,-1],
# [-1,-1,-1,-1,-1,-1],
# [-1,-1,-1,-1,-1,-1],
# [-1,35,-
# 1,-1,13,-1],
# [-1,-1,-1,-1,-1,-1],
# [-1,15,-1,-1,-1,-1]]
# Output: 4
# Explanation:
# A
# t the beginning, you start at square 1 [at row 5, column 0].
# You decide to move
# to square 2, and must take the ladder to square 15.
# You then decide to move to s
# quare 17 (row 3, column 5), and must take the snake to square 13.
# You then decid
# e to move to square 14, and must take the ladder to square 35.
# You then decide t
# o move to square 36, ending the game.
# It can be shown that you need at least 4 m
# oves to reach the N*N-th square, so the answer is 4.
```

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

import collections


class Solution(object):
    def snakesAndLadders(self, board):
        """
        :type board: List[List[int]]
        :rtype: int
        """
        def coordinate(n, s):
            a, b = divmod(s-1, n)
            r = n-1-a
            c = b if r%2 != n%2 else n-1-b
            return r, c

        n = len(board)
        lookup = {1: 0}
        q = collections.deque([1])
        while q:
            s = q.popleft()
            if s == n*n:
                return lookup[s]
            for s2 in xrange(s+1, min(s+6, n*n)+1):
                r, c = coordinate(n, s2)
                if board[r][c] != -1:
                    s2 = board[r][c]
                if s2 not in lookup:
                    lookup[s2] = lookup[s]+1
                    q.append(s2)
        return -1
```

minimum-value-to-get-positive-step-by-step-sum.py

```
# minimum-value-to-get-positive-step-by-step-sum is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minStartValue(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        min_prefix, prefix = 0, 0
        for num in nums:
            prefix += num
            min_prefix = min(min_prefix, prefix)
        return 1-min_prefix
```

decoded-string-at-index.py

```
# DESC
# An encoded string S is given.  To find and write the decoded string to a tape, t
# he encoded string is read one character at a time and the following steps are ta
# ken:
# Constraints:
# Example 3:
# 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:
# Example 2:

# NOTE
# The decoded string is guaranteed to have less than 2^63 letters.
# S starts with a letter.
# 2 <= S.length <= 100
# If the character read is a letter, that letter is written onto the tape.
# S will only contain lowercase letters and digits 2 through 9.
# It's guaranteed that K is less than or equal to the length of the decoded string.
# If the character read is a digit (say d), the entire current tape is repeatedly
# written d-1 more times in total.
# 1 <= K <= 10^9

# EXAMPLE
# Input: S = "a234567899999999999999", K = 1
# Output: "a"
# Explanation:
# The decode
# d string is "a" repeated 8301530446056247680 times.  The 1st letter is "a".
# Input: S = "ha22", K = 5
# Output: "h"
# Explanation:
# The decoded string is "hahaha
# ha".  The 5th letter is "h".
# Input: S = "leet2code3", K = 10
# Output: "o"
# Explanation:
# The decoded string is
# "leetleetcodeleetleetcodeleetleetcode".
# The 10th letter in the string is "o".

# Time:  O(n)
# 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

```
# DESC
# Example 2:
# Example 1:
# Given a linked list, rotate the list to the right by k places, where k is non-ne
# gative.

# NOTE
#

# EXAMPLE
# Input: 1->2->3->4->5->NULL, k = 2
# Output: 4->5->1->2->3->NULL
# Explanation:
# rotat
# e 1 steps to the right: 5->1->2->3->4->NULL
# rotate 2 steps to the right: 4->5->1
# ->2->3->NULL
# 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 st
# eps 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
```

most-common-word.py

```
# DESC
# Words in the list of banned words are given in lowercase, and free of punctuatio
# n.  Words in the paragraph are not case sensitive.  The answer is in lowercase.
# Note:
# Example:
# Given a paragraph and a list of banned words, return the most frequent word that
#  is not in the list of banned words.  It is guaranteed there is at least one wor
# d that isn't banned, and that the answer is unique.

# NOTE
# 0 <= banned.length <= 100.
# Words only consist of letters, never apostrophes or other punctuation symbols.
# 1 <= paragraph.length <= 1000.
# paragraph only consists of letters, spaces, or the punctuation symbols !?',;.
# The answer is unique, and written in lowercase (even if its occurrences in parag
# raph may have uppercase symbols, and even if it is a proper noun.)
# 1 <= banned[i].length <= 10.
# There are no hyphens or hyphenated words.

# EXAMPLE
# Input:
# paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."
# ba
# nned = ["hit"]
# Output: "ball"
# Explanation:
# "hit" occurs 3 times, but it is a ba
# nned word.
# "ball" occurs twice (and no other word does), so it is the most frequ
# ent non-banned word in the paragraph.
# Note that words in the paragraph are not
# case sensitive,
# that punctuation is ignored (even if adjacent to words, such as
# "ball,"),
# and that "hit" isn't the answer even though it occurs more because it
#  is banned.

# Time:  O(m + n), m is the size of banned, n is the size of paragraph
# Space: O(m + n)

import collections


class Solution(object):
    def mostCommonWord(self, paragraph, banned):
        """
        :type paragraph: str
        :type banned: List[str]
        :rtype: str
        """
        lookup = set(banned)
        counts = collections.Counter(word.strip("!?',.")
                                     for word in paragraph.lower().split())

        result = ''
        for word in counts:
            if (not result or counts[word] > counts[result]) and \
               word not in lookup:
```

```
        result = word
    return result
```

minimum-path-sum.py

```
# DESC
# Note: You can only move either down or right at any point in time.
# Example:
# Given a m x n grid filled with non-negative numbers, find a path from top left t
# o bottom right which minimizes the sum of all numbers along its path.

# NOTE
#

# EXAMPLE
# Input:
# [
#   [1,3,1],
#   [1,5,1],
#   [4,2,1]
# ]
# Output: 7
# Explanation: Because the pa
# th 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]
```

minimum-insertion-steps-to-make-a-string-palindrome.py

```
# minimum-insertion-steps-to-make-a-string-palindrome is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def minInsertions(self, s):
        """
        :type s: str
        :rtype: int
        """
        def longestCommonSubsequence(text1, text2):
            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)]

        return len(s)-longestCommonSubsequence(s, s[::-1])
```

decrypt-string-from-alphabet-to-integer-mapping.py

```
# decrypt-string-from-alphabet-to-integer-mapping is not found.
# Time:  O(n)
# Space: O(1)

# forward solution
class Solution(object):
    def freqAlphabets(self, s):
        """
        :type s: str
        :rtype: str
        """
        def alpha(num):
            return chr(ord('a') + int(num)-1)

        i = 0
        result = []
        while i < len(s):
            if i+2 < len(s) and s[i+2] == '#':
                result.append(alpha(s[i:i+2]))
                i += 3
            else:
                result.append(alpha(s[i]))
                i += 1
        return "".join(result)


# Time:  O(n)
# Space: O(1)
# backward solution
class Solution2(object):
    def freqAlphabets(self, s):
        """
        :type s: str
        :rtype: str
        """
        def alpha(num):
            return chr(ord('a') + int(num)-1)

        i = len(s)-1
        result = []
        while i >= 0:
            if s[i] == '#':
                result.append(alpha(s[i-2:i]))
                i -= 3
            else:
                result.append(alpha(s[i]))
                i -= 1
        return "".join(reversed(result))

# Time:  O(n)
# Space: O(1)
import re


# regex solution
class Solution3(object):
    def freqAlphabets(self, s):
        """
```

```
    :type s: str
    :rtype: str
    """
    def alpha(num):
        return chr(ord('a') + int(num)-1)

    return "".join(alpha(i[:2]) for i in re.findall(r"\d\d#|\d", s))
```

number-of-subarrays-with-bounded-maximum.py

```
# DESC
# We are given an array A of positive integers, and two positive integers L and R
# (L <= R).
# Return the number of (contiguous, non-empty) subarrays such that the value of th
# e maximum array element in that subarray is at least L and at most R.
# Note:

# NOTE
# The length of A will be in the range of [1, 50000].
# L, R  and A[i] will be an integer in the range [0, 10^9].

# EXAMPLE
# Example :
# Input:
# A = [2, 1, 4, 3]
# L = 2
# R = 3
# Output: 3
# Explanation: There are
# three subarrays that meet the requirements: [2], [2, 1], [3].

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numSubarrayBoundedMax(self, A, L, R):
        """
        :type A: List[int]
        :type L: int
        :type R: int
        :rtype: int
        """
        def count(A, bound):
            result, curr = 0, 0
            for i in A :
                curr = curr + 1 if i <= bound else 0
                result += curr
            return result

        return count(A, R) - count(A, L-1)
```

spiral-matrix-ii.py

```
# DESC
# Given a positive integer n, generate a square matrix filled with elements from 1
#  to n2 in spiral order.
# Example:

# NOTE
#

# 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
```

same-tree.py

```
# DESC
# Example 2:
# Given two binary trees, write a function to check if they are the same or not.
# Example 3:
# Example 1:
# Two binary trees are considered the same if they are structurally identical and
# the nodes have the same value.

# NOTE
#

# EXAMPLE
# Input:     1         1
#           / \       / \
#          2   1     1   2
#
#
#  [1,2,1],   [1,1,2]
#
# Output: false
# Input:     1         1
#           / \       / \
#          2   3     2   3
#
#
#  [1,2,3],   [1,2,3]
#
# Output: true
# Input:     1         1
#           /           \
#          2             2
#
#
#  [1,2],     [1,null,2]
#
# Output: false

# 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 p, a tree node
    # @param q, a tree node
    # @return a boolean
    def isSameTree(self, p, q):
        if p is None and q is None:
            return True

        if p is not None and q is not None:
            return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

        return False
```

parse-lisp-expression.py

```
# DESC
# The syntax for these expressions is given as follows.
# You are given a string expression representing a Lisp-like expression to return
# the integer value of.
# Note:
# Evaluation Examples:

# NOTE
# The answer and all intermediate calculations of that answer are guaranteed to fi
# t in a 32-bit integer.
# The length of expression is at most 2000.  (It is also non-empty, as that would
# not be a legal expression.)
# Finally, there is the concept of scope.  When an expression of a variable name i
# s evaluated, within the context of that evaluation, the innermost scope (in term
# s of parentheses) is checked first for the value of that variable, and then oute
# r scopes are checked sequentially.  It is guaranteed that every expression is le
# gal.  Please see the examples for more details on scope.
# The given string expression is well formatted: There are no leading or trailing
# spaces, there is only a single space separating different components of the stri
# ng, and no space between adjacent parentheses.  The expression is guaranteed to
# be legal and evaluate to an integer.
# A let-expression takes the form (let v1 e1 v2 e2 ... vn en expr), where let is a
# lways the string "let", then there are 1 or more pairs of alternating variables
# and expressions, meaning that the first variable v1 is assigned the value of the
#  expression e1, the second variable v2 is assigned the value of the expression e
# 2, and so on sequentially; and then the value of this let-expression is the valu
# e of the expression expr.
# For the purposes of this question, we will use a smaller subset of variable name
# s.  A variable starts with a lowercase letter, then zero or more lowercase lette
# rs or digits.  Additionally for your convenience, the names "add", "let", or "mu
# lt" are protected and will never be used as variable names.
# An expression is either an integer, a let-expression, an add-expression, a mult-
# expression, or an assigned variable.  Expressions always evaluate to a single in
# teger.
# (An integer could be positive or negative.)
# A mult-expression takes the form (mult e1 e2) where mult is always the string "m
# ult", there are always two expressions e1, e2, and this expression evaluates to
# the multiplication of the evaluation of e1 and the evaluation of e2.
# An add-expression takes the form (add e1 e2) where add is always the string "add
# ", there are always two expressions e1, e2, and this expression evaluates to the
#  addition of the evaluation of e1 and the evaluation of e2.

# EXAMPLE
# Input: (add 1 2)
# Output: 3
#
# Input: (mult 3 (add 2 3))
# Output: 15
#
# Input: (let x
# 2 (mult x 5))
# Output: 10
#
# Input: (let x 2 (mult x (let x 3 y 4 (add x y))))
# Outp
# ut: 14
# Explanation: In the expression (add x y), when checking for the value of
# the variable x,
```

```
# we check from the innermost scope to the outermost in the contex
# t of the variable we are trying to evaluate.
# Since x = 3 is found first, the val
# ue of x is 3.
#
# Input: (let x 3 x 2 x)
# Output: 2
# Explanation: Assignment in let s
# tatements is processed sequentially.
#
# Input: (let x 1 y 2 x (add x y) (add x y))
#
# Output: 5
# Explanation: The first (add x y) evaluates as 3, and is assigned to x
# .
# The second (add x y) evaluates as 3+2 = 5.
#
# Input: (let x 2 (add (let x 3 (let
#  x 4 x)) x))
# Output: 6
# Explanation: Even though (let x 4 x) has a deeper scope,
# it is outside the context
# of the final x in the add-expression.  That final x wi
# ll equal 2.
#
# Input: (let a1 3 b2 (add a1 1) b2)
# Output 4
# Explanation: Variable
# names can contain digits after the first character.

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

class Solution(object):
    def evaluate(self, expression):
        """
        :type expression: str
        :rtype: int
        """
        def getval(lookup, x):
            return lookup.get(x, x)

        def evaluate(tokens, lookup):
            if tokens[0] in ('add', 'mult'):
                a, b = map(int, map(lambda x: getval(lookup, x), tokens[1:]))
                return str(a+b if tokens[0] == 'add' else a*b)
            for i in xrange(1, len(tokens)-1, 2):
                if tokens[i+1]:
                    lookup[tokens[i]] = getval(lookup, tokens[i+1])
            return getval(lookup, tokens[-1])

        tokens, lookup, stk = [''], {}, []
        for c in expression:
            if c == '(':
                if tokens[0] == 'let':
                    evaluate(tokens, lookup)
                stk.append((tokens, dict(lookup)))
                tokens =  ['']
            elif c == ' ':
                tokens.append('')
```

```
        elif c == ')':
            val = evaluate(tokens, lookup)
            tokens, lookup = stk.pop()
            tokens[-1] += val
        else:
            tokens[-1] += c
    return int(tokens[0])
```

palindromic-substrings.py

```python
# DESC
# 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 differ
# ent substrings even they consist of same characters.
# Example 2:
# Note:
# Example 1:

# NOTE
# The input string length won't exceed 1000.

# EXAMPLE
# Input: "aaa"
# Output: 6
# Explanation: Six palindromic strings: "a", "a", "a", "aa"
# , "aa", "aaa".
# Input: "abc"
# Output: 3
# Explanation: Three palindromic strings: "a", "b", "c".

# 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

```
# DESC
# Return all non-negative integers of length N such that the absolute difference b
# etween every two consecutive digits is K.
# Note:
# Example 1:
# Example 2:
# Note that every number in the answer must not have leading zeros except for the
# number 0 itself. For example, 01 has one leading zero and is invalid, but 0 is v
# alid.
# You may return the answer in any order.

# NOTE
# 1 <= N <= 9
# 0 <= K <= 9

# EXAMPLE
# Input: N = 2, K = 1
# Output: [10,12,21,23,32,34,43,45,54,56,65,67,76,78,87,89,98]
# Input: N = 3, K = 7
# Output: [181,292,707,818,929]
# Explanation: Note that 070 is
# not a valid number, because it has leading zeroes.

# 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
```

employee-free-time.py

```python
# employee-free-time is not found.
# Time:  O(m * logn), m is the number of schedule, n is the number of employees, m >= n
# Space: O(n)

import heapq


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


class Solution(object):
    def employeeFreeTime(self, schedule):
        """
        :type schedule: List[List[Interval]]
        :rtype: List[Interval]
        """
        result = []
        min_heap = [(emp[0].start, eid, 0) for eid, emp in enumerate(schedule)]
        heapq.heapify(min_heap)
        last_end = -1
        while min_heap:
            t, eid, i = heapq.heappop(min_heap)
            if 0 <= last_end < t:
                result.append(Interval(last_end, t))
            last_end = max(last_end, schedule[eid][i].end)
            if i+1 < len(schedule[eid]):
                heapq.heappush(min_heap, (schedule[eid][i+1].start, eid, i+1))
        return result
```

special-binary-string.py

```
# DESC
# Note:
# Given a special string S, a move consists of choosing two consecutive, non-empty
# , special substrings of S, and swapping them.  (Two strings are consecutive if t
# he last character of the first string is exactly one index before the first char
# acter of the second string.)
# Special binary strings are binary strings with the following two properties:
# At the end of any number of moves, what is the lexicographically largest resulti
# ng string possible?
# Example 1:

# NOTE
# S is guaranteed to be a special binary string as defined above.
# The number of 0's is equal to the number of 1's.
# S has length at most 50.
# Every prefix of the binary string has at least as many 1's as 0's.

# EXAMPLE
# Input: S = "11011000"
# Output: "11100100"
# Explanation:
# The strings "10" [occuring
#  at S[1]] and "1100" [at S[3]] are swapped.
# This is the lexicographically larges
# t string possible after some number of swaps.

# Time:  f(n) = k * f(n/k) + n/k * klogk <= O(logn * nlogk) <= O(n^2)
#        n is the length of S, k is the max number of special strings in each depth
# Space: O(n)

class Solution(object):
    def makeLargestSpecial(self, S):
        """
        :type S: str
        :rtype: str
        """
        result = []
        anchor = count = 0
        for i, v in enumerate(S):
            count += 1 if v == '1' else -1
            if count == 0:
                result.append("1{}0".format(self.makeLargestSpecial(S[anchor+1:i])))
                anchor = i+1
        result.sort(reverse = True)
        return "".join(result)
```

minimum-possible-integer-after-at-most-k-adjacent-swaps-on-digits.py

```python
# minimum-possible-integer-after-at-most-k-adjacent-swaps-on-digits is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections


class BIT(object):  # Fenwick Tree, 1-indexed
    def __init__(self, n):
        self.__bit = [0] * n

    def add(self, i, val):
        while i < len(self.__bit):
            self.__bit[i] += val
            i += (i & -i)

    def sum(self, i):
        result = 0
        while i > 0:
            result += self.__bit[i]
            i -= (i & -i)
        return result


class Solution(object):
    def minInteger(self, num, k):
        """
        :type num: str
        :type k: int
        :rtype: str
        """
        lookup = collections.defaultdict(list)
        bit = BIT(len(num)+1)
        for i in reversed(xrange(len(num))):
            bit.add(i+1, 1)
            lookup[int(num[i])].append(i+1)
        result = []
        for _ in xrange(len(num)):
            for d in xrange(10):
                if lookup[d] and bit.sum(lookup[d][-1]-1) <= k:
                    k -= bit.sum(lookup[d][-1]-1)
                    bit.add(lookup[d].pop(), -1)
                    result.append(d)
                    break
        return "".join(map(str, result))
```

generate-random-point-in-a-circle.py

```
# DESC
# Example 1:
# Explanation of Input Syntax:
# Given the radius and x-y positions of the center of a circle, write a function r
# andPoint which generates a uniform random point in the circle.
# The input is two lists: the subroutines called and their arguments. Solution's c
# onstructor has three arguments, the radius, x-position of the center, and y-posi
# tion of the center of the circle. randPoint has no arguments. Arguments are alwa
# ys wrapped with a list, even if there aren't any.
# Example 2:
# Note:

# NOTE
# randPoint returns a size 2 array containing x-position and y-position of the ran
# dom point, in that order.
# radius and x-y position of the center of the circle is passed into the class con
# structor.
# input and output values are in floating-point.
# a point on the circumference of the circle is considered to be in the circle.

# EXAMPLE
# Input:
# ["Solution","randPoint","randPoint","randPoint"]
# [[1,0,0],[],[],[]]
# Outp
# ut: [null,[-0.72939,-0.65505],[-0.78502,-0.28626],[-0.83119,-0.19803]]
# Input:
# ["Solution","randPoint","randPoint","randPoint"]
# [[10,5,-7.5],[],[],[]]
#
# Output: [null,[11.52438,-8.33273],[2.46992,-16.21705],[11.13430,-12.42337]]

# Time:  O(1)
# Space: O(1)

import random
import math

class Solution(object):

    def __init__(self, radius, x_center, y_center):
        """
        :type radius: float
        :type x_center: float
        :type y_center: float
        """
        self.__radius = radius
        self.__x_center = x_center
        self.__y_center = y_center


    def randPoint(self):
        """
        :rtype: List[float]
        """
        r = (self.__radius) * math.sqrt(random.uniform(0, 1))
        theta = (2*math.pi) * random.uniform(0, 1)
        return (r*math.cos(theta) + self.__x_center,
```

```
            r*math.sin(theta) + self.__y_center)
```

longest-well-performing-interval.py

```python
# DESC
# A day is considered to be a tiring day if and only if the number of hours worked
#  is (strictly) greater than 8.
# Return the length of the longest well-performing interval.
# Example 1:
# Constraints:
# 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.
# We are given hours, a list of the number of hours worked per day for a given emp
# loyee.

# NOTE
# 1 <= hours.length <= 10000
# 0 <= hours[i] <= 16

# EXAMPLE
# Input: hours = [9,9,6,0,6,6,9]
# Output: 3
# Explanation: The longest well-performin
# g interval is [9,9,6].

# 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

```
# DESC
# You may assume the two numbers do not contain any leading zero, except the numbe
# r 0 itself.
# Follow up:
#
# What if you cannot modify the input lists? In other words, reversing
#  the lists is not allowed.
# 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.
# Example:

# NOTE
#

# 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

```
# DESC
# If there isn't any rectangle, return 0.
# Example 3:
# Note:
# Given a set of points in the xy-plane, determine the minimum area of any rectang
# le formed from these points, with sides not necessarily parallel to the x and y
# axes.
# Example 4:
# Example 2:
# Example 1:

# NOTE
# 1 <= points.length <= 50
# 0 <= points[i][0] <= 40000
# All points are distinct.
# 0 <= points[i][1] <= 40000
# Answers within 10^-5 of the actual value will be accepted as correct.

# EXAMPLE
# Input: [[1,2],[2,1],[1,0],[0,1]]
# Output: 2.00000
# Explanation: The minimum area r
# ectangle occurs at [1,2],[2,1],[1,0],[0,1], with an area of 2.
# Input: [[0,3],[1,2],[3,1],[1,3],[2,1]]
# Output: 0
# Explanation: There is no possib
# le rectangle to form from these points.
# Input: [[3,1],[1,1],[0,1],[2,1],[3,3],[3,2],[0,2],[2,3]]
# Output: 2.00000
# Explana
# tion: The minimum area rectangle occurs at [2,1],[2,3],[3,3],[3,1], with an area
#  of 2.
# Input: [[0,1],[2,1],[1,1],[1,0],[2,0]]
# Output: 1.00000
# Explanation: The minimum
# area rectangle occurs at [1,0],[1,1],[2,1],[2,0], with an area of 1.

# 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
```

binary-tree-tilt.py

```
# DESC
# Given a binary tree, return the tilt of the whole tree.
# The tilt of a tree node is defined as the absolute difference between the sum of
#  all left subtree node values and the sum of all right subtree node values. Null
#  node has tilt 0.
# Note:
# Example:
# The tilt of the whole tree is defined as the sum of all nodes' tilt.

# NOTE
# The sum of node values in any subtree won't exceed the range of 32-bit integer.
# All the tilt values won't exceed the range of 32-bit integer.

# EXAMPLE
# Input:
#          1
#        /   \
#       2     3
# Output: 1
# Explanation:
# Tilt of no
# de 2 : 0
# Tilt of node 3 : 0
# Tilt of node 1 : |2-3| = 1
# Tilt of binary tree : 0 +
#  0 + 1 = 1

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

class Solution(object):
    def findTilt(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def postOrderTraverse(root, tilt):
            if not root:
                return 0, tilt
            left, tilt = postOrderTraverse(root.left, tilt)
            right, tilt = postOrderTraverse(root.right, tilt)
            tilt += abs(left-right)
            return left+right+root.val, tilt

        return postOrderTraverse(root, 0)[1]
```

meeting-rooms-ii.py

```
# meeting-rooms-ii is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    # @param {Interval[]} intervals
    # @return {integer}
    def minMeetingRooms(self, intervals):
        result, curr = 0, 0
        line = [x for i, j in intervals for x in [[i, 1], [j, -1]]]
        line.sort()
        for _, num in line:
            curr += num
            result = max(result, curr)
        return result


# Time:  O(nlogn)
# Space: O(n)
class Solution2(object):
    # @param {Interval[]} intervals
    # @return {integer}
    def minMeetingRooms(self, intervals):
        starts, ends = [], []
        for start, end in intervals:
            starts.append(start)
            ends.append(end)

        starts.sort()
        ends.sort()

        s, e = 0, 0
        min_rooms, cnt_rooms = 0, 0
        while s < len(starts):
            if starts[s] < ends[e]:
                cnt_rooms += 1  # Acquire a room.
                # Update the min number of rooms.
                min_rooms = max(min_rooms, cnt_rooms)
                s += 1
            else:
                cnt_rooms -= 1  # Release a room.
                e += 1

        return min_rooms


# Time: O(nlogn)
# Space: O(n)
from heapq import heappush, heappop


class Solution3(object):
    def minMeetingRooms(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: int
        """
        if not intervals:
```

```
    return 0

intervals.sort(key=lambda x: x[0])
free_rooms = []

heappush(free_rooms, intervals[0][1])
for interval in intervals[1:]:
    if free_rooms[0] <= interval[0]:
        heappop(free_rooms)

    heappush(free_rooms, interval[1])

return len(free_rooms)
```

random-pick-with-blacklist.py

```
# DESC
# Note:
# Example 1:
# Given a blacklist B containing unique integers from [0, N), write a function to
# return a uniform random integer from [0, N) which is NOT in B.
# Math.random()
# Optimize it such that it minimizes the call to system's Math.random().
# The input is two lists: the subroutines called and their arguments. Solution's c
# onstructor has two arguments, N and the blacklist B. pick has no arguments. Argu
# ments are always wrapped with a list, even if there aren't any.
# Example 3:
# Explanation of Input Syntax:
# Example 2:
# Example 4:

# NOTE
# [0, N) does NOT include N. See interval notation.
# 0 <= B.length < min(100000, N)
# 1 <= N <= 1000000000

# EXAMPLE
# Input:
# ["Solution","pick","pick","pick"]
# [[3,[1]],[],[],[]]
# Output: [null,0,0,2]
# Input:
# ["Solution","pick","pick","pick"]
# [[4,[2]],[],[],[]]
# Output: [null,1,3,1]
# Input:
# ["Solution","pick","pick","pick"]
# [[1,[]],[],[],[]]
# Output: [null,0,0,0]
# Input:
# ["Solution","pick","pick","pick"]
# [[2,[]],[],[],[]]
# Output: [null,1,1,1]

# Time:  ctor: O(b)
#        pick: O(1)
# Space: O(b)

import random


class Solution(object):

    def __init__(self, N, blacklist):
        """
        :type N: int
        :type blacklist: List[int]
        """
        self.__n = N-len(blacklist)
        self.__lookup = {}
        white = iter(set(range(self.__n, N))-set(blacklist))
        for black in blacklist:
            if black < self.__n:
                self.__lookup[black] = next(white)
```

```
    def pick(self):
        """
        :rtype: int
        """
        index = random.randint(0, self.__n-1)
        return self.__lookup[index] if index in self.__lookup else index


# Time:  ctor: O(blogb)
#        pick: O(logb)
# Space: O(b)
import random

class Solution2(object):

    def __init__(self, N, blacklist):
        """
        :type N: int
        :type blacklist: List[int]
        """
        self.__n = N-len(blacklist)
        blacklist.sort()
        self.__blacklist = blacklist

    def pick(self):
        """
        :rtype: int
        """
        index = random.randint(0, self.__n-1)
        left, right = 0, len(self.__blacklist)-1
        while left <= right:
            mid = left+(right-left)//2
            if index+mid < self.__blacklist[mid]:
                right = mid-1
            else:
                left = mid+1
        return index+left
```

clumsy-factorial.py

```
# DESC
# Additionally, the division that we use is floor division such that 10 * 9 / 8 eq
# uals 11.  This guarantees the result is an integer.
# Example 1:
# We instead make a clumsy factorial: using the integers in decreasing order, we s
# wap out the multiply operations for a fixed rotation of operations: multiply (*)
# , divide (/), add (+) and subtract (-) in this order.
# Example 2:
# Normally, the factorial of a positive integer n is the product of all positive i
# ntegers less than or equal to n.  For example, factorial(10) = 10 * 9 * 8 * 7 *
# 6 * 5 * 4 * 3 * 2 * 1.
# For example, clumsy(10) = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1.  However, thes
# e 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.
# Note:
# Implement the clumsy function as defined above: given an integer N, it returns t
# he clumsy factorial of N.

# NOTE
# 1 <= N <= 10000
# -2^31 <= answer <= 2^31 - 1  (The answer is guaranteed to fit within a 32-bit in
# teger.)

# EXAMPLE
# Input: 4
# Output: 7
# Explanation: 7 = 4 * 3 / 2 + 1
# Input: 10
# Output: 12
# Explanation: 12 = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1

# 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
#     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
```

occurrences-after-bigram.py

```
# DESC
# Example 1:
# third
# Note:
# Given words first and second, consider occurrences in some text of the form "fir
# st second third", where second comes immediately after first, and third comes im
# mediately after second.
# For each such occurrence, add "third" to the answer, and return the answer.
# Example 2:

# NOTE
# 1 <= first.length, second.length <= 10
# 1 <= text.length <= 1000
# text consists of space separated words, where each word consists of lowercase En
# glish letters.
# first and second consist of lowercase English letters.

# EXAMPLE
# Input: text = "we will we will rock you", first = "we", second = "will"
# Output:
# ["we","rock"]
# Input: text = "alice is a good girl she is a good student", first = "a", second
# = "good"
# Output: ["girl","student"]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findOcurrences(self, text, first, second):
        """
        :type text: str
        :type first: str
        :type second: str
        :rtype: List[str]
        """
        result = []
        first += ' '
        second += ' '
        third = []
        i, j, k = 0, 0, 0
        while k < len(text):
            c = text[k]
            k += 1
            if i != len(first):
                if c == first[i]:
                    i += 1
                else:
                    i = 0
                continue
            if j != len(second):
                if c == second[j]:
                    j += 1
                else:
                    k -= j+1
                    i, j = 0, 0
                continue
            if c != ' ':
```

```
            third.append(c)
            continue
        k -= len(second) + len(third) + 1
        i, j = 0, 0
        result.append("".join(third))
        third = []
    if third:
        result.append("".join(third))
    return result
```

max-sum-of-sub-matrix-no-larger-than-k.py

```
# max-sum-of-sub-matrix-no-larger-than-k is not found.
# Time:  O(min(m, n)^2 * max(m, n) * log(max(m, n)))
# Space: O(max(m, n))

from bisect import bisect_left, insort

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

        m = min(len(matrix), len(matrix[0]))
        n = max(len(matrix), len(matrix[0]))
        result = float("-inf")

        for i in xrange(m):
            sums = [0] * n
            for j in xrange(i, m):
                for l in xrange(n):
                    sums[l] += matrix[j][l] if m == len(matrix) else matrix[l][j]

                # Find the max subarray no more than K.
                accu_sum_set, accu_sum = [0], 0
                for sum in sums:
                    accu_sum += sum
                    it = bisect_left(accu_sum_set, accu_sum - k)  # Time: O(logn)
                    if it != len(accu_sum_set):
                        result = max(result, accu_sum - accu_sum_set[it])
                    insort(accu_sum_set, accu_sum)  # Time: O(n)

        return result


# Time:  O(min(m, n)^2 * max(m, n) * log(max(m, n))) ~ O(min(m, n)^2 * max(m, n)^2)
# Space: O(max(m, n))
class Solution_TLE(object):
    def maxSumSubmatrix(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        class BST(object):  # not avl, rbtree
            def __init__(self, val):
                self.val = val
                self.left = None
                self.right = None

            def insert(self, val):  # Time: O(h) = O(logn) ~ O(n)
                curr = self
                while curr:
                    if curr.val >= val:
                        if curr.left:
```

```
                    curr = curr.left
                else:
                    curr.left = BST(val)
                    return
            else:
                if curr.right:
                    curr = curr.right
                else:
                    curr.right = BST(val)
                    return

    def lower_bound(self, val):  # Time: O(h) = O(logn) ~ O(n)
        result, curr = None, self
        while curr:
            if curr.val >= val:
                result, curr = curr, curr.left
            else:
                curr = curr.right
        return result


if not matrix:
    return 0

m = min(len(matrix), len(matrix[0]))
n = max(len(matrix), len(matrix[0]))
result = float("-inf")

for i in xrange(m):
    sums = [0] * n
    for j in xrange(i, m):
        for l in xrange(n):
            sums[l] += matrix[j][l] if m == len(matrix) else matrix[l][j]

        # Find the max subarray no more than K.
        accu_sum_set = BST(0)
        accu_sum = 0
        for sum in sums:
            accu_sum += sum
            node = accu_sum_set.lower_bound(accu_sum - k)
            if node:
                result = max(result, accu_sum - node.val)
            accu_sum_set.insert(accu_sum)

return result
```

minimum-window-substring.py

```
# DESC
# Example:
# Note:
# Given a string S and a string T, find the minimum window in S which will contain
#  all the characters in T in complexity O(n).

# NOTE
# If there is such window, you are guaranteed that there will always be only one u
# nique minimum window in S.
# If there is no such window in S that covers all characters in T, return the empt
# y string "".

# EXAMPLE
# Input: S = "ADOBECODEBANC", T = "ABC"
# Output: "BANC"

# Time:  O(n)
# Space: O(k), k is the number of different characters

class Solution(object):
    def minWindow(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        current_count = [0 for i in xrange(52)]
        expected_count = [0 for i in xrange(52)]

        for char in t:
            expected_count[ord(char) - ord('a')] += 1

        i, count, start, min_width, min_start = 0, 0, 0, float("inf"), 0
        while i < len(s):
            current_count[ord(s[i]) - ord('a')] += 1
            if current_count[ord(s[i]) - ord('a')] <= expected_count[ord(s[i]) - ord('a')]:
                count += 1

            if count == len(t):
                while expected_count[ord(s[start]) - ord('a')] == 0 or \
                      current_count[ord(s[start]) - ord('a')] > expected_count[ord(s[start]) - ord('a')]:
                    current_count[ord(s[start]) - ord('a')] -= 1
                    start += 1

                if min_width > i - start + 1:
                    min_width = i - start + 1
                    min_start = start
            i += 1

        if min_width == float("inf"):
            return ""

        return s[min_start:min_start + min_width]
```

nested-list-weight-sum-ii.py

```python
# nested-list-weight-sum-ii is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def depthSumInverse(self, nestedList):
        """
        :type nestedList: List[NestedInteger]
        :rtype: int
        """
        def depthSumInverseHelper(list, depth, result):
            if len(result) < depth + 1:
                result.append(0)
            if list.isInteger():
                result[depth] += list.getInteger()
            else:
                for l in list.getList():
                    depthSumInverseHelper(l, depth + 1, result)

        result = []
        for list in nestedList:
            depthSumInverseHelper(list, 0, result)

        sum = 0
        for i in reversed(xrange(len(result))):
            sum += result[i] * (len(result) - i)
        return sum
```

baseball-game.py

```
# DESC
# Example 1:
# Each round's operation is permanent and could have an impact on the round before
#  and the round after.
# Example 2:
# Given a list of strings, each string can be one of the 4 following types:
# You're now a baseball game point recorder.
# You need to return the sum of the points you could get in all the rounds.
# Note:

# NOTE
# The size of the input list will be between 1 and 1000.
# "C" (an operation, which isn't a round's score): Represents the last valid round
# 's points you get were invalid and should be removed.
# "D" (one round's score): Represents that the points you get in this round are th
# e doubled data of the last valid round's points.
# Integer (one round's score): Directly represents the number of points you get in
#  this round.
# "+" (one round's score): Represents that the points you get in this round are th
# e sum of the last two valid round's points.
# Every integer represented in the list will be between -30000 and 30000.

# EXAMPLE
# Input: ["5","2","C","D","+"]
# Output: 30
# Explanation:
# Round 1: You could get 5 p
# oints. The sum is: 5.
# Round 2: You could get 2 points. The sum is: 7.
# Operation
# 1: The round 2's data was invalid. The sum is: 5.
# Round 3: You could get 10 po
# ints (the round 2's data has been removed). The sum is: 15.
# Round 4: You could g
# et 5 + 10 = 15 points. The sum is: 30.
# Input: ["5","-2","4","C","D","9","+","+"]
# Output: 27
# Explanation:
# Round 1: You
# could get 5 points. The sum is: 5.
# Round 2: You could get -2 points. The sum is:
#  3.
# Round 3: You could get 4 points. The sum is: 7.
# Operation 1: The round 3's d
# ata is invalid. The sum is: 3.
# Round 4: You could get -4 points (the round 3's
#  data has been removed). The sum is: -1.
# Round 5: You could get 9 points. The su
# m is: 8.
# Round 6: You could get -4 + 9 = 5 points. The sum is 13.
# Round 7: You c
# ould get 9 + 5 = 14 points. The sum is 27.

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

class Solution(object):
    def calPoints(self, ops):
```

```
        """
        :type ops: List[str]
        :rtype: int
        """
        history = []
        for op in ops:
            if op == '+':
                history.append(history[-1] + history[-2])
            elif op == 'D':
                history.append(history[-1] * 2)
            elif op == 'C':
                history.pop()
            else:
                history.append(int(op))
        return sum(history)
```

design-hashset.py

```
# DESC
# Design a HashSet without using any built-in hash table libraries.
# To be specific, your design should include these functions:
# Note:
# Example:

# NOTE
# add(value): Insert a value into the HashSet.
# Please do not use the built-in HashSet library.
# contains(value) : Return whether the value exists in the HashSet or not.
# remove(value): Remove a value in the HashSet. If the value does not exist in the
#  HashSet, do nothing.
# All values will be in the range of [0, 1000000].
# The number of operations will be in the range of [1, 10000].

# EXAMPLE
# MyHashSet hashSet = new MyHashSet();
# hashSet.add(1);
# hashSet.add(2);
#
# hashSet.contains(1);    // returns true
# hashSet.contains(3);    // return
# s false (not found)
# hashSet.add(2);
# hashSet.contains(2);    // returns
#  true
# hashSet.remove(2);
# hashSet.contains(2);    // returns false (alr
# eady removed)

# Time:  O(1)
# Space: O(n)

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:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        node.next, node.prev = None, None  # make node clean

    def find(self, key):
        curr = self.head
        while curr:
            if curr.key == key:
                break
            curr = curr.next
        return curr


class MyHashSet(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__data = [LinkedList() for _ in xrange(10000)]

    def add(self, key):
        """
        :type key: int
        :rtype: void
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        if not node:
            l.insert(ListNode(key, 0))

    def remove(self, key):
        """
        :type key: int
        :rtype: void
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        if node:
            l.delete(node)

    def contains(self, key):
        """
        Returns true if this set did not already contain the specified element
        :type key: int
        :rtype: bool
        """
        l = self.__data[key % len(self.__data)]
        node = l.find(key)
        return node is not None
```

count-all-valid-pickup-and-delivery-options.py

```python
# count-all-valid-pickup-and-delivery-options is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countOrders(self, n):
        """
        :type n: int
        :rtype: int
        """
        MOD = 10**9+7
        result = 1
        for i in reversed(xrange(2, 2*n+1, 2)):
            result = result * i*(i-1)//2 % MOD
        return result
```

next-greater-element-iii.py

```
# DESC
# Example 1:
# Example 2:
# 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 va
# lue than n. If no such positive 32-bit integer exists, you need to return -1.

# NOTE
#

# EXAMPLE
# Input: 12
# Output: 21
# 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

```
# DESC
# Example:
# Given a collection of distinct integers, return all possible permutations.

# NOTE
#

# 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]
            #                 -> 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)
```

rearrange-words-in-a-sentence.py

```python
# rearrange-words-in-a-sentence is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def arrangeWords(self, text):
        """
        :type text: str
        :rtype: str
        """
        result = text.split()
        result[0] = result[0].lower()
        result.sort(key=len)
        result[0] = result[0].title()
        return " ".join(result)
```

number-of-longest-increasing-subsequence.py

```
# DESC
# Example 2:
# Example 1:
# Given an unsorted array of integers, find the number of longest increasing subse
# quence.
# Note:
# Length of the given array will be not exceed 2000 and the answer is guaran
# teed to be fit in 32-bit signed int.

# NOTE
#

# EXAMPLE
# Input: [2,2,2,2,2]
# Output: 5
# Explanation: The length of longest continuous incre
# asing subsequence is 1, and there are 5 subsequences' length is 1, so output 5.
# Input: [1,3,5,4,7]
# Output: 2
# Explanation: The two longest increasing subsequence
#  are [1, 3, 4, 7] and [1, 3, 5, 7].

# 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
```

build-array-where-you-can-find-the-maximum-exactly-k-comparisons.py

```python
# build-array-where-you-can-find-the-maximum-exactly-k-comparisons is not found.
# Time:  O(n * m * k)
# Space: O(m * k)

class Solution(object):
    def numOfArrays(self, n, m, k):
        """
        :type n: int
        :type m: int
        :type k: int
        :rtype: int
        """
        MOD = 10**9 + 7
        # dp[l][i][j] = number of ways of constructing array length l with max element i at search cost j
        dp = [[[0]*(k+1) for _ in xrange(m+1)] for _ in xrange(2)]
        # prefix_dp[l][i][j] = sum(dp[l][i][j] for i in [1..i])
        prefix_dp = [[[0]*(k+1) for _ in xrange(m+1)] for _ in xrange(2)]
        for i in xrange(1, m+1):
            dp[1][i][1] = 1
            prefix_dp[1][i][1] = (prefix_dp[1][i-1][1] + dp[1][i][1])%MOD
        for l in xrange(2, n+1):
            for i in xrange(1, m+1):
                for j in xrange(1, k+1):
                    dp[l%2][i][j] = (i*dp[(l-1)%2][i][j]%MOD + prefix_dp[(l-1)%2][i-1][j-1])%MOD
                    prefix_dp[l%2][i][j] = (prefix_dp[l%2][i-1][j] + dp[l%2][i][j])%MOD
        return prefix_dp[n%2][m][k]
```

redundant-connection.py

```
# DESC
# Update (2017-09-26):
#
# We have overhauled the problem description + test cases an
# d specified clearly the graph is an undirected graph. For the directed graph fol
# low up please see Redundant Connection II). We apologize for any inconvenience c
# aused.
# Note:
# Example 2:
# The resulting graph is given as a 2D-array of edges.  Each element of edges is a
#  pair [u, v] with u < v, that represents an undirected edge connecting nodes u a
# nd v.
# The given input is a graph that started as a tree with N nodes (with distinct va
# lues 1, 2, ..., N), with one additional edge added.  The added edge has two diff
# erent vertices chosen from 1 to N, and was not an edge that already existed.
# Example 1:
# Return an edge that can be removed so that the resulting graph is a tree of N no
# des.  If there are multiple answers, return the answer that occurs last in the g
# iven 2D-array.  The answer edge [u, v] should be in the same format, with u < v.
# In this problem, a tree is an undirected graph that is connected and has no cycles.

# NOTE
# The size of the input 2D-array will be between 3 and 1000.
# Every integer represented in the 2D-array will be between 1 and N, where N is th
# e size of the input array.

# EXAMPLE
# Input: [[1,2], [1,3], [2,3]]
# Output: [2,3]
# Explanation: The given undirected gra
# ph will be like this:
#   1
#  / \
# 2 - 3
# Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]
# Output: [1,4]
# Explanation: The given
# undirected graph will be like this:
# 5 - 1 - 2
#     |   |
#     4 - 3

# Time:  O(nlog*n) ~= O(n), n is the length of the positions
# 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 findRedundantConnection(self, edges):
        """
        :type edges: List[List[int]]
        :rtype: List[int]
        """
        union_find = UnionFind(len(edges)+1)
        for edge in edges:
            if not union_find.union_set(*edge):
                return edge
        return []
```

divide-chocolate.py

```
# divide-chocolate is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def maximizeSweetness(self, sweetness, K):
        """
        :type sweetness: List[int]
        :type K: int
        :rtype: int
        """
        def check(sweetness, K, x):
            curr, cuts = 0, 0
            for s in sweetness:
                curr += s
                if curr >= x:
                    cuts += 1
                    curr = 0
            return cuts >= K+1

        left, right = min(sweetness), sum(sweetness)//(K+1)
        while left <= right:
            mid = left + (right-left)//2
            if not check(sweetness, K, mid):
                right = mid-1
            else:
                left = mid+1
        return right
```

top-k-frequent-words.py

```
# DESC
# Example 2:
# Note:
# Follow up:
# Your answer should be sorted by frequency from highest to lowest. If two words h
# ave the same frequency, then the word with the lower alphabetical order comes fi
# rst.
# Given a non-empty list of words, return the k most frequent elements.
# Example 1:

# NOTE
# Try to solve it in O(n log k) time and O(n) extra space.
# Input words contain only lowercase letters.
# You may assume k is always valid, 1  k  number of unique elements.

# EXAMPLE
# Input: ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"],
# k = 4
# Output: ["the", "is", "sunny", "day"]
# Explanation: "the", "is", "sunny" an
# d "day" are the four most frequent words,
#     with the number of occurrence bein
# g 4, 3, 2 and 1 respectively.
# Input: ["i", "love", "leetcode", "i", "love", "coding"], k = 2
# Output: ["i", "lo
# ve"]
# Explanation: "i" and "love" are the two most frequent words.
#     Note that
# "i" comes before "love" due to a lower alphabetical order.

# Time:  O(n + klogk) on average
# Space: O(n)

import collections
import heapq
from random import randint


class Solution(object):
    def topKFrequent(self, words, k):
        """
        :type words: List[str]
        :type k: int
        :rtype: List[str]
        """
        counts = collections.Counter(words)
        p = []
        for key, val in counts.iteritems():
            p.append((-val, key))
        self.kthElement(p, k-1)

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

    def kthElement(self, nums, k):  # O(n) on average
```

```
        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
            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)
# Heap Solution
class Solution2(object):
    def topKFrequent(self, words, k):
        """
        :type words: List[str]
        :type k: int
        :rtype: List[str]
        """
        class MinHeapObj(object):
            def __init__(self,val):
                self.val = val
            def __lt__(self,other):
                return self.val[1] > other.val[1] if self.val[0] == other.val[0] else \
                       self.val < other.val
            def __eq__(self,other):
                return self.val == other.val
            def __str__(self):
                return str(self.val)

        counts = collections.Counter(words)
        min_heap = []
        for word, count in counts.iteritems():
            heapq.heappush(min_heap, MinHeapObj((count, word)))
            if len(min_heap) == k+1:
                heapq.heappop(min_heap)
        result = []
        while min_heap:
            result.append(heapq.heappop(min_heap).val[1])
        return result[::-1]


# Time:  O(n + klogk) ~ O(n + nlogn)
# Space: O(n)
# Bucket Sort Solution
```

```
class Solution3(object):
    def topKFrequent(self, words, k):
        """
        :type words: List[str]
        :type k: int
        :rtype: List[str]
        """
        counts = collections.Counter(words)
        buckets = [[] for _ in xrange(len(words)+1)]
        for word, count in counts.iteritems():
            buckets[count].append(word)
        pairs = []
        for i in reversed(xrange(len(words))):
            for word in buckets[i]:
                pairs.append((-i, word))
            if len(pairs) >= k:
                break
        pairs.sort()
        return [pair[1] for pair in pairs[:k]]


# time: O(nlogn)
# space: O(n)

from collections import Counter


class Solution4(object):
    def topKFrequent(self, words, k):
        """
        :type words: List[str]
        :type k: int
        :rtype: List[str]
        """
        counter = Counter(words)
        candidates = counter.keys()
        candidates.sort(key=lambda w: (-counter[w], w))
        return candidates[:k]
```

matrix-cells-in-distance-order.py

```
# DESC
# (r0, c0)
# Note:
# Additionally, we are given a cell in that matrix with coordinates (r0, c0).
# Example 3:
# We are given a matrix with R rows and C columns has cells with integer coordinat
# es (r, c), where 0 <= r < R and 0 <= c < C.
# Example 2:
# Example 1:
# Return the coordinates of all cells in the matrix, sorted by their distance from
#  (r0, c0) from smallest distance to largest distance.  Here, the distance betwee
# n two cells (r1, c1) and (r2, c2) is the Manhattan distance, |r1 - r2| + |c1 - c
# 2|.  (You may return the answer in any order that satisfies this condition.)

# NOTE
# 1 <= R <= 100
# 0 <= r0 < R
# 1 <= C <= 100
# 0 <= c0 < C

# EXAMPLE
# Input: R = 2, C = 3, r0 = 1, c0 = 2
# Output: [[1,2],[0,2],[1,1],[0,1],[1,0],[0,0]
# ]
# Explanation: The distances from (r0, c0) to other cells are: [0,1,1,2,2,3]
# The
# re are other answers that would also be accepted as correct, such as [[1,2],[1,1
# ],[0,2],[1,0],[0,1],[0,0]].
# Input: R = 2, C = 2, r0 = 0, c0 = 1
# Output: [[0,1],[0,0],[1,1],[1,0]]
# Explanatio
# n: The distances from (r0, c0) to other cells are: [0,1,1,2]
# The answer [[0,1],[
# 1,1],[0,0],[1,0]] would also be accepted as correct.
# Input: R = 1, C = 2, r0 = 0, c0 = 0
# Output: [[0,0],[0,1]]
# Explanation: The dista
# nces from (r0, c0) to other cells are: [0,1]

# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def allCellsDistOrder(self, R, C, r0, c0):
        """
        :type R: int
        :type C: int
        :type r0: int
        :type c0: int
        :rtype: List[List[int]]
        """
        def append(R, C, r, c, result):
            if 0 <= r < R and 0 <= c < C:
                result.append([r, c])

        result = [[r0, c0]]
        for d in xrange(1, R+C):
            append(R, C, r0-d, c0, result)
```

```
    for x in xrange(-d+1, d):
        append(R, C, r0+x, c0+abs(x)-d, result)
        append(R, C, r0+x, c0+d-abs(x), result)
    append(R, C, r0+d, c0, result)
return result
```

best-position-for-a-service-centre.py

```python
# best-position-for-a-service-centre is not found.
# Time:  O(n * iter), iter is the number of iterations
# Space: O(1)

# see reference:
# - https://en.wikipedia.org/wiki/Geometric_median
# - https://wikimedia.org/api/rest_v1/media/math/render/svg/b3fb215363358f12687100710caff0e86cd9d26b
# Weiszfeld's algorithm
class Solution(object):
    def getMinDistSum(self, positions):
        """
        :type positions: List[List[int]]
        :rtype: float
        """
        EPS = 1e-6
        def norm(p1, p2):
            return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

        def geometry_median(positions, median):
            numerator, denominator = [0.0, 0.0], 0.0
            for p in positions:
                l = norm(median, p)
                if not l:
                    continue
                numerator[0] += p[0]/l
                numerator[1] += p[1]/l
                denominator += 1/l
            if denominator == 0.0:
                return True, None
            return False, [numerator[0]/denominator, numerator[1]/denominator]

        median = [float(sum(p[0] for p in positions))/len(positions),
                  float(sum(p[1] for p in positions))/len(positions)]
        prev_median = [float("-inf"), float("-inf")]
        while norm(median, prev_median)*len(positions) > EPS:
            stopped, new_median = geometry_median(positions, median)
            if stopped:
                break
            median, prev_median = new_median, median
        return sum(norm(median, p) for p in positions)


# Time:  O(n * iter), iter is the number of iterations
# Space: O(1)
class Solution2(object):
    def getMinDistSum(self, positions):
        """
        :type positions: List[List[int]]
        :rtype: float
        """
        DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        EPS = 1e-6
        def dist(positions, p):
            return sum(((p[0]-x)**2 + (p[1]-y)**2)**0.5 for x, y in positions)

        median = [0.0, 0.0]
        median[0] = float(sum(x for x, _ in positions))/len(positions)
        median[1] = float(sum(y for _, y in positions))/len(positions)
```

```
result = dist(positions, median)
delta = float(max(max(positions, key=lambda x: x[0])[0],
                  max(positions, key=lambda x: x[1])[1])) - \
        float(min(min(positions, key=lambda x: x[0])[0],
                  min(positions, key=lambda x: x[1])[1]))
while delta > EPS:
    for dx, dy in DIRECTIONS:
        new_median = [median[0] + delta*dx, median[1] + delta*dy]
        nd = dist(positions, new_median)
        if nd < result:
            result = nd
            median = new_median
            break
    else:
        delta /= 2.0
return result
```

minimum-ascii-delete-sum-for-two-strings.py

```
# DESC
# Example 1:
# Example 2:
# Note:
# Given two strings s1, s2, find the lowest ASCII sum of deleted characters to mak
# e two strings equal.

# NOTE
# All elements of each string will have an ASCII value in [97, 122].
# 0 < s1.length, s2.length <= 1000.

# EXAMPLE
# Input: s1 = "sea", s2 = "eat"
# Output: 231
# Explanation: Deleting "s" from "sea" a
# dds the ASCII value of "s" (115) to the sum.
# Deleting "t" from "eat" adds 116 to
#  the sum.
# At the end, both strings are equal, and 115 + 116 = 231 is the minimum
#  sum possible to achieve this.
# Input: s1 = "delete", s2 = "leet"
# Output: 403
# Explanation: Deleting "dee" from "
# delete" to turn the string into "let",
# adds 100[d]+101[e]+101[e] to the sum.  De
# leting "e" from "leet" adds 101[e] to the sum.
# At the end, both strings are equa
# l to "let", and the answer is 100+101+101+101 = 403.
# If instead we turned both s
# trings into "lee" or "eet", we would get answers of 433 or 417, which are higher
# .

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

class Solution(object):
    def minimumDeleteSum(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: int
        """
        dp = [[0] * (len(s2)+1) for _ in xrange(2)]
        for j in xrange(len(s2)):
            dp[0][j+1] = dp[0][j] + ord(s2[j])

        for i in xrange(len(s1)):
            dp[(i+1)%2][0] = dp[i%2][0] + ord(s1[i])
            for j in xrange(len(s2)):
                if s1[i] == s2[j]:
                    dp[(i+1)%2][j+1] = dp[i%2][j]
                else:
                    dp[(i+1)%2][j+1] = min(dp[i%2][j+1] + ord(s1[i]), \
                                           dp[(i+1)%2][j] + ord(s2[j]))

        return dp[len(s1)%2][-1]
```

```
# Time:  O(m * n)
# Space: O(m * n)
class Solution2(object):
    def minimumDeleteSum(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: int
        """
        dp = [[0] * (len(s2)+1) for _ in xrange(len(s1)+1)]
        for i in xrange(len(s1)):
            dp[i+1][0] = dp[i][0] + ord(s1[i])
        for j in xrange(len(s2)):
            dp[0][j+1] = dp[0][j] + ord(s2[j])

        for i in xrange(len(s1)):
            for j in xrange(len(s2)):
                if s1[i] == s2[j]:
                    dp[i+1][j+1] = dp[i][j]
                else:
                    dp[i+1][j+1] = min(dp[i][j+1] + ord(s1[i]), \
                                       dp[i+1][j] + ord(s2[j]))

        return dp[-1][-1]
```

shortest-bridge.py

```
# DESC
# Return the smallest number of 0s that must be flipped.  (It is guaranteed that t
# he answer is at least 1.)
# Example 3:
# Constraints:
# In a given 2D binary array A, there are two islands.  (An island is a 4-directio
# nally connected group of 1s not connected to any other 1s.)
# Example 1:
# Now, we may change 0s to 1s so as to connect the two islands together to form 1
# island.
# Example 2:

# NOTE
# 2 <= A.length == A[0].length <= 100
# A[i][j] == 0 or A[i][j] == 1

# EXAMPLE
# 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
# Input: A = [[0,1],[1,0]]
# Output: 1
# Input: A = [[0,1,0],[0,0,0],[0,0,1]]
# Output: 2

# 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)
```

largest-perimeter-triangle.py

```
# DESC
# If it is impossible to form any triangle of non-zero area, return 0.
# Example 4:
# Example 3:
# Note:
# Given an array A of positive lengths, return the largest perimeter of a triangle
#  with non-zero area, formed from 3 of these lengths.
# Example 1:
# Example 2:

# NOTE
# 1 <= A[i] <= 10^6
# 3 <= A.length <= 10000

# EXAMPLE
# Input: [3,6,2,3]
# Output: 8
# Input: [1,2,1]
# Output: 0
# Input: [3,2,3,4]
# Output: 10
# Input: [2,1,2]
# Output: 5

# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def largestPerimeter(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        A.sort()
        for i in reversed(xrange(len(A) - 2)):
            if A[i] + A[i+1] > A[i+2]:
                return A[i] + A[i+1] + A[i+2]
        return 0
```

all-possible-full-binary-trees.py

```
# DESC
# Note:
# Example 1:
# A full binary tree is a binary tree where each node has exactly 0 or 2 children.
# Return a list of all possible full binary trees with N nodes.  Each element of t
# he answer is the root node of one possible tree.
# You may return the final list of trees in any order.
# Each node of each tree in the answer must have node.val = 0.

# NOTE
# 1 <= N <= 20

# EXAMPLE
# Input: 7
# Output: [[0,0,0,null,null,0,0,null,null,0,0],[0,0,0,null,null,0,0,0,0],
# [0,0,0,0,0,0,0],[0,0,0,0,0,null,null,null,null,0,0],[0,0,0,0,0,null,null,0,0]]
# E
# xplanation:

# Time:  O(n * 4^n / n^(3/2)) ~= sum of Catalan numbers from 1 .. N
# Space: O(n * 4^n / n^(3/2)) ~= sum of Catalan numbers from 1 .. N

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


class Solution(object):
    def __init__(self):
        self.__memo = {1: [TreeNode(0)]}

    def allPossibleFBT(self, N):
        """
        :type N: int
        :rtype: List[TreeNode]
        """
        if N % 2 == 0:
            return []

        if N not in self.__memo:
            result = []
            for i in xrange(N):
                for left in self.allPossibleFBT(i):
                    for right in self.allPossibleFBT(N-1-i):
                        node = TreeNode(0)
                        node.left = left
                        node.right = right
                        result.append(node)
            self.__memo[N] = result

        return self.__memo[N]
```

find-elements-in-a-contaminated-binary-tree.py

```python
# find-elements-in-a-contaminated-binary-tree is not found.
# 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 FindElements(object):

    def __init__(self, root):
        """
        :type root: TreeNode
        """
        def dfs(node, v, lookup):
            if not node:
                return
            node.val = v
            lookup.add(v)
            dfs(node.left, 2*v+1, lookup)
            dfs(node.right, 2*v+2, lookup)

        self.__lookup = set()
        dfs(root, 0, self.__lookup)

    def find(self, target):
        """
        :type target: int
        :rtype: bool
        """
        return target in self.__lookup
```

remove-outermost-parentheses.py

```
# DESC
# Example 2:
# Example 1:
# A valid parentheses string is either empty (""), "(" + A + ")", or A + B, where
# A and B are valid parentheses strings, and + represents string concatenation.  F
# or example, "", "()", "(())()", and "(()(()))" are all valid parentheses strings
# .
# Return S after removing the outermost parentheses of every primitive string in t
# he primitive decomposition of S.
# Given a valid parentheses string S, consider its primitive decomposition: S = P_
# 1 + P_2 + ... + P_k, where P_i are primitive valid parentheses strings.
# Example 3:
# A valid parentheses string S is primitive if it is nonempty, and there does not
# exist a way to split it into S = A+B, with A and B nonempty valid parentheses st
# rings.
# Note:

# NOTE
# S[i] is "(" or ")"
# S.length <= 10000
# S is a valid parentheses string

# EXAMPLE
# Input: "()()"
# Output: ""
# Explanation:
# The input string is "()()", with primitiv
# e decomposition "()" + "()".
# After removing outer parentheses of each part, this
#  is "" + "" = "".
# Input: "(()())(())(()(()))"
# Output: "()()()()(())"
# Explanation:
# The input strin
# g is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()
# (()))".
# After removing outer parentheses of each part, this is "()()" + "()" + "
# (()())" = "()()()()(())".
# Input: "(()())(())"
# Output: "()()()"
# Explanation:
# The input string is "(()())((
# ))", with primitive decomposition "(()())" + "(())".
# After removing outer parent
# heses of each part, this is "()()" + "()" = "()()()".

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def removeOuterParentheses(self, S):
        """
        :type S: str
        :rtype: str
        """
        deep = 1
        result, cnt = [], 0
        for c in S:
```

```
    if c == '(' and cnt >= deep:
        result.append(c)
    if c == ')' and cnt > deep:
        result.append(c)
    cnt += 1 if c == '(' else -1
return "".join(result)
```

complex-number-multiplication.py

```python
# DESC
# You need to return a string representing their multiplication. Note i2 = -1 acco
# rding to the definition.
# Given two strings representing two complex numbers.
# Example 2:
# Example 1:
# Note:

# 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 w
# ill both belong to the range of [-100, 100]. And the output should be also in th
# is form.

# EXAMPLE
# 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.
# 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.

# 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

```
# DESC
# 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.
# Example 4:
# Example 2:
# Constraints:
# Example 1:
# Example 3:
# The Linked List is represented in the input/output as a list of n nodes. Each no
# de is represented as a pair of [val, random_index] where:

# NOTE
# Number of Nodes will not exceed 1000.
# Node.random is null or pointing to a node in the linked list.
# val: an integer representing Node.val
# random_index: the index of the node (range from 0 to n-1) where random pointer p
# oints to, or null if it does not point to any node.
# -10000 <= Node.val <= 10000

# EXAMPLE
# Input: head = []
# Output: []
# Explanation: Given linked list is empty (null pointe
# r), so return null.
# Input: head = [[3,null],[3,0],[3,null]]
# Output: [[3,null],[3,0],[3,null]]
# Input: head = [[1,1],[2,1]]
# Output: [[1,1],[2,1]]
# Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
# Output: [[7,null],[13,0],[11
# ,4],[10,2],[1,0]]

# 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):
    # @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
        """
        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]
```

nth-magical-number.py

```
# DESC
# Example 4:
# Example 3:
# Example 2:
# Note:
# Return the N-th magical number.  Since the answer may be very large, return it m
# odulo 10^9 + 7.
# A positive integer is magical if it is divisible by either A or B.
# Example 1:
# 10^9 + 7

# NOTE
# 1 <= N <= 10^9
# 2 <= A <= 40000
# 2 <= B <= 40000

# EXAMPLE
# Input: N = 4, A = 2, B = 3
# Output: 6
# Input: N = 5, A = 2, B = 4
# Output: 10
# Input: N = 1, A = 2, B = 3
# Output: 2
# Input: N = 3, A = 6, B = 4
# Output: 8

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def nthMagicalNumber(self, N, A, B):
        """
        :type N: int
        :type A: int
        :type B: int
        :rtype: int
        """
        def gcd(a, b):
            while b:
                a, b = b, a % b
            return a

        def check(A, B, N, lcm, target):
            return target//A + target//B - target//lcm >= N

        lcm = A*B // gcd(A, B)
        left, right = min(A, B), max(A, B)*N
        while left <= right:
            mid = left + (right-left)//2
            if check(A, B, N, lcm, mid):
                right = mid-1
            else:
                left = mid+1
        return left % (10**9 + 7)
```

container-with-most-water.py

```
# DESC
# The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this c
# ase, the max area of water (blue section) the container can contain is 49.
# Note: You may not slant the container and n is at least 2.
# Example:
# 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 l
# ine 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
#

# 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
```

fizz-buzz.py

```
# DESC
# But for multiples of three it should output "Fizz" instead of the number and for
#  the multiples of five output "Buzz". For numbers which are multiples of both th
# ree and five output "FizzBuzz".
# Example:
# Write a program that outputs the string representation of numbers from 1 to n.

# NOTE
#

# EXAMPLE
# n = 15,
#
# Return:
# [
#     "1",
#     "2",
#     "Fizz",
#     "4",
#     "Buzz",
#     "Fizz"
# ,
#     "7",
#     "8",
#     "Fizz",
#     "Buzz",
#     "11",
#     "Fizz",
#     "13",
#
# "14",
#     "FizzBuzz"
# ]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def fizzBuzz(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        result = []

        for i in xrange(1, n+1):
            if i % 15 == 0:
                result.append("FizzBuzz")
            elif i % 5 == 0:
                result.append("Buzz")
            elif i % 3 == 0:
                result.append("Fizz")
            else:
                result.append(str(i))

        return result

    def fizzBuzz2(self, n):
```

```
        """
        :type n: int
        :rtype: List[str]
        """
        l = [str(x) for x in range(n + 1)]
        l3 = range(0, n + 1, 3)
        l5 = range(0, n + 1, 5)
        for i in l3:
            l[i] = 'Fizz'
        for i in l5:
            if l[i] == 'Fizz':
                l[i] += 'Buzz'
            else:
                l[i] = 'Buzz'
        return l[1:]

    def fizzBuzz3(self, n):
        return ['Fizz' * (not i % 3) + 'Buzz' * (not i % 5) or str(i) for i in range(1, n + 1)]

    def fizzBuzz4(self, n):
        return ['FizzBuzz'[i % -3 & -4:i % -5 & 8 ^ 12] or repr(i) for i in range(1, n + 1)]
```

find-duplicate-file-in-system.py

```
# DESC
# 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 fil
# es in the file system in terms of their paths.
# Note:
# Example 1:
# A single directory info string in the input list has the following format:
# The output is a list of group of duplicate file paths. For each group, it contai
# ns all the file paths of the files that have the same content. A file path is a
# string that has the following format:
# "root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ... fn.txt(fn_content)"
# A group of duplicate files consists of at least two files that have exactly the
# same content.
# It means there are n files (f1.txt, f2.txt ... fn.txt with content f1_content, f
# 2_content ... fn_content, respectively) in directory root/d1/d2/.../dm. Note tha
# t n >= 1 and m >= 0. If m = 0, it means the directory is just the root directory
# .
# "directory_path/file_name.txt"

# NOTE
# You may assume the directory name, file name and file content only has letters a
# nd digits, and the length of file content is in the range of [1,50].
# How to make sure the duplicated files you find are not false positive?
# Imagine you are given a real file system, how will you search files? DFS or BFS?
# If you can only read the file by 1kb each time, how will you modify your solution?
# You may assume each given directory info represents a unique directory. Director
# y path and file info are separated by a single blank space.
# No order is required for the final output.
# What is the time complexity of your modified solution? What is the most time-con
# suming part and memory consuming part of it? How to optimize?
# If the file content is very large (GB level), how will you modify your solution?
# 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.

# EXAMPLE
# 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"]]

# Time:  O(n * l), l is the average length of file content
# 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
```

implement-rand10-using-rand7.py

```python
# implement-rand10-using-rand7 is not found.
# Time:  O(1.189), counted by statistics, limit would be O(log10/log7) = O(1.183)
# Space: O(1)

import random


def rand7():
    return random.randint(1, 7)


# Reference: https://leetcode.com/problems/implement-rand10-using-rand7/discuss/151567/C++JavaPython-Average-1
class Solution(object):
    def __init__(self):
        self.__cache = []

    def rand10(self):
        """
        :rtype: int
        """
        def generate(cache):
            n = 32
            curr = sum((rand7()-1) * (7**i) for i in xrange(n))
            rang = 7**n
            while curr < rang//10*10:
                cache.append(curr%10+1)
                curr /= 10
                rang /= 10

        while not self.__cache:
            generate(self.__cache)
        return self.__cache.pop()


# Time:  O(2 * (1 + (9/49) + (9/49)^2 + ...)) = O(2/(1-(9/49)) = O(2.45)
# Space: O(1)
class Solution2(object):
    def rand10(self):
        """
        :rtype: int
        """
        while True:
            x = (rand7()-1)*7 + (rand7()-1)
            if x < 40:
                return x%10 + 1
```

fraction-to-recurring-decimal.py

```python
# DESC
# If multiple answers are possible, just return any of them.
# Example 2:
# If the fractional part is repeating, enclose the repeating part in parentheses.
# Given two integers representing the numerator and denominator of a fraction, ret
# urn the fraction in string format.
# Example 1:
# Example 3:

# NOTE
#

# EXAMPLE
# Input: numerator = 1, denominator = 2
# Output: "0.5"
# Input: numerator = 2, denominator = 1
# Output: "2"
# 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
```

find-the-shortest-superstring.py

```
# DESC
# Given an array A of strings, find any smallest string that contains each string
# in A as a substring.
# Example 1:
# Note:
# We may assume that no string in A is substring of another string in A.
# Example 2:

# NOTE
# 1 <= A.length <= 12
# 1 <= A[i].length <= 20

# EXAMPLE
# Input: ["alex","loves","leetcode"]
# Output: "alexlovesleetcode"
# Explanation: All
# permutations of "alex","loves","leetcode" would also be accepted.
# Input: ["catg","ctaagt","gcta","ttca","atgcatc"]
# Output: "gctaagttcatgcatc"

# Time:  O(n^2 * (l^2 + 2^n))
# Space: O(n^2)

class Solution(object):
    def shortestSuperstring(self, A):
        """
        :type A: List[str]
        :rtype: str
        """
        n = len(A)
        overlaps = [[0]*n for _ in xrange(n)]
        for i, x in enumerate(A):
            for j, y in enumerate(A):
                for l in reversed(xrange(min(len(x), len(y)))):
                    if y[:l].startswith(x[len(x)-l:]):
                        overlaps[i][j] = l
                        break

        dp = [[0]*n for _ in xrange(1<<n)]
        prev = [[None]*n for _ in xrange(1<<n)]
        for mask in xrange(1, 1<<n):
            for bit in xrange(n):
                if ((mask>>bit) & 1) == 0:
                    continue
                prev_mask = mask^(1<<bit)
                for i in xrange(n):
                    if ((prev_mask>>i) & 1) == 0:
                        continue
                    value = dp[prev_mask][i] + overlaps[i][bit]
                    if value > dp[mask][bit]:
                        dp[mask][bit] = value
                        prev[mask][bit] = i

        bit = max(xrange(n), key = dp[-1].__getitem__)
        words = []
        mask = (1<<n)-1
        while bit is not None:
            words.append(bit)
```

```
    mask, bit = mask^(1<<bit), prev[mask][bit]
words.reverse()
lookup = set(words)
words.extend([i for i in xrange(n) if i not in lookup])

result = [A[words[0]]]
for i in xrange(1, len(words)):
    overlap = overlaps[words[i-1]][words[i]]
    result.append(A[words[i]][overlap:])
return "".join(result)
```

path-with-maximum-probability.py

```
# path-with-maximum-probabilit is not found.
# Time:  O((|E| + |V|) * log|V|) = O(|E| * log|V|) by using binary heap,
#        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


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]
```

longest-continuous-increasing-subsequence.py

```
# DESC
# Example 1:
# Given an unsorted array of integers, find the length of longest continuous incre
# asing subsequence (subarray).
# Note:
# Length of the array will not exceed 10,000.
# Example 2:

# NOTE
#

# EXAMPLE
# Input: [1,3,5,4,7]
# Output: 3
# Explanation: The longest continuous increasing subs
# equence is [1,3,5], its length is 3.
# Even though [1,3,5,7] is also an increasin
# g subsequence, it's not a continuous one where 5 and 7 are separated by 4.
# Input: [2,2,2,2,2]
# Output: 1
# Explanation: The longest continuous increasing subs
# equence is [2], its length is 1.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findLengthOfLCIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, count = 0, 0
        for i in xrange(len(nums)):
            if i == 0 or nums[i-1] < nums[i]:
                count += 1
                result = max(result, count)
            else:
                count = 1
        return result
```

house-robber-iii.py

```
# DESC
# Example 2:
# The thief has found himself a new place for his thievery again. There is only on
# e 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 hou
# ses 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.
# Example 1:
# Determine the maximum amount of money the thief can rob tonight without alerting
#  the police.

# NOTE
#

# EXAMPLE
# 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.
# Input: [3,2,3,null,3,null,1]
#
#      3
#     / \
#    2   3
#     \   \
#      3   1
#
# Out
# put: 7
# Explanation: Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

# 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

```
# DESC
# You have a set of 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.
# Note:
# Example 2:
# Example 1:

# NOTE
# 1 <= tiles.length <= 7
# tiles consists of uppercase English letters.

# EXAMPLE
# Input: "AAABBC"
# Output: 188
# Input: "AAB"
# Output: 8
# Explanation: The possible sequences are "A", "B", "AA", "
# AB", "BA", "AAB", "ABA", "BAA".

# 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!

        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))
```

n-repeated-element-in-size-2n-array.py

```
# n-repeated-element-in-size-2n-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def repeatedNTimes(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        for i in xrange(2, len(A)):
            if A[i-1] == A[i] or A[i-2] == A[i]:
                return A[i]
        return A[0]
```

grid-illumination.py

```
# DESC
# Return an array of answers.  Each value answer[i] should be equal to the answer
# of the i-th query queries[i].
# Note:
# lamps[i]
# Initially, some number of lamps are on.  lamps[i] tells us the location of the i
# -th lamp that is on.  Each lamp that is on illuminates every square on its x-axi
# s, y-axis, and both diagonals (similar to a Queen in chess).
# On a N x N grid of cells, each cell (x, y) with 0 <= x < N and 0 <= y < N has a lamp.
# After each query (x, y) [in the order given by queries], we turn off any lamps t
# hat are at cell (x, y) or are adjacent 8-directionally (ie., share a corner or e
# dge with cell (x, y).)
# For the i-th query queries[i] = (x, y), the answer to the query is 1 if the cell
#  (x, y) is illuminated, else 0.
# Example 1:

# NOTE
# 1 <= N <= 10^9
# 0 <= lamps.length <= 20000
# 0 <= queries.length <= 20000
# lamps[i].length == queries[i].length == 2

# EXAMPLE
# Input: N = 5, lamps = [[0,0],[4,4]], queries = [[1,1],[1,0]]
# Output: [1,0]
# Expla
# nation:
# Before performing the first query we have both lamps [0,0] and [4,4] on
# .
# The grid representing which cells are lit looks like this, where [0,0] is the
# top left corner, and [4,4] is the bottom right corner:
# 1 1 1 1 1
# 1 1 0 0 1
# 1 0 1
#  0 1
# 1 0 0 1 1
# 1 1 1 1 1
# Then the query at [1, 1] returns 1 because the cell is
# lit.  After this query, the lamp at [0, 0] turns off, and the grid now looks lik
# e this:
# 1 0 0 0 1
# 0 1 0 0 1
# 0 0 1 0 1
# 0 0 0 1 1
# 1 1 1 1 1
# Before performing the
# second query we have only the lamp [4,4] on.  Now the query at [1,0] returns 0,
# because the cell is no longer lit.

# Time:  O(l + q)
# Space: O(l)

import collections


class Solution(object):
    def gridIllumination(self, N, lamps, queries):
        """
```

```
    :type N: int
    :type lamps: List[List[int]]
    :type queries: List[List[int]]
    :rtype: List[int]
    """
    directions = [(0, -1), (0, 1), (-1, 0), (1, 0),
                  (-1, -1), (1, -1), (1, -1), (1, 1)]

    lookup = set()
    row = collections.defaultdict(int)
    col = collections.defaultdict(int)
    diag = collections.defaultdict(int)
    anti = collections.defaultdict(int)

    for r, c in lamps:
        lookup.add((r, c))
        row[r] += 1
        col[c] += 1
        diag[r-c] += 1
        anti[r+c] += 1

    result = []
    for r, c in queries:
        if row[r] or col[c] or \
           diag[r-c] or anti[r+c]:
            result.append(1)
        else:
            result.append(0)
        for d in directions:
            nc, nr = r+d[0], c+d[1]
            if not (0 <= nr < N and 0 <= nc < N and \
                    (nr, nc) in lookup):
                continue
            lookup.remove((nr, nc))
            row[nr] -= 1
            col[nc] -= 1
            diag[nr-nc] -= 1
            anti[nr+nc] -= 1
    return result
```

number-of-distinct-subarrays-with-at-most-k-odd-integers.py

```
# number-of-distinct-subarrays-with-at-most-k-odd-integers is not found.
# Time:  O(n^2)
# Space: O(t), t is the size of trie

import collections


# sliding window + trie solution
class Solution(object):
    def distinctSubarraysWithAtMostKOddIntegers(self, A, K):
        def countDistinct(A, left, right, trie):  # Time: O(n), Space: O(t)
            result = 0
            for i in reversed(xrange(left, right+1)):
                if A[i] not in trie:
                    result += 1
                trie = trie[A[i]]
            return result

        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        result, left, count = 0, 0, 0
        for right in xrange(len(A)):
            count += A[right]%2
            while count > K:
                count -= A[left]%2
                left += 1
            result += countDistinct(A, left, right, trie)
        return result


# Time:  O(n^2)
# Space: O(t), t is the size of trie
# suffix tree solution
class Solution2(object):
    def distinctSubarraysWithAtMostKOddIntegers(self, A, K):
        def countDistinct(A, left, right, trie):  # Time: O(n), Space: O(t)
            result = 0
            for i in xrange(left, right+1):
                if A[i] not in trie:
                    result += 1
                trie = trie[A[i]]
            return result

        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        result = 0
        for left in xrange(len(A)):
            count = 0
            for right in xrange(left, len(A)):
                count += A[right]%2
                if count > K:
                    right -= 1
                    break
            result += countDistinct(A, left, right, trie)
        return result
```

longest-mountain-in-array.py

```python
# longest-mountain-in-arra is not found.
# Time:  O(n)
# Space: O(1)


class Solution(object):
    def longestMountain(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result, up_len, down_len = 0, 0, 0
        for i in xrange(1, len(A)):
            if (down_len and A[i-1] < A[i]) or A[i-1] == A[i]:
                up_len, down_len = 0, 0
            up_len += A[i-1] < A[i]
            down_len += A[i-1] > A[i]
            if up_len and down_len:
                result = max(result, up_len+down_len+1)
        return result
```

sort-an-array.py

```
# sort-an-arra is not found.
# 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
```

divide-array-in-sets-of-k-consecutive-numbers.py

```python
# divide-array-in-sets-of-k-consecutive-numbers is not found.
# Time:  O(nlogn)
# Space: O(n)

import collections


class Solution(object):
    def isPossibleDivide(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: bool
        """
        count = collections.Counter(nums)
        for num in sorted(count.keys()):
            c = count[num]
            if not c:
                continue
            for i in xrange(num, num+k):
                if count[i] < c:
                    return False
                count[i] -= c
        return True
```

chalkboard-xor-game.py

```
# DESC
# We are given non-negative integers nums[i] which are written on a chalkboard.  A
# lice and Bob take turns erasing exactly one number from the chalkboard, with Ali
# ce starting first.  If erasing a number causes the bitwise XOR of all the elemen
# ts of the chalkboard to become 0, then that player loses.  (Also, we'll say the
# bitwise XOR of one element is that element itself, and the bitwise XOR of no ele
# ments is 0.)
# Also, if any player starts their turn with the bitwise XOR of all the elements o
# f the chalkboard equal to 0, then that player wins.
# Return True if and only if Alice wins the game, assuming both players play optimally.
# Notes:

# NOTE
# 1 <= N <= 1000.
# 0 <= nums[i] <= 2^16.

# EXAMPLE
# Example:
# Input: nums = [1, 1, 2]
# Output: false
# Explanation:
# Alice has two choic
# es: erase 1 or erase 2.
# If she erases 1, the nums array becomes [1, 2]. The bit
# wise XOR of all the elements of the chalkboard is 1 XOR 2 = 3. Now Bob can remov
# e any element he wants, because Alice will be the one to erase the last element
# and she will lose.
# If Alice erases 2 first, now nums becomes [1, 1]. The bitwis
# e XOR of all the elements of the chalkboard is 1 XOR 1 = 0. Alice will lose.

# Time:  O(n)
# Space: O(1)

from operator import xor
from functools import reduce


class Solution(object):
    def xorGame(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        return reduce(xor, nums) == 0 or \
            len(nums) % 2 == 0
```

two-sum-iv-input-is-a-bst.py

```python
# two-sum-iv-input-is-a-bst is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def findTarget(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: bool
        """
        class BSTIterator(object):
            def __init__(self, root, forward):
                self.__node = root
                self.__forward = forward
                self.__s = []
                self.__cur = None
                self.next()

            def val(self):
                return self.__cur

            def next(self):
                while self.__node or self.__s:
                    if self.__node:
                        self.__s.append(self.__node)
                        self.__node = self.__node.left if self.__forward else self.__node.right
                    else:
                        self.__node = self.__s.pop()
                        self.__cur = self.__node.val
                        self.__node = self.__node.right if self.__forward else self.__node.left
                        break


        if not root:
            return False
        left, right = BSTIterator(root, True), BSTIterator(root, False)
        while left.val() < right.val():
            if left.val() + right.val() == k:
                return True
            elif left.val() + right.val() < k:
                left.next()
            else:
                right.next()
        return False
```

parallel-courses-ii.py

```
# parallel-courses-ii is not found.
# Time:  O((n * C(c, min(c, k))) * 2^n)
# Space: O(2^n)

import itertools


class Solution(object):
    def minNumberOfSemesters(self, n, dependencies, k):
        """
        :type n: int
        :type dependencies: List[List[int]]
        :type k: int
        :rtype: int
        """
        reqs = [0]*n
        for u, v in dependencies:
            reqs[v-1] |= 1 << (u-1)
        dp = [n]*(1<<n)
        dp[0] = 0
        for mask in xrange(1<<n):
            candidates = []
            for v in xrange(n):
                if (mask&(1<<v)) == 0 and (mask&reqs[v]) == reqs[v]:
                    candidates.append(v)
            for choice in itertools.combinations(candidates, min(len(candidates), k)):
                new_mask = mask
                for v in choice:
                    new_mask |= 1<<v
                dp[new_mask] = min(dp[new_mask], dp[mask]+1)
        return dp[-1]


# Time:  O(nlogn + e), e is the number of edges in graph
# Space: O(n + e)
import collections
import heapq

# wrong greedy solution
# since the priority of courses are hard to decide especially for those courses with zero indegrees are of the
# e.x.
# 9
# [[1,4],[1,5],[3,5],[3,6],[2,6],[2,7],[8,4],[8,5],[9,6],[9,7]]
# 3
class Solution_WA(object):
    def minNumberOfSemesters(self, n, dependencies, k):
        """
        :type n: int
        :type dependencies: List[List[int]]
        :type k: int
        :rtype: int
        """
        def dfs(graph, i, depths):
            if depths[i] == -1:
                depths[i] = max(dfs(graph, child, depths) for child in graph[i])+1 if i in graph else 1
            return depths[i]

        degrees = [0]*n
```

```
graph = collections.defaultdict(list)
for u, v in dependencies:
    graph[u-1].append(v-1)
    degrees[v-1] += 1
depths = [-1]*n
for i in xrange(n):
    dfs(graph, i, depths)
max_heap = []
for i in xrange(n):
    if not degrees[i]:
        heapq.heappush(max_heap, (-depths[i], i))
result = 0
while max_heap:
    new_q = []
    for _ in xrange(min(len(max_heap), k)):
        _, node = heapq.heappop(max_heap)
        if node not in graph:
            continue
        for child in graph[node]:
            degrees[child] -= 1
            if not degrees[child]:
                new_q.append(child)
    result += 1
    for node in new_q:
        heapq.heappush(max_heap, (-depths[node], node))
return result
```

one-edit-distance.py

```python
# one-edit-distance is not found.
# Time:  O(m + n)
# Space: O(1)

class Solution(object):
    def isOneEditDistance(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        m, n = len(s), len(t)
        if m > n:
            return self.isOneEditDistance(t, s)
        if n - m > 1:
            return False

        i, shift = 0, n - m
        while i < m and s[i] == t[i]:
            i += 1
        if shift == 0:
            i += 1
        while i < m and s[i] == t[i + shift]:
            i += 1

        return i == m
```

search-suggestions-system.py

```
# search-suggestions-system is not found.
# Time:  ctor: O(n * l), n is the number of products
#                      , l is the average length of product name
#        suggest: O(l^2)
# Space: O(t), t is the number of nodes in trie

import collections


class TrieNode(object):

    def __init__(self):
        self.__TOP_COUNT = 3
        self.leaves = collections.defaultdict(TrieNode)
        self.infos = []

    def insert(self, words, i):
        curr = self
        for c in words[i]:
            curr = curr.leaves[c]
            curr.add_info(words, i)

    def add_info(self, words, i):
        self.infos.append(i)
        self.infos.sort(key=lambda x: words[x])
        if len(self.infos) > self.__TOP_COUNT:
            self.infos.pop()


class Solution(object):
    def suggestedProducts(self, products, searchWord):
        """
        :type products: List[str]
        :type searchWord: str
        :rtype: List[List[str]]
        """
        trie = TrieNode()
        for i in xrange(len(products)):
            trie.insert(products, i)
        result = [[] for _ in xrange(len(searchWord))]
        for i, c in enumerate(searchWord):
            if c not in trie.leaves:
                break
            trie = trie.leaves[c]
            result[i] = map(lambda x: products[x], trie.infos)
        return result


# Time:  ctor: O(n * l * log(n * l)), n is the number of products
#                                   , l is the average length of product name
#        suggest: O(l^2)
# Space: O(t), t is the number of nodes in trie
class TrieNode2(object):

    def __init__(self):
        self.__TOP_COUNT = 3
        self.leaves = collections.defaultdict(TrieNode2)
        self.infos = []
```

```
    def insert(self, words, i):
        curr = self
        for c in words[i]:
            curr = curr.leaves[c]
            curr.add_info(i)

    def add_info(self, i):
        if len(self.infos) == self.__TOP_COUNT:
            return
        self.infos.append(i)


class Solution2(object):
    def suggestedProducts(self, products, searchWord):
        """
        :type products: List[str]
        :type searchWord: str
        :rtype: List[List[str]]
        """
        products.sort()
        trie = TrieNode2()
        for i in xrange(len(products)):
            trie.insert(products, i)
        result = [[] for _ in xrange(len(searchWord))]
        for i, c in enumerate(searchWord):
            if c not in trie.leaves:
                break
            trie = trie.leaves[c]
            result[i] = map(lambda x: products[x], trie.infos)
        return result


# Time:  ctor: O(n * l * log(n * l)), n is the number of products
#                                   , l is the average length of product name
#        suggest: O(l^2 * n)
# Space: O(n * l)
import bisect


class Solution3(object):
    def suggestedProducts(self, products, searchWord):
        """
        :type products: List[str]
        :type searchWord: str
        :rtype: List[List[str]]
        """
        products.sort()  # Time: O(n * l * log(n * l))
        result = []
        prefix = ""
        for i, c in enumerate(searchWord):  # Time: O(l)
            prefix += c
            start = bisect.bisect_left(products, prefix)  # Time: O(log(n * l))
            new_products = []
            for j in xrange(start, len(products)):  # Time: O(n * l)
                if not (i < len(products[j]) and products[j][i] == c):
                    break
                new_products.append(products[j])
            products = new_products
            result.append(products[:3])
```

```
    return result
```

the-dining-philosophers.py

```python
# the-dining-philosophers is not found.
# 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()
```

minimum-absolute-difference-in-bst.py

```python
# DESC
# Given a binary search tree with non-negative values, find the minimum absolute d
# ifference between values of any two nodes.
# Example:
# Note:

# NOTE
# This question is the same as 783: https://leetcode.com/problems/minimum-distance
# -between-bst-nodes/
# There are at least two nodes in this BST.

# EXAMPLE
# Input:
#
#    1
#     \
#      3
#     /
#    2
#
# Output:
# 1
#
# Explanation:
# The minimum absolu
# te difference is 1, which is the difference between 2 and 1 (or between 2 and 3)
# .

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

class Solution(object):
    def getMinimumDifference(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def inorderTraversal(root, prev, result):
            if not root:
                return (result, prev)

            result, prev = inorderTraversal(root.left, prev, result)
            if prev: result = min(result, root.val - prev.val)
            return inorderTraversal(root.right, root, result)

        return inorderTraversal(root, None, float("inf"))[0]
```

find-k-th-smallest-pair-distance.py

```
# DESC
# Example 1:
# Given an integer array, return the k-th smallest distance among all the pairs. T
# he distance of a pair (A, B) is defined as the absolute difference between A and
#  B.
# Note:

# NOTE
# 1 <= k <= len(nums) * (len(nums) - 1) / 2.
# 2 <= len(nums) <= 10000.
# 0 <= nums[i] < 1000000.

# EXAMPLE
# Input:
# nums = [1,3,1]
# k = 1
# Output: 0
# Explanation:
# Here are all the pairs:
# (1,3
# ) -> 2
# (1,1) -> 0
# (3,1) -> 2
# Then the 1st smallest distance pair is (1,1), and i
# ts distance is 0.

# Time:  O(nlogn + nlogw), n = len(nums), w = max(nums)-min(nums)
# Space: O(1)

class Solution(object):
    def smallestDistancePair(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        # Sliding window solution
        def possible(guess, nums, k):
            count, left = 0, 0
            for right, num in enumerate(nums):
                while num-nums[left] > guess:
                    left += 1
                count += right-left
            return count >= k

        nums.sort()
        left, right = 0, nums[-1]-nums[0]+1
        while left < right:
            mid = left + (right-left)/2
            if possible(mid, nums, k):
                right = mid
            else:
                left = mid+1
        return left
```

word-abbreviation.py

```
# word-abbreviation is not found.
# Time:  O(n * l) ~ O(n^2 * l^2)
# Space: O(n * l)

import collections


class Solution(object):
    def wordsAbbreviation(self, dict):
        """
        :type dict: List[str]
        :rtype: List[str]
        """
        def isUnique(prefix, words):
            return sum(word.startswith(prefix) for word in words) == 1

        def toAbbr(prefix, word):
            abbr = prefix + str(len(word) - 1 - len(prefix)) + word[-1]
            return abbr if len(abbr) < len(word) else word

        abbr_to_word = collections.defaultdict(set)
        word_to_abbr = {}

        for word in dict:
            prefix = word[:1]
            abbr_to_word[toAbbr(prefix, word)].add(word)

        for abbr, conflicts in abbr_to_word.iteritems():
            if len(conflicts) > 1:
                for word in conflicts:
                    for i in xrange(2, len(word)):
                        prefix = word[:i]
                        if isUnique(prefix, conflicts):
                            word_to_abbr[word] = toAbbr(prefix, word)
                            break
            else:
                word_to_abbr[conflicts.pop()] = abbr

        return [word_to_abbr[word] for word in dict]
```

excel-sheet-column-number.py

```
# DESC
# Example 3:
# Constraints:
# Given a column title as appear in an Excel sheet, return its corresponding colum
# n number.
# For example:
# Example 1:
# Example 2:

# NOTE
# s consists only of uppercase English letters.
# 1 <= s.length <= 7
# s is between "A" and "FXSHRXW".

# EXAMPLE
# Input: "AB"
# Output: 28
# Input: "A"
# Output: 1
# A -> 1
#     B -> 2
#     C -> 3
#     ...
#     Z -> 26
#     AA -> 27
#     AB -> 28
#     ...
# Input: "ZY"
# Output: 701

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def titleToNumber(self, s):
        """
        :type s: str
        :rtype: int
        """
        result = 0
        for i in xrange(len(s)):
            result *= 26
            result += ord(s[i]) - ord('A') + 1
        return result
```

the-maze-ii.py

```
# the-maze-ii is not found.
# Time:  O(max(r, c) * wlogw)
# Space: O(w)

import heapq


class Solution(object):
    def shortestDistance(self, maze, start, destination):
        """
        :type maze: List[List[int]]
        :type start: List[int]
        :type destination: List[int]
        :rtype: int
        """
        start, destination = tuple(start), tuple(destination)

        def neighbors(maze, node):
            for dir in [(-1, 0), (0, 1), (0, -1), (1, 0)]:
                cur_node, dist = list(node), 0
                while 0 <= cur_node[0]+dir[0] < len(maze) and \
                      0 <= cur_node[1]+dir[1] < len(maze[0]) and \
                      not maze[cur_node[0]+dir[0]][cur_node[1]+dir[1]]:
                    cur_node[0] += dir[0]
                    cur_node[1] += dir[1]
                    dist += 1
                yield dist, tuple(cur_node)

        heap = [(0, start)]
        visited = set()
        while heap:
            dist, node = heapq.heappop(heap)
            if node in visited: continue
            if node == destination:
                return dist
            visited.add(node)
            for neighbor_dist, neighbor in neighbors(maze, node):
                heapq.heappush(heap, (dist+neighbor_dist, neighbor))

        return -1
```

majority-element-ii.py

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

# NOTE
#

# EXAMPLE
# Input: [3,2,3]
# Output: [3]
# 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]
```

delete-leaves-with-a-given-value.py

```python
# delete-leaves-with-a-given-value is not found.
# 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 removeLeafNodes(self, root, target):
        """
        :type root: TreeNode
        :type target: int
        :rtype: TreeNode
        """
        if not root:
            return None
        root.left = self.removeLeafNodes(root.left, target)
        root.right = self.removeLeafNodes(root.right, target)
        return None if root.left == root.right and root.val == target else root
```

student-attendance-record-ii.py

```
# DESC
# Given a positive integer n, return the number of all possible attendance records
#  with length n, which will be regarded as rewardable. The answer may be very lar
# ge, return it after mod 109 + 7.
# Example 1:
# Note:
# The value of n won't exceed 100,000.
# A student attendance record is a string that only contains the following three c
# haracters:
# A record is regarded as rewardable if it doesn't contain more than one 'A' (abse
# nt) or more than two continuous 'L' (late).

# NOTE
# 'A' : Absent.
# 'P' : Present.
# 'L' : Late.

# EXAMPLE
# Input: n = 2
# Output: 8
# Explanation:
# There are 8 records with length 2 will be r
# egarded as rewardable:
# "PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"
# Only "AA"
#  won't be regarded as rewardable owing to more than one absent times.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def checkRecord(self, n):
        """
        :type n: int
        :rtype: int
        """
        M = 1000000007
        a0l0, a0l1, a0l2, a1l0, a1l1, a1l2 = 1, 0, 0, 0, 0, 0
        for i in xrange(n+1):
            a0l2, a0l1, a0l0 = a0l1, a0l0, (a0l0 + a0l1 + a0l2) % M
            a1l2, a1l1, a1l0 = a1l1, a1l0, (a0l0 + a1l0 + a1l1 + a1l2) % M
        return a1l0
```

shortest-distance-to-a-character.py

```
# DESC
# Example 1:
# Note:
# Given a string S and a character C, return an array of integers representing the
#  shortest distance from the character C in the string.

# NOTE
# C is a single character, and guaranteed to be in string S.
# All letters in S and C are lowercase.
# S string length is in [1, 10000].

# EXAMPLE
# Input: S = "loveleetcode", C = 'e'
# Output: [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]

# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def shortestToChar(self, S, C):
        """
        :type S: str
        :type C: str
        :rtype: List[int]
        """
        result = [len(S)] * len(S)
        prev = -len(S)
        for i in itertools.chain(xrange(len(S)),
                                 reversed(xrange(len(S)))):
            if S[i] == C:
                prev = i
            result[i] = min(result[i], abs(i-prev))
        return result
```

minimum-index-sum-of-two-lists.py

```python
# DESC
# Note:
# Example 2:
# You need to help them find out their common interest with the least list index s
# um. If there is a choice tie between answers, output all of them with no order r
# equirement. You could assume there always exists an answer.
# Suppose Andy and Doris want to choose a restaurant for dinner, and they both hav
# e a list of favorite restaurants represented by strings.
# Example 1:

# NOTE
# The length of strings in both lists will be in the range of [1, 30].
# No duplicates in both lists.
# The length of both lists will be in the range of [1, 1000].
# The index is starting from 0 to the list length minus 1.

# EXAMPLE
# Input:
# ["Shogun", "Tapioca Express", "Burger King", "KFC"]
# ["KFC", "Shogun", "Bu
# rger King"]
# Output: ["Shogun"]
# Explanation: The restaurant they both like and ha
# ve the least index sum is "Shogun" with index sum 1 (0+1).
# Input:
# ["Shogun", "Tapioca Express", "Burger King", "KFC"]
# ["Piatti", "The Grill
#  at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]
# Output: ["Shogun"]
# Expl
# anation: The only restaurant they both like is "Shogun".

# Time:  O((m + n) * l), m is the size of list1, n is the size of list2
# Space: O(m * l), l is the average length of string

class Solution(object):
    def findRestaurant(self, list1, list2):
        """
        :type list1: List[str]
        :type list2: List[str]
        :rtype: List[str]
        """
        lookup = {}
        for i, s in enumerate(list1):
            lookup[s] = i

        result = []
        min_sum = float("inf")
        for j, s in enumerate(list2):
            if j > min_sum:
                break
            if s in lookup:
                if j + lookup[s] < min_sum:
                    result = [s]
                    min_sum = j + lookup[s]
                elif j + lookup[s] == min_sum:
                    result.append(s)
        return result
```

reordered-power-of-2.py

```
# DESC
# Return true if and only if we can do this in a way such that the resulting numbe
# r is a power of 2.
# Example 1:
# Example 5:
# Starting with a positive integer N, we reorder the digits in any order (includin
# g the original order) such that the leading digit is not zero.
# Note:
# Example 2:
# Example 3:
# Example 4:

# NOTE
# 1 <= N <= 10^9

# EXAMPLE
# Input: 46
# Output: true
# Input: 24
# Output: false
# Input: 10
# Output: false
# Input: 16
# Output: true
# Input: 1
# Output: true

# 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))
```

self-dividing-numbers.py

```
# DESC
# Also, a self-dividing number is not allowed to contain the digit zero.
# Example 1:
# Note:
# 128 % 1 == 0
# Given a lower and upper number bound, output a list of every possible self divid
# ing number, including the bounds if possible.
# A self-dividing number is a number that is divisible by every digit it contains.
# For example, 128 is a self-dividing number because 128 % 1 == 0, 128 % 2 == 0, a
# nd 128 % 8 == 0.

# NOTE
# The boundaries of each input argument are 1 <= left <= right <= 10000.

# EXAMPLE
# Input:
# left = 1, right = 22
# Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]

# Time:  O(nlogr) = O(n)
# Space: O(logr) = O(1)

class Solution(object):
    def selfDividingNumbers(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: List[int]
        """
        def isDividingNumber(num):
            n = num
            while n > 0:
                n, r = divmod(n, 10)
                if r == 0 or (num%r) != 0:
                    return False
            return True

        return [num for num in xrange(left, right+1) if isDividingNumber(num)]


# Time:  O(nlogr) = O(n)
# Space: O(logr) = O(1)
import itertools


class Solution2(object):
    def selfDividingNumbers(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: List[int]
        """
        return [num for num in xrange(left, right+1) \
                if not any(itertools.imap(lambda x: int(x) == 0 or num%int(x) != 0, str(num)))]
```

max-area-of-island.py

```
# DESC
# Example 1:
# Note: The length of each dimension in the given grid does not exceed 50.
# Example 2:
# Find the maximum area of an island in the given 2D array. (If there is no island
# , the maximum area is 0.)
# Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (rep
# resenting land) connected 4-directionally (horizontal or vertical.) You may assu
# me all four edges of the grid are surrounded by water.

# NOTE
#

# EXAMPLE
# [[0,0,0,0,0,0,0,0]]
# [[0,0,1,0,0,0,0,1,0,0,0,0,0],
#  [0,0,0,0,0,0,0,1,1,1,0,0,0],
#  [0,1,1,0,1,0,0,0,0,
# 0,0,0,0],
#  [0,1,0,0,1,1,0,0,1,0,1,0,0],
#  [0,1,0,0,1,1,0,0,1,1,1,0,0],
#  [0,0,0,0,
# 0,0,0,0,0,0,1,0,0],
#  [0,0,0,0,0,0,0,1,1,1,0,0,0],
#  [0,0,0,0,0,0,0,1,1,0,0,0,0]]

# Time:  O(m * n)
# Space: O(m * n), the max depth of dfs may be m * n

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

        def dfs(i, j, grid, area):
            if not (0 <= i < len(grid) and \
                    0 <= j < len(grid[0]) and \
                    grid[i][j] > 0):
                return False
            grid[i][j] *= -1
            area[0] += 1
            for d in directions:
                dfs(i+d[0], j+d[1], grid, area)
            return True

        result = 0
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                area = [0]
                if dfs(i, j, grid, area):
                    result = max(result, area[0])
        return result
```

smallest-string-starting-from-leaf.py

```
# DESC
# Example 3:
# Find the lexicographically smallest string that starts at a leaf of this tree an
# d ends at the root.
# Example 2:
# (As a reminder, any shorter prefix of a string is lexicographically smaller: for
#  example, "ab" is lexicographically smaller than "aba".  A leaf of a node is a n
# ode that has no children.)
# Note:
# Given the root of a binary tree, each node has a value from 0 to 25 representing
#  the letters 'a' to 'z': a value of 0 represents 'a', a value of 1 represents 'b
# ', and so on.
# Example 1:

# 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.

# EXAMPLE
# Input: [25,1,3,1,3,0,2]
# Output: "adz"
# Input: [0,1,2,3,4,3,4]
# Output: "dba"
# Input: [2,2,1,null,1,0,null,0]
# Output: "abc"

# Time:  O(n + l * h), l is the number of leaves
# 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]
```

number-of-distinct-islands-ii.py

```
# number-of-distinct-islands-ii is not found.
# Time:  O((m * n) * log(m * n))
# Space: O(m * n)

class Solution(object):
    def numDistinctIslands2(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        def dfs(i, j, grid, island):
            if not (0 <= i < len(grid) and \
                    0 <= j < len(grid[0]) and \
                    grid[i][j] > 0):
                return False
            grid[i][j] *= -1
            island.append((i, j))
            for d in directions:
                dfs(i+d[0], j+d[1], grid, island)
            return True

        def normalize(island):
            shapes = [[] for _ in xrange(8)]
            for x, y in island:
                rotations_and_reflections = [[ x,  y], [ x, -y], [-x, y], [-x, -y],
                                             [ y,  x], [ y, -x], [-y, x], [-y, -x]]
                for i in xrange(len(rotations_and_reflections)):
                    shapes[i].append(rotations_and_reflections[i])
            for shape in shapes:
                shape.sort()  # Time: O(ilogi), i is the size of the island, the max would be (m * n)
                origin = list(shape[0])
                for p in shape:
                    p[0] -= origin[0]
                    p[1] -= origin[1]
            return min(shapes)

        islands = set()
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                island = []
                if dfs(i, j, grid, island):
                    islands.add(str(normalize(island)))
        return len(islands)
```

profitable-schemes.py

```
# DESC
# Example 2:
# If a gang member participates in one crime, that member can't participate in ano
# ther crime.
# Note:
# The i-th crime generates a profit[i] and requires group[i] gang members to parti
# cipate.
# How many schemes can be chosen?  Since the answer may be very large, return it m
# odulo 10^9 + 7.
# Example 1:
# Let's call a profitable scheme any subset of these crimes that generates at leas
# t P profit, and the total number of gang members participating in that subset of
#  crimes is at most G.
# There are G people in a gang, and a list of various crimes they could commit.

# NOTE
# 1 <= group[i] <= 100
# 1 <= G <= 100
# 1 <= group.length = profit.length <= 100
# 0 <= P <= 100
# 0 <= profit[i] <= 100

# EXAMPLE
# Input: G = 10, P = 5, group = [2,3,5], profit = [6,7,8]
# Output: 7
# Explanation:
#
# To make a profit of at least 5, the gang could commit any crimes, as long as the
# y commit one.
# There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2),
# and (0,1,2).
# Input: G = 5, P = 3, group = [2,2], profit = [2,3]
# Output: 2
# Explanation:
# To ma
# ke a profit of at least 3, the gang could either commit crimes 0 and 1, or just
# crime 1.
# In total, there are 2 schemes.

# Time:  O(n * p * g)
# Space: O(p * g)

import itertools


class Solution(object):
    def profitableSchemes(self, G, P, group, profit):
        """
        :type G: int
        :type P: int
        :type group: List[int]
        :type profit: List[int]
        :rtype: int
        """
        dp = [[0 for _ in xrange(G+1)] for _ in xrange(P+1)]
        dp[0][0] = 1
        for p, g in itertools.izip(profit, group):
            for i in reversed(xrange(P+1)):
```

```
        for j in reversed(xrange(G-g+1)):
            dp[min(i+p, P)][j+g] += dp[i][j]
return sum(dp[P]) % (10**9 + 7)
```

delete-nodes-and-return-forest.py

```
# DESC
# Constraints:
# to_delete
# After deleting all nodes with a value in to_delete, we are left with a forest (a
#  disjoint union of trees).
# Given the root of a binary tree, each node in the tree has a distinct value.
# Return the roots of the trees in the remaining forest.  You may return the resul
# t in any order.
# Example 1:

# NOTE
# The number of nodes in the given tree is at most 1000.
# to_delete contains distinct values between 1 and 1000.
# to_delete.length <= 1000
# Each node has a distinct value between 1 and 1000.

# EXAMPLE
# Input: root = [1,2,3,4,5,6,7], to_delete = [3,5]
# Output: [[1,2,null,4],[6],[7]]

# 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
```

fixed-point.py

```
# fixed-point is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def fixedPoint(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        left, right = 0, len(A)-1
        while left <= right:
            mid = left + (right-left)//2
            if A[mid] >= mid:
                right = mid-1
            else:
                left = mid+1
        return left if A[left] == left else -1
```

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

```
# DESC
# Given a matrix consisting of 0s and 1s, we may choose any number of columns in t
# he matrix and flip every cell in that column.  Flipping a cell changes the value
#  of that cell from 0 to 1 or from 1 to 0.
# Return the maximum number of rows that have all values equal after some number o
# f flips.
# Example 1:
# Example 3:
# Example 2:
# Note:

# NOTE
# All matrix[i].length's are equal
# 1 <= matrix[i].length <= 300
# 1 <= matrix.length <= 300
# matrix[i][j] is 0 or 1

# EXAMPLE
# Input: [[0,1],[1,0]]
# Output: 2
# Explanation: After flipping values in the first c
# olumn, both rows have equal values.
# Input: [[0,1],[1,1]]
# Output: 1
# Explanation: After flipping no values, 1 row has
# all values equal.
# 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.

# 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())
```

distinct-echo-substrings.py

```
# DESC
# Return the number of distinct non-empty substrings of text that can be written a
# s the concatenation of some string with itself (i.e. it can be written as a + a
# where a is some string).
# Constraints:
# Example 1:
# Example 2:

# NOTE
# text has only lowercase English letters.
# 1 <= text.length <= 2000

# EXAMPLE
# Input: text = "abcabcabc"
# Output: 3
# Explanation: The 3 substrings are "abcabc",
# "bcabca" and "cabcab".
# Input: text = "leetcodeleetcode"
# Output: 2
# Explanation: The 2 substrings are "ee
# " and "leetcodeleetcode".

# Time:  O(n^2 + d), d is the duplicated of result substrings size
# Space: O(r), r is the size of result substrings set

class Solution(object):
    def distinctEchoSubstrings(self, text):
        """
        :type text: str
        :rtype: int
        """
        def KMP(text, l, result):
            prefix = [-1]*(len(text)-l)
            j = -1
            for i in xrange(1, len(prefix)):
                while j > -1 and text[l+j+1] != text[l+i]:
                    j = prefix[j]
                if text[l+j+1] == text[l+i]:
                    j += 1
                prefix[i] = j
                if (j+1) and (i+1) % ((i+1) - (j+1)) == 0 and \
                   (i+1) // ((i+1) - (j+1)) % 2 == 0:
                    result.add(text[l:l+i+1])
            return len(prefix)-(prefix[-1]+1) \
                   if prefix[-1]+1 and len(prefix) % (len(prefix)-(prefix[-1]+1)) == 0 \
                   else float("inf")

        result = set()
        i, l = 0, len(text)-1
        while i < l:  # aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcdefabcdefabcdef
            l = min(l, i + KMP(text, i, result));
            i += 1
        return len(result)


# Time:  O(n^2 + d), d is the duplicated of result substrings size
# Space: O(r), r is the size of result substrings set
class Solution2(object):
```

```
    def distinctEchoSubstrings(self, text):
        """
        :type text: str
        :rtype: int
        """
        result = set()
        for l in xrange(1, len(text)//2+1):
            count = sum(text[i] == text[i+l] for i in xrange(l))
            for i in xrange(len(text)-2*l):
                if count == l:
                    result.add(text[i:i+l])
                count += (text[i+l] == text[i+l+l]) - (text[i] == text[i+l])
            if count == l:
                result.add(text[len(text)-2*l:len(text)-2*l+l])
        return len(result)


# Time:  O(n^2 + d), d is the duplicated of result substrings size
# Space: O(r), r is the size of result substrings set
class Solution3(object):
    def distinctEchoSubstrings(self, text):
        """
        :type text: str
        :rtype: int
        """
        MOD = 10**9+7
        D = 27  # a-z and ''
        result = set()
        for i in xrange(len(text)-1):
            left, right, pow_D = 0, 0, 1
            for l in xrange(1, min(i+2, len(text)-i)):
                left = (D*left + (ord(text[i-l+1])-ord('a')+1)) % MOD
                right = (pow_D*(ord(text[i+l])-ord('a')+1) + right) % MOD
                if left == right:  # assumed no collision
                    result.add(left)
                pow_D = (pow_D*D) % MOD
        return len(result)


# Time:  O(n^3 + d), d is the duplicated of result substrings size
# Space: O(r), r is the size of result substrings set
class Solution_TLE(object):
    def distinctEchoSubstrings(self, text):
        """
        :type text: str
        :rtype: int
        """
        def compare(text, l, s1, s2):
            for i in xrange(l):
                if text[s1+i] != text[s2+i]:
                    return False
            return True

        MOD = 10**9+7
        D = 27  # a-z and ''
        result = set()
        for i in xrange(len(text)):
            left, right, pow_D = 0, 0, 1
            for l in xrange(1, min(i+2, len(text)-i)):
                left = (D*left + (ord(text[i-l+1])-ord('a')+1)) % MOD
```

```
        right = (pow_D*(ord(text[i+l])-ord('a')+1) + right) % MOD
        if left == right and compare(text, l, i-l+1, i+1):
            result.add(text[i+1:i+1+l])
        pow_D = (pow_D*D) % MOD
return len(result)
```

binary-watch.py

```
# DESC
# Example:
# Note:
# A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6
#  LEDs on the bottom represent the minutes (0-59).
# For example, the above binary watch reads "3:25".
# Given a non-negative integer n which represents the number of LEDs that are curr
# ently on, return all possible times the watch could represent.
# Each LED represents a zero or one, with the least significant bit on the right.

# NOTE
# The hour must not contain a leading zero, for example "01:00" is not valid, it s
# hould be "1:00".
# The order of output does not matter.
# The minute must be consist of two digits and may contain a leading zero, for exa
# mple "10:2" is not valid, it should be "10:02".

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

# Time:  O(1)
# Space: O(1)


class Solution(object):
    def readBinaryWatch(self, num):
        """
        :type num: int
        :rtype: List[str]
        """
        def bit_count(bits):
            count = 0
            while bits:
                bits &= bits-1
                count += 1
            return count

        return ['%d:%02d' % (h, m) for h in xrange(12) for m in xrange(60)
                if bit_count(h) + bit_count(m) == num]

    def readBinaryWatch2(self, num):
        """
        :type num: int
        :rtype: List[str]
        """
        return ['{0}:{1}'.format(str(h), str(m).zfill(2))
                for h in range(12) for m in range(60)
                if (bin(h) + bin(m)).count('1') == num]
```

split-array-into-fibonacci-sequence.py

```
# DESC
# Also, note that when splitting the string into pieces, each piece must not have
# extra leading zeroes, except if the piece is the number 0 itself.
# Return any Fibonacci-like sequence split from S, or return [] if it cannot be done.
# Example 5:
# Note:
# Example 1:
# Example 2:
# Example 3:
# Given a string S of digits, such as S = "123456579", we can split it into a Fibo
# nacci-like sequence [123, 456, 579].
# Example 4:
# Formally, a Fibonacci-like sequence is a list F of non-negative integers such that:

# NOTE
# S contains only digits.
# 1 <= S.length <= 200
# and F[i] + F[i+1] = F[i+2] for all 0 <= i < F.length - 2.
# 0 <= F[i] <= 2^31 - 1, (that is, each integer fits a 32-bit signed integer type);
# F.length >= 3;

# EXAMPLE
# Input: "11235813"
# Output: [1,1,2,3,5,8,13]
# Input: "123456579"
# Output: [123,456,579]
# Input: "1101111"
# Output: [110, 1, 111]
# Explanation: The output [11, 0, 11, 11] w
# ould also be accepted.
# Input: "112358130"
# Output: []
# Explanation: The task is impossible.
# Input: "0123"
# Output: []
# Explanation: Leading zeroes are not allowed, so "01", "
# 2", "3" is not valid.

# Time:  O(n^3)
# Space: O(n)


class Solution(object):
    def splitIntoFibonacci(self, S):
        """
        :type S: str
        :rtype: List[int]
        """
        def startswith(S, k, x):
            y = 0
            for i in xrange(k, len(S)):
                y = 10*y + int(S[i])
                if y == x:
                    return i-k+1
                elif y > x:
                    break
            return 0
```

```
MAX_INT = 2**31-1
a = 0
for i in xrange(len(S)-2):
    a = 10*a + int(S[i])
    b = 0
    for j in xrange(i+1, len(S)-1):
        b = 10*b + int(S[j])
        fib = [a, b]
        k = j+1
        while k < len(S):
            if fib[-2] > MAX_INT-fib[-1]:
                break
            c = fib[-2]+fib[-1]
            length = startswith(S, k, c)
            if length == 0:
                break
            fib.append(c)
            k += length
        else:
            return fib
        if b == 0:
            break
    if a == 0:
        break
return []
```

minimum-factorization.py

```
# minimum-factorization is not found.
# Time:  O(loga)
# Space: O(1)

class Solution(object):
    def smallestFactorization(self, a):
        """
        :type a: int
        :rtype: int
        """
        if a < 2:
            return a
        result, mul = 0, 1
        for i in reversed(xrange(2, 10)):
            while a % i == 0:
                a /= i
                result = mul*i + result
                mul *= 10
        return  result if a == 1 and result < 2**31 else 0
```

advantage-shuffle.py

```
# DESC
# Example 2:
# Return any permutation of A that maximizes its advantage with respect to B.
# Example 1:
# Given two arrays A and B of equal size, the advantage of A with respect to B is
# the number of indices i for which A[i] > B[i].
# Note:

# NOTE
# 0 <= A[i] <= 10^9
# 1 <= A.length = B.length <= 10000
# 0 <= B[i] <= 10^9

# EXAMPLE
# Input: A = [12,24,8,32], B = [13,25,32,11]
# Output: [24,32,8,12]
# Input: A = [2,7,11,15], B = [1,10,4,11]
# Output: [2,11,7,15]

# 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]
```

intersection-of-two-linked-lists.py

```
# DESC
# Example 2:
# Write a program to find the node at which the intersection of two singly linked
# lists begins.
# For example, the following two linked lists:
# Example 1:
# begin to intersect at node c1.
# Notes:
# Example 3:

# NOTE
# You may assume there are no cycles anywhere in the entire linked structure.
# Each value on each linked list is in the range [1, 10^9].
# Your code should preferably run in O(n) time and use only O(1) memory.
# The linked lists must retain their original structure after the function returns.
# If the two linked lists have no intersection at all, return null.

# EXAMPLE
# Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2,
# skipB = 3
# Output: Reference of the node with value = 8
# Input Explanation: The in
# tersected node's value is 8 (note that this must not be 0 if the two lists inter
# sect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads
#  as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are
#  3 nodes before the intersected node in B.
# Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB
# = 1
# Output: Reference of the node with value = 2
# Input Explanation: The intersec
# ted node's value is 2 (note that this must not be 0 if the two lists intersect).
#  From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3
# ,2,4]. There are 3 nodes before the intersected node in A; There are 1 node befo
# re the intersected node in B.
# Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
# Ou
# tput: null
# Input Explanation: From the head of A, it reads as [2,6,4]. From the
# head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal
#  must be 0, while skipA and skipB can be arbitrary values.
# Explanation: The two
# lists do not intersect, so return null.

# Time:  O(m + n)
# Space: O(1)

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

class Solution(object):
    # @param two ListNodes
    # @return the intersected ListNode
    def getIntersectionNode(self, headA, headB):
        curA, curB = headA, headB
        begin, tailA, tailB = None, None, None
```

```
        # a->c->b->c
        # b->c->a->c
        while curA and curB:
            if curA == curB:
                begin = curA
                break

            if curA.next:
                curA = curA.next
            elif tailA is None:
                tailA = curA
                curA = headB
            else:
                break

            if curB.next:
                curB = curB.next
            elif tailB is None:
                tailB = curB
                curB = headA
            else:
                break

        return begin
```

string-without-aaa-or-bbb.py

```
# DESC
# Example 1:
# Note:
# Example 2:
# Given two integers A and B, return any string S such that:

# NOTE
# S has length A + B and contains exactly A 'a' letters, and exactly B 'b' letters;
# The substring 'aaa' does not occur in S;
# It is guaranteed such an S exists for the given A and B.
# The substring 'bbb' does not occur in S.
# 0 <= B <= 100
# 0 <= A <= 100

# EXAMPLE
# Input: A = 4, B = 1
# Output: "aabaa"
# Input: A = 1, B = 2
# Output: "abb"
# Explanation: "abb", "bab" and "bba" are all co
# rrect answers.

# 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)
```

minimum-falling-path-sum-ii.py

```
# DESC
# Example 1:
# Constraints:
# Return the minimum sum of a falling path with non-zero shifts.
# Given a square grid of integers arr, a falling path with non-zero shifts is a ch
# oice of exactly one element from each row of arr, such that no two elements chos
# en in adjacent rows are in the same column.

# NOTE
# -99 <= arr[i][j] <= 99
# 1 <= arr.length == arr[i].length <= 200

# EXAMPLE
# Input: arr = [[1,2,3],[4,5,6],[7,8,9]]
# Output: 13
# Explanation:
# The possible fal
# ling paths are:
# [1,5,9], [1,5,7], [1,6,7], [1,6,8],
# [2,4,8], [2,4,9], [2,6,7], [
# 2,6,8],
# [3,4,8], [3,4,9], [3,5,7], [3,5,9]
# The falling path with the smallest su
# m is [1,5,7], so the answer is 13.

# Time:  O(m * n)
# Space: O(1)

import heapq


class Solution(object):
    def minFallingPathSum(self, arr):
        """
        :type arr: List[List[int]]
        :rtype: int
        """
        for i in xrange(1, len(arr)):
            smallest_two = heapq.nsmallest(2, arr[i-1])
            for j in xrange(len(arr[0])):
                arr[i][j] += smallest_two[1] if arr[i-1][j] == smallest_two[0] else smallest_two[0]
        return min(arr[-1])
```

minimum-number-of-k-consecutive-bit-flips.py

```
# DESC
# Example 3:
# Return the minimum number of K-bit flips required so that there is no 0 in the a
# rray.  If it is not possible, return -1.
# In an array A containing only 0s and 1s, a K-bit flip consists of choosing a (co
# ntiguous) subarray of length K and simultaneously changing every 0 in the subarr
# ay to 1, and every 1 in the subarray to 0.
# Example 1:
# Note:
# Example 2:

# NOTE
# 1 <= A.length <= 30000
# 1 <= K <= A.length

# EXAMPLE
# Input: A = [0,1,0], K = 1
# Output: 2
# Explanation: Flip A[0], then flip A[2].
# Input: A = [0,0,0,1,0,1,1,0], K = 3
# Output: 3
# Explanation:
# Flip A[0],A[1],A[2]:
# A becomes [1,1,1,1,0,1,1,0]
# Flip A[4],A[5],A[6]: A becomes [1,1,1,1,1,0,0,0]
# Fli
# p A[5],A[6],A[7]: A becomes [1,1,1,1,1,1,1,1]
# Input: A = [1,1,0], K = 2
# Output: -1
# Explanation: No matter how we flip subarray
# s of size 2, we can't make the array become [1,1,1].

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minKBitFlips(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        result, curr = 0, 0
        for i in xrange(len(A)):
            if i >= K:
                curr -= A[i-K]//2
            if curr & 1 ^ A[i] == 0:
                if i+K > len(A):
                    return -1
                A[i] += 2
                curr, result = curr+1, result+1
        return result
```

find-mode-in-binary-search-tree.py

```
# DESC
# For example:
#
# Given BST [1,null,2,2],
# return [2].
# Assume a BST is defined as follows:
# Note: If a tree has more than one mode, you can return them in any order.
# Follow up: Could you do that without using any extra space? (Assume that the imp
# licit stack space incurred due to recursion does not count).
# Given a binary search tree (BST) with duplicates, find all the mode(s) (the most
#  frequently occurred element) in the given BST.

# NOTE
# The right subtree of a node contains only nodes with keys greater than or equal
# to the node's key.
# Both the left and right subtrees must also be binary search trees.
# The left subtree of a node contains only nodes with keys less than or equal to t
# he node's key.

# EXAMPLE
# 1
#     \
#      2
#     /
#    2

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMode(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        def inorder(root, prev, cnt, max_cnt, result):
            if not root:
                return prev, cnt, max_cnt

            prev, cnt, max_cnt = inorder(root.left, prev, cnt, max_cnt, result)
            if prev:
                if root.val == prev.val:
                    cnt += 1
                else:
                    cnt = 1
            if cnt > max_cnt:
                max_cnt = cnt
                del result[:]
                result.append(root.val)
            elif cnt == max_cnt:
                result.append(root.val)
            return inorder(root.right, root, cnt, max_cnt, result)

        if not root:
            return []
        result = []
        inorder(root, None, 1, 0, result)
        return result
```

ransom-note.py

```
# DESC
# Each letter in the magazine string can only be used once in your ransom note.
# Example 3:
# Constraints:
# Example 1:
# Example 2:
# Given an arbitrary ransom note string and another string containing letters from
#  all the magazines, write a function that will return true if the ransom note ca
# n be constructed from the magazines ; otherwise, it will return false.

# NOTE
# You may assume that both strings contain only lowercase letters.

# EXAMPLE
# Input: ransomNote = "aa", magazine = "ab"
# Output: false
# Input: ransomNote = "aa", magazine = "aab"
# Output: true
# Input: ransomNote = "a", magazine = "b"
# Output: false

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canConstruct(self, ransomNote, magazine):
        """
        :type ransomNote: str
        :type magazine: str
        :rtype: bool
        """
        counts = [0] * 26
        letters = 0

        for c in ransomNote:
            if counts[ord(c) - ord('a')] == 0:
                letters += 1
            counts[ord(c) - ord('a')] += 1

        for c in magazine:
            counts[ord(c) - ord('a')] -= 1
            if counts[ord(c) - ord('a')] == 0:
                letters -= 1
                if letters == 0:
                    break

        return letters == 0

# Time:  O(n)
# Space: O(1)
import collections

class Solution2(object):
    def canConstruct(self, ransomNote, magazine):
        """
        :type ransomNote: str
        :type magazine: str
        :rtype: bool
```

```
    """
    return not collections.Counter(ransomNote) - collections.Counter(magazine)
```

shortest-path-to-get-all-keys.py

```
# DESC
# Return the lowest number of moves to acquire all keys.  If it's impossible, retu
# rn -1.
# For some 1 <= K <= 6, there is exactly one lowercase and one uppercase letter of
#  the first K letters of the English alphabet in the grid.  This means that there
#  is exactly one key for each lock, and one lock for each key; and also that the
# letters used to represent the keys and locks were chosen in the same order as th
# e English alphabet.
# Note:
# We are given a 2-dimensional grid. "." is an empty cell, "#" is a wall, "@" is t
# he starting point, ("a", "b", ...) are keys, and ("A", "B", ...) are locks.
# Example 1:
# We start at the starting point, and one move consists of walking one space in on
# e of the 4 cardinal directions.  We cannot walk outside the grid, or walk into a
#  wall.  If we walk over a key, we pick it up.  We can't walk over a lock unless
# we have the corresponding key.
# Example 2:

# NOTE
# 1 <= grid[0].length <= 30
# The number of keys is in [1, 6].  Each key has a different letter and opens exac
# tly one lock.
# grid[i][j] contains only '.', '#', '@', 'a'-'f' and 'A'-'F'
# 1 <= grid.length <= 30

# EXAMPLE
# Input: ["@..aA","..B#.","....b"]
# Output: 6
# Input: ["@.a.#","###.#","b.A.B"]
# Output: 8

# Time:  O(k*r*c + |E|log|V|) = O(k*r*c + (k*|V|)*log|V|)
#                             = O(k*r*c + (k*(k*2^k))*log(k*2^k))
#                             = O(k*r*c + (k*(k*2^k))*(logk + k*log2))
#                             = O(k*r*c + (k*(k*2^k))*k)
#                             = O(k*r*c + k^3*2^k)
# Space: O(|V|) = O(k*2^k)

import collections
import heapq


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

        def bfs(grid, source, locations):
            r, c = locations[source]
            lookup = [[False]*(len(grid[0])) for _ in xrange(len(grid))]
            lookup[r][c] = True
            q = collections.deque([(r, c, 0)])
            dist = {}
            while q:
                r, c, d = q.popleft()
```

```
        if source != grid[r][c] != '.':
            dist[grid[r][c]] = d
            continue
        for direction in directions:
            cr, cc = r+direction[0], c+direction[1]
            if not ((0 <= cr < len(grid)) and
                    (0 <= cc < len(grid[cr]))):
                continue
            if grid[cr][cc] != '#' and not lookup[cr][cc]:
                lookup[cr][cc] = True
                q.append((cr, cc, d+1))
    return dist

locations = {place: (r, c)
             for r, row in enumerate(grid)
             for c, place in enumerate(row)
             if place not in '.#'}
dists = {place: bfs(grid, place, locations) for place in locations}

# Dijkstra's algorithm
min_heap = [(0, '@', 0)]
best = collections.defaultdict(lambda: collections.defaultdict(
                                        lambda: float("inf")))
best['@'][0] = 0
target_state = 2**sum(place.islower() for place in locations)-1
while min_heap:
    cur_d, place, state = heapq.heappop(min_heap)
    if best[place][state] < cur_d:
        continue
    if state == target_state:
        return cur_d
    for dest, d in dists[place].iteritems():
        next_state = state
        if dest.islower():
            next_state |= (1 << (ord(dest)-ord('a')))
        elif dest.isupper():
            if not (state & (1 << (ord(dest)-ord('A')))):
                continue
        if cur_d+d < best[dest][next_state]:
            best[dest][next_state] = cur_d+d
            heapq.heappush(min_heap, (cur_d+d, dest, next_state))
return -1
```

shortest-path-visiting-all-nodes.py

```
# DESC
# Note:
# An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given a
# s graph.
# graph.length = N
# Example 2:
# Return the length of the shortest path that visits every node. You may start and
#  stop at any node, you may revisit nodes multiple times, and you may reuse edges
# .
# graph.length = N, and j != i is in the list graph[i] exactly once, if and only i
# f nodes i and j are connected.
# Example 1:

# NOTE
# 0 <= graph[i].length < graph.length
# 1 <= graph.length <= 12

# EXAMPLE
# Input: [[1],[0,2,4],[1,3,4],[2],[1,2]]
# Output: 4
# Explanation: One possible path
# is [0,1,4,2,3]
# Input: [[1,2,3],[0],[0],[0]]
# Output: 4
# Explanation: One possible path is [1,0,2,0,3]

# Time:  O(n * 2^n)
# Space: O(n * 2^n)

import collections


class Solution(object):
    def shortestPathLength(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: int
        """
        dp = [[float("inf")]*(len(graph))
              for _ in xrange(1 << len(graph))]
        q = collections.deque()
        for i in xrange(len(graph)):
            dp[1 << i][i] = 0
            q.append((1 << i, i))
        while q:
            state, node = q.popleft()
            steps = dp[state][node]
            for nei in graph[node]:
                new_state = state | (1 << nei)
                if dp[new_state][nei] == float("inf"):
                    dp[new_state][nei] = steps+1
                    q.append((new_state, nei))
        return min(dp[-1])
```

couples-holding-hands.py

```
# DESC
# The couples' initial seating is given by row[i] being the value of the person wh
# o is initially sitting in the i-th seat.
# The people and seats are represented by an integer from 0 to 2N-1, the couples a
# re numbered in order, the first couple being (0, 1), the second couple being (2,
#  3), and so on with the last couple being (2N-2, 2N-1).
# Example 2:
# Note:
# Example 1:
# N couples sit in 2N seats arranged in a row and want to hold hands.  We want to
# know the minimum number of swaps so that every couple is sitting side by side.
# A swap consists of choosing any two people, then they stand up and switch seats.

# NOTE
# row is guaranteed to be a permutation of 0...len(row)-1.
# len(row) is even and in the range of [4, 60].

# EXAMPLE
# Input: row = [3, 2, 0, 1]
# Output: 0
# Explanation: All couples are already seated
# side by side.
# Input: row = [0, 2, 1, 3]
# Output: 1
# Explanation: We only need to swap the second
#  (row[1]) and third (row[2]) person.

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

class Solution(object):
    def minSwapsCouples(self, row):
        """
        :type row: List[int]
        :rtype: int
        """
        N = len(row)//2
        couples = [[] for _ in xrange(N)]
        for seat, num in enumerate(row):
            couples[num//2].append(seat//2)
        adj = [[] for _ in xrange(N)]
        for couch1, couch2 in couples:
            adj[couch1].append(couch2)
            adj[couch2].append(couch1)

        result = 0
        for couch in xrange(N):
            if not adj[couch]: continue
            couch1, couch2 = couch, adj[couch].pop()
            while couch2 != couch:
                result += 1
                adj[couch2].remove(couch1)
                couch1, couch2 = couch2, adj[couch2].pop()
        return result  # also equals to N - (# of cycles)
```

xor-queries-of-a-subarray.py

```python
# xor-queries-of-a-subarra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def xorQueries(self, arr, queries):
        """
        :type arr: List[int]
        :type queries: List[List[int]]
        :rtype: List[int]
        """
        for i in xrange(1, len(arr)):
            arr[i] ^= arr[i-1]
        return [arr[right] ^ arr[left-1] if left else arr[right] for left, right in queries]
```

meeting-rooms.py

```
# meeting-rooms is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def canAttendMeetings(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: bool
        """
        intervals.sort(key=lambda x: x[0])

        for i in xrange(1, len(intervals)):
            if intervals[i][0] < intervals[i-1][1]:
                return False
        return True
```

split-array-with-same-average.py

```
# DESC
# Note:
# Return true if and only if after such a move, it is possible that the average va
# lue of B is equal to the average value of C, and B and C are both non-empty.
# In a given integer array A, we must move every element of A to either list B or
# list C. (B and C initially start empty.)

# NOTE
# The length of A will be in the range [1, 30].
# A[i] will be in the range of [0, 10000].

# EXAMPLE
# Example :
# Input:
# [1,2,3,4,5,6,7,8]
# Output: true
# Explanation: We can split the a
# rray into [1,4,5,8] and [2,3,6,7], and both of them have the average of 4.5.

# Time:  O(n^4)
# Space: O(n^3)

class Solution(object):
    def splitArraySameAverage(self, A):
        """
        :type A: List[int]
        :rtype: bool
        """
        def possible(total, n):
            for i in xrange(1, n//2+1):
                if total*i%n == 0:
                    return True
            return False
        n, s = len(A), sum(A)
        if not possible(n, s):
            return False

        sums = [set() for _ in xrange(n//2+1)]
        sums[0].add(0)
        for num in A:  # O(n) times
            for i in reversed(xrange(1, n//2+1)):  # O(n) times
                for prev in sums[i-1]:  # O(1) + O(2) + ... O(n/2) = O(n^2) times
                    sums[i].add(prev+num)
        for i in xrange(1, n//2+1):
            if s*i%n == 0 and s*i//n in sums[i]:
                return True
        return False
```

strobogrammatic-number-iii.py

```
# strobogrammatic-number-iii is not found.
# Time:  O(5^(n/2))
# Space: O(n)

class Solution(object):
    lookup = {'0':'0', '1':'1', '6':'9', '8':'8', '9':'6'}
    cache = {}

    # @param {string} low
    # @param {string} high
    # @return {integer}
    def strobogrammaticInRange(self, low, high):
        count = self.countStrobogrammaticUntil(high, False) - \
                self.countStrobogrammaticUntil(low, False) + \
                self.isStrobogrammatic(low)
        return count if count >= 0 else 0

    def countStrobogrammaticUntil(self, num,  can_start_with_0):
        if can_start_with_0 and num in self.cache:
            return self.cache[num]

        count = 0
        if len(num) == 1:
            for c in ['0', '1', '8']:
                if num[0] >= c:
                    count += 1
            self.cache[num] = count
            return count

        for key, val in self.lookup.iteritems():
            if can_start_with_0 or key != '0':
                if num[0] > key:
                    if len(num) == 2:  # num is like "21"
                        count += 1
                    else:  # num is like "201"
                        count += self.countStrobogrammaticUntil('9' * (len(num) - 2), True)
                elif num[0] == key:
                    if len(num) == 2:  # num is like 12".
                        if num[-1] >= val:
                            count += 1
                    else:
                        if num[-1] >= val:  # num is like "102".
                            count += self.countStrobogrammaticUntil(self.getMid(num), True)
                        elif (self.getMid(num) != '0' * (len(num) - 2)):  # num is like "110".
                            count += self.countStrobogrammaticUntil(self.getMid(num), True) - \
                                     self.isStrobogrammatic(self.getMid(num))

        if not can_start_with_0: # Sum up each length.
            for i in xrange(len(num) - 1, 0, -1):
                count += self.countStrobogrammaticByLength(i)
        else:
            self.cache[num] = count

        return count

    def getMid(self, num):
        return num[1:len(num) - 1]
```

```
def countStrobogrammaticByLength(self, n):
    if n == 1:
        return 3
    elif n == 2:
        return 4
    elif n == 3:
        return 4 * 3
    else:
        return 5 * self.countStrobogrammaticByLength(n - 2)

def isStrobogrammatic(self, num):
    n = len(num)
    for i in xrange((n+1) / 2):
        if num[n-1-i] not in self.lookup or \
           num[i] != self.lookup[num[n-1-i]]:
            return False
    return True
```

number-of-substrings-with-only-1s.py

```python
# number-of-substrings-with-only-1s is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numSub(self, s):
        """
        :type s: str
        :rtype: int
        """
        MOD = 10**9+7
        result, count = 0, 0
        for c in s:
            count = count+1 if c == '1' else 0
            result = (result+count)%MOD
        return result
```

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

```
# DESC
# Example 1:
# Initially, all next pointers are set to NULL.
# 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:
# Follow up:
# Constraints:
# 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.

# NOTE
# The number of nodes in the given tree is less than 4096.
# Recursive approach is fine, you may assume implicit stack space does not count a
# s extra space for this problem.
# You may only use constant extra space.
# -1000 <= node.val <= 1000

# EXAMPLE
# struct Node {
#   int val;
#   Node *left;
#   Node *right;
#   Node *next;
# }
# Input: root = [1,2,3,4,5,6,7]
# Output: [1,#,2,3,#,4,5,6,7,#]
# Explanation: Given t
# he 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.

# 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:
            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)
```

smallest-sufficient-team.py

```
# DESC
# Example 1:
# You may return the answer in any order.  It is guaranteed an answer exists.
# Constraints:
# Return any sufficient team of the smallest possible size, represented by the ind
# ex of each person.
# req_skills
# Consider a sufficient team: a set of people such that for every required skill i
# n req_skills, there is at least one person in the team who has that skill.  We c
# an represent these teams by the index of each person: for example, team = [0, 1,
#  3] represents the people with skills people[0], people[1], and people[3].
# In a project, you have a list of required skills req_skills, and a list of peopl
# e.  The i-th person people[i] contains a list of skills that person has.
# Example 2:

# NOTE
# 1 <= people[i].length, req_skills[i].length, people[i][j].length <= 16
# 1 <= req_skills.length <= 16
# req_skills[i][j], people[i][j][k] are lowercase English letters.
# It is guaranteed a sufficient team exists.
# Elements of req_skills and people[i] are (respectively) distinct.
# 1 <= people.length <= 60
# Every skill in people[i] is a skill in req_skills.

# EXAMPLE
# Input: req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],[
# "nodejs","reactjs"]]
# Output: [0,2]
# Input: req_skills = ["algorithms","math","java","reactjs","csharp","aws"], peopl
# e = [["algorithms","math","java"],["algorithms","math","reactjs"],["java","cshar
# p","aws"],["reactjs","csharp"],["csharp","math"],["aws","java"]]
# Output: [1,2]

# Time:  O(m * 2^n), n is the number of skills
#                    m is the number of people
# Space: O(2^n)

class Solution(object):
    def smallestSufficientTeam(self, req_skills, people):
        """
        :type req_skills: List[str]
        :type people: List[List[str]]
        :rtype: List[int]
        """
        lookup = {v: i for i, v in enumerate(req_skills)}
        dp = {0: []}
        for i, p in enumerate(people):
            his_skill_set = 0
            for skill in p:
                if skill in lookup:
                    his_skill_set |= 1 << lookup[skill]
            for skill_set, people in dp.items():
                with_him = skill_set | his_skill_set
                if with_him == skill_set: continue
                if with_him not in dp or \
                   len(dp[with_him]) > len(people)+1:
                    dp[with_him] = people + [i]
        return dp[(1<<len(req_skills))-1]
```

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

```
# DESC
# Given a binary tree, return the sum of values of nodes with even-valued grandpar
# ent.  (A grandparent of a node is the parent of its parent, if it exists.)
# Constraints:
# Example 1:
# If there are no nodes with an even-valued grandparent, return 0.

# NOTE
# The value of nodes is between 1 and 100.
# The number of nodes in the tree is between 1 and 10^4.

# EXAMPLE
# 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 a
# re the even-value grandparents.

# 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

```
# DESC
# Example 1:
# ZeroEvenOdd
# Each of the threads is given a printNumber method to output an integer. Modify t
# he given program to output the series 010203040506... where the length of the se
# ries must be 2n.
# Example 2:
# Suppose you are given the following code:
# The same instance of ZeroEvenOdd will be passed to three different threads:

# NOTE
# Thread A will call zero() which should only output 0's.
# Thread C will call odd() which should only output odd numbers.
# Thread B will call even() which should only ouput even numbers.

# EXAMPLE
# Input: n = 2
# Output: "0102"
# Explanation: There are three threads being fired asy
# nchronously. One of them calls zero(), the other calls even(), and the last one
# calls odd(). "0102" is the correct output.
# class ZeroEvenOdd {
#   public ZeroEvenOdd(int n) { ... }      // constructor
#   pu
# blic void zero(printNumber) { ... }  // only output 0's
#   public void even(print
# Number) { ... }  // only output even numbers
#   public void odd(printNumber) { ..
# . }   // only output odd numbers
# }
# 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()
                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()
```

number-of-ways-to-stay-in-the-same-place-after-some-steps.py

```python
# number-of-ways-to-stay-in-the-same-place-after-some-steps is not found.
# Time:  O(n^2), n is the number of steps
# Space: O(n)

class Solution(object):
    def numWays(self, steps, arrLen):
        """
        :type steps: int
        :type arrLen: int
        :rtype: int
        """
        MOD = int(1e9+7)
        l = min(1+steps//2, arrLen)
        dp = [0]*(l+2)
        dp[1] = 1
        while steps > 0:
            steps -= 1
            new_dp = [0]*(l+2)
            for i in xrange(1, l+1):
                new_dp[i] = (dp[i] + dp[i-1] + dp[i+1]) % MOD
            dp = new_dp
        return dp[1]
```

range-sum-query-immutable.py

```
# range-sum-query-immutable is not found.
# Time:  ctor:   O(n),
#        lookup: O(1)
# Space: O(n)

class NumArray(object):
    def __init__(self, nums):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        self.accu = [0]
        for num in nums:
            self.accu.append(self.accu[-1] + num),

    def sumRange(self, i, j):
        """
        sum of elements nums[i..j], inclusive.
        :type i: int
        :type j: int
        :rtype: int
        """
        return self.accu[j + 1] - self.accu[i]
```

count-primes.py

```
# DESC
# We start off with a table of n numbers. Let's look at the first number, 2. We kn
# ow all multiples of 2 must not be primes, so we mark them off as non-primes. The
# n we look at the next number, 3. Similarly, all multiples of 3 such as 3 × 2 = 6
# , 3 × 3 = 9, ... must not be primes, so we mark them off as well. Now we look at
#  the next number, 4, which was already marked off. What does this tell you? Shou
# ld you mark off all multiples of 4 as well?
# The Sieve of Eratosthenes uses an extra O(n) memory and its runtime complexity i
# s O(n log log n). For the more mathematically inclined readers, you can read mor
# e about its algorithm complexity on Wikipedia.
# In fact, we can mark off multiples of 5 starting at 5 × 5 = 25, because 5 × 2 =
# 10 was already marked off by multiple of 2, similarly 5 × 3 = 15 was already mar
# ked off by multiple of 3. Therefore, if the current number is p, we can always m
# ark off multiples of p starting at p2, then in increments of p: p2 + p, p2 + 2p,
#  ... Now what should be the terminating loop condition?
# The Sieve of Eratosthenes is one of the most efficient ways to find all prime nu
# mbers up to n. But don't let that name scare you, I promise that the concept is
# surprisingly simple.
# Let's write down all of 12's factors:
# It is easy to say that the terminating loop condition is p < n, which is certain
# ly correct but not efficient. Do you still remember Hint #3?
# 4 is not a prime because it is divisible by 2, which means all multiples of 4 mu
# st also be divisible by 2 and were already marked off. So we can skip 4 immediat
# ely and go to the next number, 5. Now, all multiples of 5 such as 5 × 2 = 10, 5
# × 3 = 15, 5 × 4 = 20, 5 × 5 = 25, ... can be marked off. There is a slight optim
# ization here, we do not need to start from 5 × 2 = 10. Where should we start mar
# king off?
# Our total runtime has now improved to O(n1.5), which is slightly better. Is ther
# e a faster approach?
# As you can see, calculations of 4 × 3 and 6 × 2 are not necessary. Therefore, we
#  only need to consider factors up to n because, if n is divisible by some numbe
# r p, then n = p × q and since p  q, we could derive that p  n.
# Yes, the terminating loop condition can be p < n, as all non-primes  n must h
# ave already been marked off. When the loop terminates, all the numbers in the ta
# ble that are non-marked are prime.
# As we know the number must not be divisible by any number > n / 2, we can immedi
# ately cut the total iterations half by dividing only up to n / 2. Could we still
#  do better?
# Let's start with a isPrime function. To determine if a number is prime, we need
# to check if it is not divisible by any number less than n. The runtime complexit
# y of isPrime function would be O(n) and hence counting the total prime numbers u
# p to n would be O(n2). Could we do better?
# Count the number of prime numbers less than a non-negative number, n.
# Example:
# Sieve of Eratosthenes: algorithm steps for primes below 121. "Sieve of Eratosthe
# nes Animation" by SKopp is licensed under CC BY 2.0.

# NOTE
#

# EXAMPLE
# Input: 10
# Output: 4
# Explanation: There are 4 prime numbers less than 10, they ar
# e 2, 3, 5, 7.

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

```python
class Solution(object):
    # @param {integer} n
    # @return {integer}
    def countPrimes(self, n):
        if n <= 2:
            return 0

        is_prime = [True]*(n//2)
        cnt = len(is_prime)
        for i in xrange(3, n, 2):
            if i * i >= n:
                break
            if not is_prime[i//2]:
                continue
            for j in xrange(i*i, n, 2*i):
                if not is_prime[j//2]:
                    continue
                cnt -= 1
                is_prime[j//2] = False

        return cnt

    def countPrimes2(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n < 3:
            return 0
        primes = [True] * n
        primes[0] = primes[1] = False
        for i in range(2, int(n ** 0.5) + 1):
            if primes[i]:
                primes[i * i: n: i] = [False] * len(primes[i * i: n: i])
        return sum(primes)
```

sort-array-by-parity-ii.py

```python
# DESC
# Given an array A of non-negative integers, half of the integers in A are odd, an
# d half of the integers are even.
# Example 1:
# Sort the array so that whenever A[i] is odd, i is odd; and whenever A[i] is even
# , i is even.
# You may return any answer array that satisfies this condition.
# Note:

# NOTE
# A.length % 2 == 0
# 0 <= A[i] <= 1000
# 2 <= A.length <= 20000

# EXAMPLE
# Input: [4,2,5,7]
# Output: [4,5,2,7]
# Explanation: [4,7,2,5], [2,5,4,7], [2,7,4,5]
# would also have been accepted.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def sortArrayByParityII(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        j = 1
        for i in xrange(0, len(A), 2):
            if A[i] % 2:
                while A[j] % 2:
                    j += 2
                A[i], A[j] = A[j], A[i]
        return A
```

word-ladder-ii.py

```
# DESC
# Given two words (beginWord and endWord), and a dictionary's word list, find all
# shortest transformation sequence(s) from beginWord to endWord, such that:
# Example 2:
# Example 1:
# Note:

# NOTE
# You may assume no duplicates in the word list.
# Each transformed word must exist in the word list. Note that beginWord is not a
# transformed word.
# Only one letter can be changed at a time
# Return an empty list if there is no such transformation sequence.
# All words have the same length.
# All words contain only lowercase alphabetic characters.
# You may assume beginWord and endWord are non-empty and are not the same.

# EXAMPLE
# Input:
# beginWord = "hit",
# endWord = "cog",
# wordList = ["hot","dot","dog","lot","
# log","cog"]
#
# Output:
# [
#   ["hit","hot","dot","dog","cog"],
#   ["hit","hot","lot","
# log","cog"]
# ]
# Input:
# beginWord = "hit"
# endWord = "cog"
# wordList = ["hot","dot","dog","lot","lo
# g"]
#
# Output: []
#
# 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 collections import defaultdict
from string import ascii_lowercase


class Solution(object):
    def findLadders(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: List[str]
        :rtype: List[List[str]]
        """
        dictionary = set(wordList)
        result, cur, visited, found, trace = [], [beginWord], set([beginWord]), False, defaultdict(list)
```

```
        while cur and not found:
            for word in cur:
                visited.add(word)

            next = set()
            for word in cur:
                for i in xrange(len(word)):
                    for c in ascii_lowercase:
                        candidate = word[:i] + c + word[i + 1:]
                        if candidate not in visited and candidate in dictionary:
                            if candidate == endWord:
                                found = True
                            next.add(candidate)
                            trace[candidate].append(word)
            cur = next

        if found:
            self.backtrack(result, trace, [], endWord)

        return result

    def backtrack(self, result, trace, path, word):
        if not trace[word]:
            result.append([word] + path)
        else:
            for prev in trace[word]:
                self.backtrack(result, trace, [word] + path, prev)
```

find-all-numbers-disappeared-in-an-array.py

```python
# find-all-numbers-disappeared-in-an-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findDisappearedNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        for i in xrange(len(nums)):
            if nums[abs(nums[i]) - 1] > 0:
                nums[abs(nums[i]) - 1] *= -1

        result = []
        for i in xrange(len(nums)):
            if nums[i] > 0:
                result.append(i+1)
            else:
                nums[i] *= -1
        return result

    def findDisappearedNumbers2(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        return list(set(range(1, len(nums) + 1)) - set(nums))

    def findDisappearedNumbers3(self, nums):
        for i in range(len(nums)):
            index = abs(nums[i]) - 1
            nums[index] = - abs(nums[index])

        return [i + 1 for i in range(len(nums)) if nums[i] > 0]
```

matchsticks-to-square.py

```
# DESC
# Note:
# Example 2:
# Your input will be several matchsticks the girl has, represented with their stic
# k length. Your output will either be true or false, to represent whether you cou
# ld make one square using all the matchsticks the little match girl has.
# Remember the story of Little Match Girl? By now, you know exactly what matchstic
# ks the little match girl has, please find out a way you can make one square by u
# sing 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.
# Example 1:

# 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.

# EXAMPLE
# Input: [3,3,3,3,4]
# Output: false
#
# Explanation: You cannot find a way to form a s
# quare with all the matchsticks.
# 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.

# Time:  O(n * s * 2^n), s is the number of subset of which sum equals to side length.
# 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:
                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
```

powx-n.py

```
# powx-n is not found.
# Time:  O(logn) = O(1)
# Space: O(1)

class Solution(object):
    def myPow(self, x, n):
        """
        :type x: float
        :type n: int
        :rtype: float
        """
        result = 1
        abs_n = abs(n)
        while abs_n:
            if abs_n & 1:
                result *= x
            abs_n >>= 1
            x *= x

        return 1 / result if n < 0 else result


# Time:  O(logn)
# Space: O(logn)
# Recursive solution.
class Solution2(object):
    def myPow(self, x, n):
        """
        :type x: float
        :type n: int
        :rtype: float
        """
        if n < 0 and n != -n:
            return 1.0 / self.myPow(x, -n)
        if n == 0:
            return 1
        v = self.myPow(x, n / 2)
        if n % 2 == 0:
            return v * v
        else:
            return v * v * x
```

online-election.py

```
# DESC
# TopVotedCandidate.q(int t)
# Votes cast at time t will count towards our query.  In the case of a tie, the mo
# st recent vote (among tied candidates) wins.
# Note:
# Now, we would like to implement the following query function: TopVotedCandidate.
# q(int t) will return the number of the person that was leading the election at t
# ime t.
# In an election, the i-th vote was cast for persons[i] at time times[i].
# Example 1:

# NOTE
# TopVotedCandidate.q(int t) is always called with t >= times[0].
# 1 <= persons.length = times.length <= 5000
# TopVotedCandidate.q is called at most 10000 times per test case.
# 0 <= persons[i] <= persons.length
# times is a strictly increasing array with all elements in [0, 10^9].

# EXAMPLE
# Input: ["TopVotedCandidate","q","q","q","q","q","q"], [[[0,1,1,0,0,1,0],[0,5,10,
# 15,20,25,30]],[3],[12],[25],[15],[24],[8]]
# Output: [null,0,1,1,0,0,1]
# Explanatio
# n:
# At time 3, the votes are [0], and 0 is leading.
# At time 12, the votes are [0
# ,1,1], and 1 is leading.
# At time 25, the votes are [0,1,1,0,0,1], and 1 is leadi
# ng (as ties go to the most recent vote.)
# This continues for 3 more queries at ti
# me 15, 24, and 8.

# Time:  ctor: O(n)
#        q:    O(logn)
# Space: O(n)

import collections
import itertools
import bisect


class TopVotedCandidate(object):

    def __init__(self, persons, times):
        """
        :type persons: List[int]
        :type times: List[int]
        """
        lead = -1
        self.__lookup, count = [], collections.defaultdict(int)
        for t, p in itertools.izip(times, persons):
            count[p] += 1
            if count[p] >= count[lead]:
                lead = p
                self.__lookup.append((t, lead))

    def q(self, t):
        """
```

```
        :type t: int
        :rtype: int
        """
        return self.__lookup[bisect.bisect(self.__lookup,
                                           (t, float("inf")))-1][1]
```

remove-zero-sum-consecutive-nodes-from-linked-list.py

```
# remove-zero-sum-consecutive-nodes-from-linked-list is not found.
# 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:
            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
```

range-addition.py

```python
# range-addition is not found.
# Time:  O(k + n)
# Space: O(1)

class Solution(object):
    def getModifiedArray(self, length, updates):
        """
        :type length: int
        :type updates: List[List[int]]
        :rtype: List[int]
        """
        result = [0] * length
        for update in updates:
            result[update[0]] += update[2]
            if update[1]+1 < length:
                result[update[1]+1] -= update[2]

        for i in xrange(1, length):
            result[i] += result[i-1]

        return result
```

sort-list.py

```
# DESC
# Example 1:
# Example 2:
# Sort a linked list in O(n log n) time using constant space complexity.

# NOTE
#

# EXAMPLE
# Input: 4->2->1->3
# Output: 1->2->3->4
# 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
```

kill-process.py

```
# kill-process is not found.
# Time:  O(n)
# Space: O(n)

import collections


# DFS solution.
class Solution(object):
    def killProcess(self, pid, ppid, kill):
        """
        :type pid: List[int]
        :type ppid: List[int]
        :type kill: int
        :rtype: List[int]
        """
        def killAll(pid, children, killed):
            killed.append(pid)
            for child in children[pid]:
                killAll(child, children, killed)

        result = []
        children = collections.defaultdict(set)
        for i in xrange(len(pid)):
            children[ppid[i]].add(pid[i])
        killAll(kill, children, result)
        return result


# Time:  O(n)
# Space: O(n)
# BFS solution.
class Solution2(object):
    def killProcess(self, pid, ppid, kill):
        """
        :type pid: List[int]
        :type ppid: List[int]
        :type kill: int
        :rtype: List[int]
        """
        def killAll(pid, children, killed):
            killed.append(pid)
            for child in children[pid]:
                killAll(child, children, killed)

        result = []
        children = collections.defaultdict(set)
        for i in xrange(len(pid)):
            children[ppid[i]].add(pid[i])
        q = collections.deque()
        q.append(kill)
        while q:
            p = q.popleft()
            result.append(p)
            for child in children[p]:
                q.append(child)
        return result
```

strobogrammatic-number-ii.py

```python
# strobogrammatic-number-ii is not found.
# Time:  O(n^2 * 5^(n/2))
# Space: O(n)

class Solution(object):
    lookup = {'0':'0', '1':'1', '6':'9', '8':'8', '9':'6'}

    # @param {integer} n
    # @return {string[]}
    def findStrobogrammatic(self, n):
        return self.findStrobogrammaticRecu(n, n)

    def findStrobogrammaticRecu(self, n, k):
        if k == 0:
            return ['']
        elif k == 1:
            return ['0', '1', '8']

        result = []
        for num in self.findStrobogrammaticRecu(n, k - 2):
            for key, val in self.lookup.iteritems():
                if n != k or key != '0':
                    result.append(key + num + val)

        return result
```

guess-number-higher-or-lower-ii.py

```
# DESC
# Example:
# We are playing the Guess Game. The game is as follows:
# Given a particular n  1, find out how much money you need to have to guarantee
# a win.
# 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.

# NOTE
#

# EXAMPLE
# n = 10, I pick 8.
#
# First round:  You guess 5, I tell you that it's higher. You p
# ay $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 t
# he number I picked.
#
# You end up paying $5 + $7 + $9 = $21.

# 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]
```

height-checker.py

```
# DESC
# Students are asked to stand in non-decreasing order of heights for an annual photo.
# Return the minimum number of students that must move in order for all students t
# o be standing in non-decreasing order of height.
# Example 2:
# Notice that when a group of students is selected they can reorder in any possibl
# e way between themselves and the non selected students remain on their seats.
# Example 3:
# Example 1:
# Constraints:

# NOTE
# 1 <= heights[i] <= 100
# 1 <= heights.length <= 100

# EXAMPLE
# Input: heights = [1,1,4,2,1,3]
# Output: 3
# Explanation:
# Current array : [1,1,4,2,
# 1,3]
# Target array  : [1,1,1,2,3,4]
# On index 2 (0-based) we have 4 vs 1 so we hav
# e to move this student.
# On index 4 (0-based) we have 1 vs 3 so we have to move t
# his student.
# On index 5 (0-based) we have 3 vs 4 so we have to move this student
# .
# Input: heights = [1,2,3,4,5]
# Output: 0
# Input: heights = [5,1,2,3,4]
# Output: 5

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

import itertools


class Solution(object):
    def heightChecker(self, heights):
        """
        :type heights: List[int]
        :rtype: int
        """
        return sum(i != j for i, j in itertools.izip(heights, sorted(heights)))
```

flatten-nested-list-iterator.py

```
# DESC
# Each element is either an integer, or a list -- whose elements may also be integ
# ers or other lists.
# Example 2:
# Given a nested list of integers, implement an iterator to flatten it.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: [1,[4,[6]]]
# Output: [1,4,6]
# Explanation: By calling next repeatedly until
#  hasNext returns false,
#              the order of elements returned by next sho
# uld be: [1,4,6].
# Input: [[1,1],2,[1,1]]
# Output: [1,1,2,1,1]
# Explanation: By calling next repeated
# ly until hasNext returns false,
#              the order of elements returned by
# next should be: [1,1,2,1,1].

# Time:  O(n), n is the number of the integers.
# 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

```
# DESC
# For example, given the following tree:
# The flattened tree should look like:
# Given a binary tree, flatten it to a linked list in-place.

# NOTE
#

# EXAMPLE
# 1
#    / \
#   2   5
#  / \   \
# 3   4   6
# 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

```
# DESC
# Note:
# 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 paper
# s have no more than h citations each."
# If there are several possible values for h, the maximum one is taken as the h-index.
# Given an array of citations sorted in ascending order (each citation is a non-ne
# gative integer) of a researcher, write a function to compute the researcher's h-
# index.
# Example:
# Follow up:

# NOTE
# Could you solve it in logarithmic time complexity?
# This is a follow up problem to H-Index, where citations is now guaranteed to be
# sorted in ascending order.

# EXAMPLE
# Input: citations = [0,1,3,5,6]
# Output: 3
# Explanation: [0,1,3,5,6] means the res
# earcher 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.

# 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
```

longest-increasing-path-in-a-matrix.py

```
# DESC
# Example 1:
# Given an integer matrix, find the length of the longest increasing path.
# Example 2:
# From each cell, you can either move to four directions: left, right, up or down.
#  You may NOT move diagonally or move outside of the boundary (i.e. wrap-around i
# s not allowed).

# NOTE
#

# EXAMPLE
# Input: nums =
# [
#   [3,4,5],
#   [3,2,6],
#   [2,2,1]
# ]
# Output: 4
# Explanation: The
# longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.
# Input: nums =
# [
#   [9,9,4],
#   [6,6,8],
#   [2,1,1]
# ]
# Output: 4
# Explanation: The
# longest increasing path is [1, 2, 6, 9].

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

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

        def longestpath(matrix, i, j, max_lengths):
            if max_lengths[i][j]:
                return max_lengths[i][j]

            max_depth = 0
            directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
            for d in directions:
                x, y = i + d[0], j + d[1]
                if 0 <= x < len(matrix) and 0 <= y < len(matrix[0]) and \
                   matrix[x][y] < matrix[i][j]:
                    max_depth = max(max_depth, longestpath(matrix, x, y, max_lengths))
            max_lengths[i][j] = max_depth + 1
            return max_lengths[i][j]

        res = 0
```

```
max_lengths = [[0 for _ in xrange(len(matrix[0]))] for _ in xrange(len(matrix))]
for i in xrange(len(matrix)):
    for j in xrange(len(matrix[0])):
        res = max(res, longestpath(matrix, i, j, max_lengths))

return res
```

maximum-depth-of-n-ary-tree.py

```python
# DESC
# The maximum depth is the number of nodes along the longest path from the root no
# de down to the farthest leaf node.
# Nary-Tree input serialization is represented in their level order traversal, eac
# h group of children is separated by the null value (See examples).
# Example 1:
# Example 2:
# Constraints:
# Given a n-ary tree, find its maximum depth.

# NOTE
# The total number of nodes is between [0, 10^4].
# The depth of the n-ary tree is less than or equal to 1000.

# EXAMPLE
# Input: root = [1,null,3,2,4,null,5,6]
# Output: 3
# Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,1
# 2,null,13,null,null,14]
# Output: 5

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

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


class Solution(object):
    def maxDepth(self, root):
        """
        :type root: Node
        :rtype: int
        """
        if not root:
            return 0
        depth = 0
        for child in root.children:
            depth = max(depth, self.maxDepth(child))
        return 1+depth
```

path-crossing.py

```python
# DESC
# Constraints:
# Given a string path, where path[i] = 'N', 'S', 'E' or 'W', each representing mov
# ing one unit north, south, east, or west, respectively. You start at the origin
# (0, 0) on a 2D plane and walk on the path specified by path.
# Return True if the path crosses itself at any point, that is, if at any time you
#  are on a location you've previously visited. Return False otherwise.
# Example 1:
# Example 2:

# NOTE
# path will only consist of characters in {'N', 'S', 'E', 'W}
# 1 <= path.length <= 10^4

# EXAMPLE
# Input: path = "NESWW"
# Output: true
# Explanation: Notice that the path visits the
# origin twice.
# Input: path = "NES"
# Output: false
# Explanation: Notice that the path doesn't cro
# ss any point more than once.

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

class Solution(object):
    def isPathCrossing(self, path):
        """
        :type path: str
        :rtype: bool
        """
        x = y = 0
        lookup = {(0, 0)}
        for c in path:
            if c == 'E':
                x += 1
            elif c == 'W':
                x -= 1
            elif c == 'N':
                y += 1
            elif c == 'S':
                y -= 1
            if (x, y) in lookup:
                return True
            lookup.add((x, y))
        return False
```

insert-into-a-binary-search-tree.py

```
# DESC
# Note that there may exist multiple valid ways for the insertion, as long as the
# tree remains a BST after insertion. You can return any of them.
# This tree is also valid:
# Given the root node of a binary search tree (BST) and a value to be inserted int
# o the tree, insert the value into the BST. Return the root node of the BST after
#  the insertion. It is guaranteed that the new value does not exist in the origin
# al BST.
# Constraints:
# You can return this binary search tree:
# For example,

# NOTE
# Each node will have a unique integer value from 0 to -10^8, inclusive.
# -10^8 <= val <= 10^8
# The number of nodes in the given tree will be between 0 and 10^4.
# It's guaranteed that val does not exist in the original BST.

# EXAMPLE
# 5
#        /   \
#       2     7
#      / \
#     1   3
#           \
#            4
# Given the tree:
#         4
#        / \
#       2   7
#      / \
#     1   3
# And the valu
# e to insert: 5
# 4
#        /   \
#       2     7
#      / \   /
#     1   3 5

# Time:  O(h)
# Space: O(1)

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


class Solution(object):
    def insertIntoBST(self, root, val):
        """
        :type root: TreeNode
        :type val: int
        :rtype: TreeNode
        """
        curr, parent = root, None
```

```
        while curr:
            parent = curr
            if val <= curr.val:
                curr = curr.left
            else:
                curr = curr.right
        if not parent:
            root = TreeNode(val)
        elif val <= parent.val:
            parent.left = TreeNode(val)
        else:
            parent.right = TreeNode(val)
        return root


# Time:  O(h)
# Space: O(h)
class Solution2(object):
    def insertIntoBST(self, root, val):
        """
        :type root: TreeNode
        :type val: int
        :rtype: TreeNode
        """
        if not root:
            root = TreeNode(val)
        else:
            if val <= root.val:
                root.left = self.insertIntoBST(root.left, val)
            else:
                root.right = self.insertIntoBST(root.right, val)
        return root
```

average-salary-excluding-the-minimum-and-maximum-salary.py

```python
# average-salary-excluding-the-minimum-and-maximum-salar is not found.
# Time:  O(n)
# Space: O(1)

# one pass solution
class Solution(object):
    def average(self, salary):
        """
        :type salary: List[int]
        :rtype: float
        """
        total, mi, ma = 0, float("inf"), float("-inf")
        for s in salary:
            total += s
            mi, ma = min(mi, s), max(ma, s)
        return 1.0*(total-mi-ma)/(len(salary)-2)


# Time:  O(n)
# Space: O(1)
# one-liner solution
class Solution2(object):
    def average(self, salary):
        """
        :type salary: List[int]
        :rtype: float
        """
        return 1.0*(sum(salary)-min(salary)-max(salary))/(len(salary)-2)
```

check-if-all-1s-are-at-least-length-k-places-away.py

```python
# check-if-all-1s-are-at-least-length-k-places-awa is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def kLengthApart(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: bool
        """
        prev = -k-1
        for i in xrange(len(nums)):
            if not nums[i]:
                continue
            if i-prev <= k:
                return False
            prev = i
        return True
```

longest-happy-prefix.py

```python
# longest-happy-prefix is not found.
# Time:  O(n)
# Space: O(n)

# kmp solution
class Solution(object):
    def longestPrefix(self, s):
        """
        :type s: str
        :rtype: str
        """
        def getPrefix(pattern):
            prefix = [-1]*len(pattern)
            j = -1
            for i in xrange(1, len(pattern)):
                while j != -1 and pattern[j+1] != pattern[i]:
                    j = prefix[j]
                if pattern[j+1] == pattern[i]:
                    j += 1
                prefix[i] = j
            return prefix

        return s[:getPrefix(s)[-1]+1]


# Time:  O(n) on average
# Space: O(1)
# rolling-hash solution
class Solution2(object):
    def longestPrefix(self, s):
        """
        :type s: str
        :rtype: str
        """
        M = 10**9+7
        D = 26
        def check(l, s):
            for i in xrange(l):
                if s[i] != s[len(s)-l+i]:
                    return False
            return True

        result, prefix, suffix, power = 0, 0, 0, 1
        for i in xrange(len(s)-1):
            prefix = (prefix*D + (ord(s[i])-ord('a'))) % M
            suffix = (suffix + (ord(s[len(s)-(i+1)])-ord('a'))*power) % M
            power = (power*D)%M
            if prefix == suffix:
                # we assume M is a very large prime without hash collision
                # assert(check(i+1, s))
                result = i+1
        return s[:result]
```

number-of-lines-to-write-string.py

```
# DESC
# Note:
# Now answer two questions: how many lines have at least one character from S, and
#  what is the width used by the last such line? Return your answer as an integer
# list of length 2.
# We are to write the letters of a given string S, from left to right into lines.
# Each line has maximum width 100 units, and if writing a letter would cause the w
# idth of the line to exceed 100 units, it is written on the next line. We are giv
# en an array widths, an array where widths[0] is the width of 'a', widths[1] is t
# he width of 'b', ..., and widths[25] is the width of 'z'.

# NOTE
# S will only contain lowercase letters.
# widths[i] will be in the range of [2, 10].
# widths is an array of length 26.
# The length of S will be in the range [1, 1000].

# EXAMPLE
# Example :
# Input:
# widths = [10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,1
# 0,10,10,10,10,10,10,10,10,10]
# S = "abcdefghijklmnopqrstuvwxyz"
# Output: [3, 60]
# Expl
# anation:
# All letters have the same length of 10. To write all 26 letters,
# we ne
# ed two full lines and one line with 60 units.
# Example :
# Input:
# widths = [4,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10
# ,10,10,10,10,10,10,10,10,10]
# S = "bbbccccdddaaa"
# Output: [2, 4]
# Explanation:
# All le
# tters except 'a' have the same length of 10, and
# "bbbccccdddaa" will cover 9 * 1
# 0 + 2 * 4 = 98 units.
# For the last 'a', it is written on the second line because
#
# there is only 2 units left in the first line.
# So the answer is 2 lines, plus 4
# units in the second line.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numberOfLines(self, widths, S):
        """
        :type widths: List[int]
        :type S: str
        :rtype: List[int]
        """
        result = [1, 0]
        for c in S:
```

```
    w = widths[ord(c)-ord('a')]
    result[1] += w
    if result[1] > 100:
        result[0] += 1
        result[1] = w
return result
```

find-kth-bit-in-nth-binary-string.py

```python
# find-kth-bit-in-nth-binary-string is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findKthBit(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        flip, l = 0, 2**n-1
        while k > 1:
            if k == l//2+1:
                flip ^= 1
                break
            if k > l//2:
                k = l+1-k
                flip ^= 1
            l //= 2
        return str(flip)
```

prime-number-of-set-bits-in-binary-representation.py

```
# DESC
# (Recall that the number of set bits an integer has is the number of 1s present w
# hen written in binary.  For example, 21 written in binary is 10101 which has 3 s
# et bits.  Also, 1 is not a prime.)
# Example 2:
# Given two integers L and R, find the count of numbers in the range [L, R] (inclu
# sive) having a prime number of set bits in their binary representation.
# Note:
# Example 1:

# NOTE
# R - L will be at most 10000.
# L, R will be integers L <= R in the range [1, 10^6].

# EXAMPLE
# Input: L = 10, R = 15
# Output: 5
# Explanation:
# 10 -> 1010 (2 set bits, 2 is prime)
#
# 11 -> 1011 (3 set bits, 3 is prime)
# 12 -> 1100 (2 set bits, 2 is prime)
# 13 -> 1
# 101 (3 set bits, 3 is prime)
# 14 -> 1110 (3 set bits, 3 is prime)
# 15 -> 1111 (4 s
# et bits, 4 is not prime)
# Input: L = 6, R = 10
# Output: 4
# Explanation:
# 6 -> 110 (2 set bits, 2 is prime)
# 7
# -> 111 (3 set bits, 3 is prime)
# 9 -> 1001 (2 set bits , 2 is prime)
# 10->1010 (2
# set bits , 2 is prime)

# Time:  O(log(R - L)) = O(1)
# Space: O(1)

class Solution(object):
    def countPrimeSetBits(self, L, R):
        """
        :type L: int
        :type R: int
        :rtype: int
        """
        def bitCount(n):
            result = 0
            while n:
                n &= n-1
                result += 1
            return result

        primes = {2, 3, 5, 7, 11, 13, 17, 19}
        return sum(bitCount(i) in primes
                   for i in xrange(L, R+1))
```

path-sum-ii.py

```
# DESC
# Given the below binary tree and sum = 22,
# Note: A leaf is a node with no children.
# Return:
# Example:
# Given a binary tree and a sum, find all root-to-leaf paths where each path's sum
#  equals the given sum.

# NOTE
#

# EXAMPLE
# [
#    [5,4,11,2],
#    [5,8,4,5]
# ]
# 5
#      / \
#     4   8
#    /   / \
#   11  13  4
#  /  \    / \
# 7    2  5   1

# 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
```

ternary-expression-parser.py

```python
# ternary-expression-parser is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def parseTernary(self, expression):
        """
        :type expression: str
        :rtype: str
        """
        if not expression:
            return ""

        stack = []
        for c in expression[::-1]:
            if stack and stack[-1] == '?':
                stack.pop()  # pop '?'
                first = stack.pop()
                stack.pop()  # pop ':'
                second = stack.pop()

                if c == 'T':
                    stack.append(first)
                else:
                    stack.append(second)
            else:
                stack.append(c)


        return str(stack[-1])
```

find-in-mountain-array.py

```
# find-in-mountain-arra is not found.
# Time:  O(logn)
# Space: O(1)

# """
# This is MountainArray's API interface.
# You should not implement it, or speculate about its implementation
# """
class MountainArray(object):
   def get(self, index):
       """
       :type index: int
       :rtype int
       """
       pass

   def length(self):
       """
       :rtype int
       """
       pass


class Solution(object):
    def findInMountainArray(self, target, mountain_arr):
        """
        :type target: integer
        :type mountain_arr: MountainArray
        :rtype: integer
        """
        def binarySearch(A, left, right, check):
            while left <= right:
                mid = left + (right-left)//2
                if check(mid):
                    right = mid-1
                else:
                    left = mid+1
            return left

        peak = binarySearch(mountain_arr, 0, mountain_arr.length()-1,
                            lambda x: mountain_arr.get(x) >= mountain_arr.get(x+1))
        left = binarySearch(mountain_arr, 0, peak,
                            lambda x: mountain_arr.get(x) >= target)
        if left <= peak and mountain_arr.get(left) == target:
            return left
        right = binarySearch(mountain_arr, peak, mountain_arr.length()-1,
                             lambda x: mountain_arr.get(x) <= target)
        if right <= mountain_arr.length()-1 and mountain_arr.get(right) == target:
            return right
        return -1
```

beautiful-arrangement.py

```
# DESC
# 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 followi
# ng is true for the ith position (1 <= i <= N) in this array:
# Example 1:
# Note:
# Now given N, how many beautiful arrangements can you construct?

# NOTE
# The number at the ith position is divisible by i.
# N is a positive integer and will not exceed 15.
# i is divisible by the number at the ith position.

# EXAMPLE
# Input: 2
# Output: 2
# Explanation:
#
# The first beautiful arrangement is [1, 2]:
#
# Nu
# mber at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).
#
# Number at t
# he 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 divisib
# le by i (i=1).
#
# Number at the 2nd position (i=2) is 1, and i (i=2) is divisible
# by 1.

# 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))
```

path-sum-iv.py

```
# path-sum-iv is not found.
# Time:  O(n)
# Space: O(p), p is the number of paths

import collections


class Solution(object):
    def pathSum(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        class Node(object):
            def __init__(self, num):
                self.level = num/100 - 1
                self.i = (num%100)/10 - 1
                self.val = num%10
                self.leaf = True

            def isParent(self, other):
                return self.level == other.level-1 and \
                       self.i == other.i/2

        if not nums:
            return 0
        result = 0
        q = collections.deque()
        dummy = Node(10)
        parent = dummy
        for num in nums:
            child = Node(num)
            while not parent.isParent(child):
                result += parent.val if parent.leaf else 0
                parent = q.popleft()
            parent.leaf = False
            child.val += parent.val
            q.append(child)
        while q:
            result += q.pop().val
        return result
```

rotate-function.py

```
# DESC
# Note:
#
# n is guaranteed to be less than 105.
# F(k) = 0 * Bk[0] + 1 * Bk[1] + ... + (n-1) * Bk[n-1].
# 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:
# Calculate the maximum value of F(0), F(1), ..., F(n-1).
# Given an array of integers A and let n to be its length.
# Example:

# NOTE
#

# 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

```
# DESC
# Note:
# 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 t
# o become the maximum value of that subarray.
# Return the largest sum of the given array after partitioning.
# Example 1:

# NOTE
# 0 <= A[i] <= 10^6
# 1 <= K <= A.length <= 500

# EXAMPLE
# Input: A = [1,15,7,9,2,5,10], K = 3
# Output: 84
# Explanation: A becomes [15,15,15,
# 9,10,10,10]

# 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

```
# DESC
# Example 1:
# Example 2:
# Each special offer is represented in the form of an array, the last number repre
# sents 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.
# Note:
# You could use any of special offers as many times as you want.
# 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 m
# ore 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 spe
# cial offers.

# NOTE
# For each item, you need to buy at most 6 of them.
# There are at most 6 kinds of items, 100 special offers.
# You are not allowed to buy more items than you want, even if that would lower th
# e overall price.

# EXAMPLE
# Input: [2,5], [[3,0,5],[1,2,10]], [3,2]
# Output: 14
# Explanation:
# There are two k
# inds of items, A and B. Their prices are $2 and $5 respectively.
# In special off
# er 1, you can pay $5 for 3A and 0B
# In special offer 2, you can pay $10 for 1A an
# d 2B.
# You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offe
# r #2), and $4 for 2A.
# Input: [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]
# Output: 11
# Explanation:
# The pric
# e 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.

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

get-watched-videos-by-your-friends.py

```
# DESC
# Level 1 of videos are all watched videos by your friends, level 2 of videos are
# all watched videos by the friends of your friends and so on. In general, the lev
# el k of videos are all watched videos by people with the shortest path exactly e
# qual to k with you. Given your id and the level of videos, return the list of vi
# deos ordered by their frequencies (increasing). For videos with the same frequen
# cy order them alphabetically from least to greatest.
# Constraints:
# Example 1:
# Example 2:
# There are n people, each person has a unique id between 0 and n-1. Given the arr
# ays watchedVideos and friends, where watchedVideos[i] and friends[i] contain the
#  list of watched videos and the list of friends respectively for the person with
#  id = i.

# NOTE
# n == watchedVideos.length == friends.length
# 0 <= id < n
# 1 <= watchedVideos[i].length <= 100
# if friends[i] contains j, then friends[j] contains i
# 1 <= level < n
# 0 <= friends[i].length < n
# 0 <= friends[i][j] < n
# 2 <= n <= 100
# 1 <= watchedVideos[i][j].length <= 8

# EXAMPLE
# Input: watchedVideos = [["A","B"],["C"],["B","C"],["D"]], friends = [[1,2],[0,3]
# ,[0,3],[1,2]], id = 0, level = 1
# Output: ["B","C"]
# Explanation:
# You have id =
# 0 (green color in the figure) and your friends are (yellow color in the figure):
#
# Person with id = 1 -> watchedVideos = ["C"]
# Person with id = 2 -> watchedVideo
# s = ["B","C"]
# The frequencies of watchedVideos by your friends are:
# B -> 1
# C
# -> 2
# Input: watchedVideos = [["A","B"],["C"],["B","C"],["D"]], friends = [[1,2],[0,3]
# ,[0,3],[1,2]], id = 0, level = 2
# Output: ["D"]
# Explanation:
# You have id = 0 (gr
# een color in the figure) and the only friend of your friends is the person with
# id = 3 (yellow color in the figure).

# Time:  O(n + vlogv), v is the number of the level videos
# Space: O(w)

import collections


class Solution(object):
    def watchedVideosByFriends(self, watchedVideos, friends, id, level):
        """
```

```
        :type watchedVideos: List[List[str]]
        :type friends: List[List[int]]
        :type id: int
        :type level: int
        :rtype: List[str]
        """
        curr_level, lookup = set([id]), set([id])
        for _ in xrange(level):
            curr_level = set(j for i in curr_level for j in friends[i] if j not in lookup)
            lookup |= curr_level
        count = collections.Counter([v for i in curr_level for v in watchedVideos[i]])
        return sorted(count.keys(), key=lambda x: (count[x], x))
```

best-time-to-buy-and-sell-stock.py

```python
# DESC
# Example 2:
# Say you have an array for which the ith element is the price of a given stock on
#  day i.
# Example 1:
# If you were only permitted to complete at most one transaction (i.e., buy one an
# d sell one share of the stock), design an algorithm to find the maximum profit.
# Note that you cannot sell a stock before you buy one.

# NOTE
#

# EXAMPLE
# Input: [7,1,5,3,6,4]
# Output: 5
# Explanation: Buy on day 2 (price = 1) and sell on
#  day 5 (price = 6), profit = 6-1 = 5.
#              Not 7-1 = 6, as selling price
#  needs to be larger than buying price.
# Input: [7,6,4,3,1]
# Output: 0
# Explanation: In this case, no transaction is done,
# i.e. max profit = 0.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param prices, a list of integer
    # @return an integer
    def maxProfit(self, prices):
        max_profit, min_price = 0, float("inf")
        for price in prices:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)
        return max_profit
```

unique-number-of-occurrences.py

```python
# unique-number-of-occurrences is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def uniqueOccurrences(self, arr):
        """
        :type arr: List[int]
        :rtype: bool
        """
        count = collections.Counter(arr)
        lookup = set()
        for v in count.itervalues():
            if v in lookup:
                return False
            lookup.add(v)
        return True


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def uniqueOccurrences(self, arr):
        """
        :type arr: List[int]
        :rtype: bool
        """
        count = collections.Counter(arr)
        return len(count) == len(set(count.itervalues()))
```

perfect-rectangle.py

```
# DESC
# Example 4:
# Each rectangle is represented as a bottom-left point and a top-right point. For
# example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left p
# oint is (1, 1) and top-right point is (2, 2)).
# Given N axis-aligned rectangles where N > 0, determine if they all together form
#  an exact cover of a rectangular region.
# Example 2:
# Example 3:
# Example 1:

# NOTE
#

# EXAMPLE
# rectangles = [
#   [1,1,3,3],
#   [3,1,4,2],
#   [3,2,4,4],
#   [1,3,2,4],
#   [2,3,3,4]
# ]
#
#
# Return true. All 5 rectangles together form an exact cover of a rectangular re
# gion.
# rectangles = [
#   [1,1,3,3],
#   [3,1,4,2],
#   [1,3,2,4],
#   [2,2,4,4]
# ]
#
# Return fals
# e. Because two of the rectangles overlap with each other.
# rectangles = [
#   [1,1,3,3],
#   [3,1,4,2],
#   [1,3,2,4],
#   [3,2,4,4]
# ]
#
# Return fals
# e. Because there is a gap in the top center.
# rectangles = [
#   [1,1,2,3],
#   [1,3,2,4],
#   [3,1,4,2],
#   [3,2,4,4]
# ]
#
# Return fals
# e. Because there is a gap between the two rectangular regions.

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

from collections import defaultdict
```

```
class Solution(object):
    def isRectangleCover(self, rectangles):
        """
        :type rectangles: List[List[int]]
        :rtype: bool
        """
        left = min(rec[0] for rec in rectangles)
        bottom = min(rec[1] for rec in rectangles)
        right = max(rec[2] for rec in rectangles)
        top = max(rec[3] for rec in rectangles)

        points = defaultdict(int)
        for l, b, r, t in rectangles:
            for p, q in zip(((l, b), (r, b), (l, t), (r, t)), (1, 2, 4, 8)):
                if points[p] & q:
                    return False
                points[p] |= q

        for px, py in points:
            if left < px < right or bottom < py < top:
                if points[(px, py)] not in (3, 5, 10, 12, 15):
                    return False

        return True
```

find-and-replace-in-string.py

```
# DESC
# Example 1:
# To some string S, we will perform some replacement operations that replace group
# s of letters with new ones (not necessarily the same size).
# Example 2:
# Each replacement operation has 3 parameters: a starting index i, a source word x
#  and a target word y.  The rule is that if x starts at position i in the origina
# l string S, then we will replace that occurrence of x with y.  If not, we do not
# hing.
# Using another example on S = "abcd", if we have both the replacement operation i
#  = 0, x = "ab", y = "eee", as well as another replacement operation i = 2, x = "
# ec", y = "ffff", this second operation does nothing because in the original stri
# ng S[2] = 'c', which doesn't match x[0] = 'e'.
# Notes:
# All these operations occur simultaneously.  It's guaranteed that there won't be
# any overlap in replacement: for example, S = "abc", indexes = [0, 1], sources =
# ["ab","bc"] is not a valid test case.
# For example, if we have S = "abcd" and we have some replacement operation i = 2,
#  x = "cd", y = "ffff", then because "cd" starts at position 2 in the original st
# ring S, we will replace it with "ffff".

# NOTE
# 0 <= indexes.length = sources.length = targets.length <= 100
# 0 < indexes[i] < S.length <= 1000
# All characters in given inputs are lowercase letters.

# EXAMPLE
# Input: S = "abcd", indexes = [0,2], sources = ["a","cd"], targets = ["eee","ffff
# "]
# Output: "eeebffff"
# Explanation: "a" starts at index 0 in S, so it's replaced
# by "eee".
# "cd" starts at index 2 in S, so it's replaced by "ffff".
# Input: S = "abcd", indexes = [0,2], sources = ["ab","ec"], targets = ["eee","fff
# f"]
# Output: "eeecd"
# Explanation: "ab" starts at index 0 in S, so it's replaced b
# y "eee".
# "ec" doesn't starts at index 2 in the original S, so we do nothing.

# Time:  O(n + m), m is the number of targets
# Space: O(n)

class Solution(object):
    def findReplaceString(self, S, indexes, sources, targets):
        """
        :type S: str
        :type indexes: List[int]
        :type sources: List[str]
        :type targets: List[str]
        :rtype: str
        """
        S = list(S)
        bucket = [None] * len(S)
        for i in xrange(len(indexes)):
            if all(indexes[i]+k < len(S) and
                   S[indexes[i]+k] == sources[i][k]
                   for k in xrange(len(sources[i]))):
```

```
                bucket[indexes[i]] = (len(sources[i]), list(targets[i]))
        result = []
        last = 0
        for i in xrange(len(S)):
            if bucket[i]:
                result.extend(bucket[i][1])
                last = i + bucket[i][0]
            elif i >= last:
                result.append(S[i])
        return "".join(result)


# Time:  O(mlogm + m * n)
# Space: O(n + m)
class Solution2(object):
    def findReplaceString(self, S, indexes, sources, targets):
        """
        :type S: str
        :type indexes: List[int]
        :type sources: List[str]
        :type targets: List[str]
        :rtype: str
        """
        for i, s, t in sorted(zip(indexes, sources, targets), reverse=True):
            if S[i:i+len(s)] == s:
                S = S[:i] + t + S[i+len(s):]

        return S
```

nim-game.py

```
# DESC
# Example:
# You are playing the following Nim Game with your friend: There is a heap of ston
# es on the table, each time one of you take turns to remove 1 to 3 stones. The on
# e who removes the last stone will be the winner. You will take the first turn to
#  remove the stones.
# Both of you are very clever and have optimal strategies for the game. Write a fu
# nction to determine whether you can win the game given the number of stones in t
# he heap.

# NOTE
#

# EXAMPLE
# Input: 4
# Output: false
# Explanation: If there are 4 stones in the heap, then you
#  will never win the game;
#              No matter 1, 2, or 3 stones you remove, t
# he last stone will always be
#              removed by your friend.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def canWinNim(self, n):
        """
        :type n: int
        :rtype: bool
        """
        return n % 4 != 0
```

number-of-valid-words-for-each-puzzle.py

```
# number-of-valid-words-for-each-puzzle is not found.
# Time:  O(n*l + m*L), m is the number of puzzles, L is the length of puzzles
#                    , n is the number of words, l is the max length of words
# Space: O(L!)

class Solution(object):
    def findNumOfValidWords(self, words, puzzles):
        """
        :type words: List[str]
        :type puzzles: List[str]
        :rtype: List[int]
        """
        L = 7
        def search(node, puzzle, start, first, met_first):
            result = 0
            if "_end" in node and met_first:
                result += node["_end"];
            for i in xrange(start, len(puzzle)):
                if puzzle[i] not in node:
                    continue
                result += search(node[puzzle[i]], puzzle, i+1,
                                 first, met_first or (puzzle[i] == first))
            return result

        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        for word in words:
            count = set(word)
            if len(count) > L:
                continue
            word = sorted(count)
            end = reduce(dict.__getitem__, word, trie)
            end["_end"] = end["_end"]+1 if "_end" in end else 1
        result = []
        for puzzle in puzzles:
            first = puzzle[0]
            result.append(search(trie, sorted(puzzle), 0, first, False))
        return result


# Time:  O(m*2^(L-1) + n*(l+m)), m is the number of puzzles, L is the length of puzzles
#                              , n is the number of words, l is the max length of words
# Space: O(m*2^(L-1))
import collections


class Solution2(object):
    def findNumOfValidWords(self, words, puzzles):
        """
        :type words: List[str]
        :type puzzles: List[str]
        :rtype: List[int]
        """
        L = 7
        lookup = collections.defaultdict(list)
        for i in xrange(len(puzzles)):
            bits = []
            base = 1 << (ord(puzzles[i][0])-ord('a'))
```

```
    for j in xrange(1, L):
        bits.append(ord(puzzles[i][j])-ord('a'))
    for k in xrange(2**len(bits)):
        bitset = base
        for j in xrange(len(bits)):
            if k & (1<<j):
                bitset |= 1<<bits[j]
        lookup[bitset].append(i)
result = [0]*len(puzzles)
for word in words:
    bitset = 0
    for c in word:
        bitset |= 1<<(ord(c)-ord('a'))
    if bitset not in lookup:
        continue
    for i in lookup[bitset]:
        result[i] += 1
return result
```

water-bottles.py

```python
# water-bottles is not found.
# Time:  O(logn/logm), n is numBottles, m is numExchange
# Space: O(1)

class Solution(object):
    def numWaterBottles(self, numBottles, numExchange):
        """
        :type numBottles: int
        :type numExchange: int
        :rtype: int
        """
        result = numBottles
        while numBottles >= numExchange:
            numBottles, remainder = divmod(numBottles, numExchange)
            result += numBottles
            numBottles += remainder
        return result
```

stream-of-characters2.py

```
# stream-of-characters2 is not found.
# Time:  ctor:  O(n + p^2), n is the total size of patterns
#                        , p is the number of patterns
#        query: O(m + z), m is the total size of query string
#                       , z is the number of all matched strings
#                       , query time would be O(m) if we don't use all the matched patterns
# Space: O(t + p^2), t is the total size of ac automata trie
#                  , space could be further improved by DAT (double-array trie)

# Aho-Corasick automata
# reference:
# - http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/02/Small02.pdf
# - http://algo.pw/algo/64/python

import collections


class AhoNode(object):
    def __init__(self):
        self.children = collections.defaultdict(AhoNode)
        self.suffix = None
        self.outputs = []


class AhoTrie(object):

    def step(self, letter):
        while self.__node and letter not in self.__node.children:
            self.__node = self.__node.suffix
        self.__node = self.__node.children[letter] if self.__node else self.__root
        return self.__node.outputs  # Time:  O(z), it would be O(m) if we don't use all the matched patterns

    def __init__(self, patterns):
        self.__root = self.__create_ac_trie(patterns)
        self.__node = self.__create_ac_suffix_and_output_links(self.__root)

    def __create_ac_trie(self, patterns):  # Time:  O(n), Space: O(t)
        root = AhoNode()
        for i, pattern in enumerate(patterns):
            node = root
            for c in pattern:
                node = node.children[c]
            node.outputs.append(i)
        return root

    def __create_ac_suffix_and_output_links(self, root):  # Time:  O(n + p^2), Space: O(t + p^2)
        queue = collections.deque()
        for node in root.children.itervalues():
            queue.append(node)
            node.suffix = root

        while queue:
            node = queue.popleft()
            for c, child in node.children.iteritems():
                queue.append(child)
                suffix = node.suffix
                while suffix and c not in suffix.children:
                    suffix = suffix.suffix
```

```
                child.suffix = suffix.children[c] if suffix else root
                child.outputs += child.suffix.outputs  # Time: O(p^2)

        return root


class StreamChecker(object):

    def __init__(self, words):
        """
        :type words: List[str]
        """
        self.__trie = AhoTrie(words)

    def query(self, letter):  # O(m) times
        """
        :type letter: str
        :rtype: bool
        """
        return len(self.__trie.step(letter)) > 0


# Your StreamChecker object will be instantiated and called as such:
# obj = StreamChecker(words)
# param_1 = obj.query(letter)
```

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

```python
# DESC
# Return the following binary tree:
# Given inorder and postorder traversal of a tree, construct the binary tree.
# For example, given
# Note:
#
# You may assume that duplicates do not exist in the tree.

# NOTE
#

# EXAMPLE
# inorder = [9,3,15,20,7]
# postorder = [9,15,7,20,3]
# 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
```

strange-printer.py

```
# DESC
# There is a strange printer with the following two special requirements:
# Given a string consists of lower English letters only, your job is to count the
# minimum number of turns the printer needed in order to print it.
# Example 1:
# Hint: Length of the given string will not exceed 100.
# Example 2:

# NOTE
# The printer can only print a sequence of the same character each time.
# At each turn, the printer can print new characters starting from and ending at a
# ny places, and will cover the original existing characters.

# EXAMPLE
# Input: "aba"
# Output: 2
# Explanation: Print "aaa" first and then print "b" from th
# e second place of the string, which will cover the existing character 'a'.
# Input: "aaabbb"
# Output: 2
# Explanation: Print "aaa" first and then print "bbb".

# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def strangePrinter(self, s):
        """
        :type s: str
        :rtype: int
        """
        def dp(s, i, j, lookup):
            if i > j:
                return 0
            if (i, j) not in lookup:
                lookup[(i, j)]  = dp(s, i, j-1, lookup) + 1
                for k in xrange(i, j):
                    if s[k] == s[j]:
                        lookup[(i, j)] = min(lookup[(i, j)], \
                                             dp(s, i, k, lookup) + dp(s, k+1, j-1, lookup))
            return lookup[(i, j)]

        lookup = {}
        return dp(s, 0, len(s)-1, lookup)
```

confusing-number.py

```
# confusing-number is not found.
# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def confusingNumber(self, N):
        """
        :type N: int
        :rtype: bool
        """
        lookup = {"0":"0", "1":"1", "6":"9", "8":"8", "9":"6"}

        S = str(N)
        result = []
        for i in xrange(len(S)):
            if S[i] not in lookup:
                return False
        for i in xrange((len(S)+1)//2):
            if S[i] != lookup[S[-(i+1)]]:
                return True
        return False
```

number-of-paths-with-max-score.py

```python
# DESC
# Example 3:
# In case there is no path, return [0, 0].
# You need to reach the top left square marked with the character 'E'. The rest of
#  the squares are labeled either with a numeric character 1, 2, ..., 9 or with an
#  obstacle 'X'. In one move you can go up, left or up-left (diagonally) only if t
# here is no obstacle there.
# Constraints:
# Example 2:
# Example 1:
# Return a list of two integers: the first integer is the maximum sum of numeric c
# haracters you can collect, and the second is the number of such paths that you c
# an take to get that maximum sum, taken modulo 10^9 + 7.
# You are given a square board of characters. You can move on the board starting a
# t the bottom right square marked with the character 'S'.

# NOTE
# 2 <= board.length == board[i].length <= 100

# EXAMPLE
# Input: board = ["E23","2X2","12S"]
# Output: [7,1]
# Input: board = ["E11","XXX","11S"]
# Output: [0,0]
# Input: board = ["E12","1X1","21S"]
# Output: [4,2]

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

class Solution(object):
    def pathsWithMaxScore(self, board):
        """
        :type board: List[str]
        :rtype: List[int]
        """
        MOD = 10**9+7
        directions = [[1, 0], [0, 1], [1, 1]]
        dp = [[[0, 0] for r in xrange(len(board[0])+1)]
              for r in xrange(2)]
        dp[(len(board)-1)%2][len(board[0])-1] = [0, 1]
        for r in reversed(xrange(len(board))):
            for c in reversed(xrange(len(board[0]))):
                if board[r][c] in "XS":
                    continue
                dp[r%2][c] = [0, 0]
                for dr, dc in directions:
                    if dp[r%2][c][0] < dp[(r+dr)%2][c+dc][0]:
                        dp[r%2][c] = dp[(r+dr)%2][c+dc][:]
                    elif dp[r%2][c][0] == dp[(r+dr)%2][c+dc][0]:
                        dp[r%2][c][1] = (dp[r%2][c][1]+dp[(r+dr)%2][c+dc][1]) % MOD
                if dp[r%2][c][1] and board[r][c] != 'E':
                    dp[r%2][c][0] += int(board[r][c])
        return dp[0][0]
```

race-car.py

```
# DESC
# When you get an instruction "A", your car does the following: position += speed,
#  speed *= 2.
# For example, after commands "AAR", your car goes to positions 0->1->3->3, and yo
# ur speed goes to 1->2->4->-1.
# Now for some target position, say the length of the shortest sequence of instruc
# tions to get there.
# Your car drives automatically according to a sequence of instructions A (acceler
# ate) and R (reverse).
# Your car starts at position 0 and speed +1 on an infinite number line.  (Your ca
# r can go into negative positions.)
# Note:
# When you get an instruction "R", your car does the following: if your speed is p
# ositive then speed = -1 , otherwise speed = 1.  (Your position stays the same.)

# NOTE
# 1 <= target <= 10000.

# EXAMPLE
# Example 1:
# Input:
# target = 3
# Output: 2
# Explanation:
# The shortest instruction s
# equence is "AA".
# Your position goes from 0->1->3.
# Example 2:
# Input:
# target = 6
# Output: 5
# Explanation:
# The shortest instruction s
# equence is "AAARA".
# Your position goes from 0->1->3->7->7->6.

# Time : O(nlogn), n is the value of the target
# Space: O(n)


class Solution(object):
    def racecar(self, target):
        dp = [0] * (target+1)
        for i in xrange(1, target+1):
            # 2^(k-1) <= i < 2^k
            k = i.bit_length()

            # case 1. drive exactly i at best
            #         seq(i) = A^k
            if i == 2**k-1:
                dp[i] = k
                continue

            # case 2. drive cross i at 2^k-1, and turn back to i
            #         seq(i) = A^k -> R -> seq(2^k-1 - i)
            dp[i] = k+1 + dp[2**k-1 - i]

            # case 3. drive less then 2^k-1, and turn back some distance,
```

```
        #         and turn back again to make the direction is the same
        #         seq(i) = shortest(seq(i), A^(k-1) -> R -> A^j -> R ->
        #                                   seq(i - (2^(k-1)-1) + (2^j-1)),
        #                   where 0 <= j < k-1)
        #         => dp[i] = min(dp[i], (k-1) + 1 + j + 1 +
        #                               dp[i - (2**(k-1)-1) + (2**j-1)])
        for j in xrange(k-1):
            dp[i] = min(dp[i], k+j+1 + dp[i - 2**(k-1) + 2**j])

    return dp[-1]
```

implement-strstr.py

```
# implement-strstr is not found.
# Time:  O(n + k)
# Space: O(k)

class Solution(object):
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """
        if not needle:
            return 0

        return self.KMP(haystack, needle)

    def KMP(self, text, pattern):
        prefix = self.getPrefix(pattern)
        j = -1
        for i in xrange(len(text)):
            while j > -1 and pattern[j + 1] != text[i]:
                j = prefix[j]
            if pattern[j + 1] == text[i]:
                j += 1
            if j == len(pattern) - 1:
                return i - j
        return -1

    def getPrefix(self, pattern):
        prefix = [-1] * len(pattern)
        j = -1
        for i in xrange(1, len(pattern)):
            while j > -1 and pattern[j + 1] != pattern[i]:
                j = prefix[j]
            if pattern[j + 1] == pattern[i]:
                j += 1
            prefix[i] = j
        return prefix


# Time:  O(n * k)
# Space: O(k)
class Solution2(object):
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """
        for i in xrange(len(haystack) - len(needle) + 1):
            if haystack[i : i + len(needle)] == needle:
                return i
        return -1
```

teemo-attacking.py

```
# DESC
# 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:
# Example 2:
# 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 s
# eries towards Ashe and the poisoning time duration per Teemo's attacking, you ne
# ed to output the total time that Ashe is in poisoned condition.
# Note:

# NOTE
# You may assume the numbers in the Teemo's attacking time series and his poisonin
# g time duration per attacking are non-negative integers, which won't exceed 10,0
# 00,000.
# You may assume the length of given time series array won't exceed 10000.

# EXAMPLE
# Input: [1,4], 2
# Output: 4
# Explanation: At time point 1, Teemo starts attacking A
# she and makes Ashe be poisoned immediately.
# This poisoned status will last 2 se
# conds until the end of time point 2.
# And at time point 4, Teemo attacks Ashe ag
# ain, and causes Ashe to be in poisoned status for another 2 seconds.
# So you fin
# ally need to output 4.
# Input: [1,2], 2
# Output: 3
# Explanation: At time point 1, Teemo starts attacking A
# she 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 attac
# ks 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 po
# int 2, it will stop at the end of time point 3.
# So you finally need to output 3
# .

# 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
```

set-mismatch.py

```
# DESC
# The set S originally contains numbers from 1 to n. But unfortunately, due to the
#  data error, one of the numbers in the set got duplicated to another number in t
# he set, which results in repetition of one number and loss of another number.
# Note:
# Example 1:
# Given an array nums representing the data status of this set after the error. Yo
# ur task is to firstly find the number occurs twice and then find the number that
#  is missing. Return them in the form of an array.

# NOTE
# The given array size will in the range [2, 10000].
# The given array's numbers won't have any order.

# EXAMPLE
# Input: nums = [1,2,2,4]
# Output: [2,3]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findErrorNums(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        x_xor_y = 0
        for i in xrange(len(nums)):
            x_xor_y ^= nums[i] ^ (i+1)
        bit =  x_xor_y & ~(x_xor_y-1)
        result = [0] * 2
        for i, num in enumerate(nums):
            result[bool(num & bit)] ^= num
            result[bool((i+1) & bit)] ^= i+1
        if result[0] not in nums:
            result[0], result[1] = result[1], result[0]
        return result


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def findErrorNums(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        result = [0] * 2
        for i in nums:
            if nums[abs(i)-1] < 0:
                result[0] = abs(i)
            else:
                nums[abs(i)-1] *= -1
        for i in xrange(len(nums)):
            if nums[i] > 0:
                result[1] = i+1
            else:
```

```
                nums[i] *= -1
        return result


# Time:  O(n)
# Space: O(1)
class Solution3(object):
    def findErrorNums(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        N = len(nums)
        x_minus_y = sum(nums) - N*(N+1)//2
        x_plus_y = (sum(x*x for x in nums) - N*(N+1)*(2*N+1)/6) // x_minus_y
        return (x_plus_y+x_minus_y) // 2, (x_plus_y-x_minus_y) // 2
```

minimum-cost-to-cut-a-stick.py

```
# minimum-cost-to-cut-a-stick is not found.
# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def minCost(self, n, cuts):
        """
        :type n: int
        :type cuts: List[int]
        :rtype: int
        """
        sorted_cuts = sorted(cuts + [0, n])
        dp = [[0]*len(sorted_cuts) for _ in xrange(len(sorted_cuts))]
        for l in xrange(2, len(sorted_cuts)):
            for i in xrange(len(sorted_cuts)-l):
                dp[i][i+l] = min(dp[i][j]+dp[j][i+l] for j in xrange(i+1, i+l)) + \
                             sorted_cuts[i+l]-sorted_cuts[i]
        return dp[0][len(sorted_cuts)-1]
```

number-of-connected-components-in-an-undirected-graph.py

```python
# number-of-connected-components-in-an-undirected-graph is not found.
# Time:  O(nlog*n) ~= O(n), n is the length of the positions
# Space: O(n)

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


class Solution(object):
    def countComponents(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: int
        """
        union_find = UnionFind(n)
        for i, j in edges:
            union_find.union_set(i, j)
        return union_find.count
```

closest-leaf-in-a-binary-tree.py

```python
# closest-leaf-in-a-binary-tree is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findClosestLeaf(self, root, k):
        """
        :type root: TreeNode
        :type k: int
        :rtype: int
        """
        def traverse(node, neighbors, leaves):
            if not node:
                return
            if not node.left and not node.right:
                leaves.add(node.val)
                return
            if node.left:
                neighbors[node.val].append(node.left.val)
                neighbors[node.left.val].append(node.val)
                traverse(node.left, neighbors, leaves)
            if node.right:
                neighbors[node.val].append(node.right.val)
                neighbors[node.right.val].append(node.val)
                traverse(node.right, neighbors, leaves)

        neighbors, leaves = collections.defaultdict(list), set()
        traverse(root, neighbors, leaves)
        q, lookup = [k], set([k])
        while q:
            next_q = []
            for u in q:
                if u in leaves:
                    return u
                for v in neighbors[u]:
                    if v in lookup:
                        continue
                    lookup.add(v)
                    next_q.append(v)
            q = next_q
        return 0
```

largest-divisible-subset.py

```
# DESC
# If there are multiple solutions, return any subset is fine.
# Given a set of distinct positive integers, find the largest subset such that eve
# ry pair (Si, Sj) of elements in this subset satisfies:
# Example 1:
# Si % Sj = 0 or Sj % Si = 0.
# Example 2:

# NOTE
#

# EXAMPLE
# Input: [1,2,3]
# Output: [1,2] (of course, [1,3] will also be ok)
# 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]
```

find-the-longest-substring-containing-vowels-in-even-counts.py

```python
# find-the-longest-substring-containing-vowels-in-even-counts is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findTheLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        VOWELS = "aeiou"
        result, mask, lookup = 0, 0, [len(s)]*(2**len(VOWELS))
        lookup[0] = -1
        for i, c in enumerate(s):
            index = VOWELS.find(c)
            mask ^= (1 << index) if index >= 0 else 0
            if lookup[mask] == len(s):
                lookup[mask] = i
            result = max(result, i-lookup[mask])
        return result
```

word-subsets.py

```
# DESC
# Now say a word a from A is universal if for every b in B, b is a subset of a.
# Example 3:
# Example 4:
# Example 1:
# We are given two arrays A and B of words.  Each word is a string of lowercase le
# tters.
# Example 2:
# Now, say that word b is a subset of word a if every letter in b occurs in a, inc
# luding multiplicity.  For example, "wrr" is a subset of "warrior", but is not a
# subset of "world".
# Example 5:
# Return a list of all universal words in A.  You can return the words in any order.
# Note:

# NOTE
# 1 <= A.length, B.length <= 10000
# A[i] and B[i] consist only of lowercase letters.
# 1 <= A[i].length, B[i].length <= 10
# All words in A[i] are unique: there isn't i != j with A[i] == A[j].

# EXAMPLE
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["lo","eo"]
# Ou
# tput: ["google","leetcode"]
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["l","e"]
# Outp
# ut: ["apple","google","leetcode"]
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","o"]
# Outp
# ut: ["facebook","google","leetcode"]
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["ec","oc","ce
# o"]
# Output: ["facebook","leetcode"]
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","oo"]
# Out
# put: ["facebook","google"]

# 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
# DESC
# Example 2:
# Return true if and only if it is possible to assign integers to variable names s
# o as to satisfy all the given equations.
# Example 3:
# Example 5:
# Example 1:
# Example 4:
# Given an array equations of strings that represent relationships between variabl
# es, each string equations[i] has length 4 and takes one of two different forms:
# "a==b" or "a!=b".  Here, a and b are lowercase letters (not necessarily differen
# t) that represent one-letter variable names.
# Note:

# NOTE
# equations[i][1] is either '=' or '!'
# equations[i][0] and equations[i][3] are lowercase letters
# equations[i][2] is '='
# equations[i].length == 4
# 1 <= equations.length <= 500

# EXAMPLE
# Input: ["b==a","a==b"]
# Output: true
# Explanation: We could assign a = 1 and b = 1
#  to satisfy both equations.
# Input: ["a==b","b!=a"]
# Output: false
# Explanation: If we assign say, a = 1 and b
# = 1, then the first equation is satisfied, but not the second.  There is no way
# to assign the variables to satisfy both equations.
# Input: ["a==b","b==c","a==c"]
# Output: true
# Input: ["a==b","b!=c","c==a"]
# Output: false
# Input: ["c==c","b==d","x!=z"]
# Output: true

# 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):
    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
```

valid-palindrome-ii.py

```python
# DESC
# Example 1:
# Example 2:
# Note:
# Given a non-empty string s, you may delete at most one character.  Judge whether
#  you can make it a palindrome.

# NOTE
# The string will only contain lowercase characters a-z.
# The maximum length of the
#  string is 50000.

# EXAMPLE
# Input: "abca"
# Output: True
# Explanation: You could delete the character 'c'.
# Input: "aba"
# Output: True

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def validPalindrome(self, s):
        """
        :type s: str
        :rtype: bool
        """
        def validPalindrome(s, left, right):
            while left < right:
                if s[left] != s[right]:
                    return False
                left, right = left+1, right-1
            return True

        left, right = 0, len(s)-1
        while left < right:
            if s[left] != s[right]:
                return validPalindrome(s, left, right-1) or validPalindrome(s, left+1, right)
            left, right = left+1, right-1
        return True
```

transpose-matrix.py

```
# DESC
# The transpose of a matrix is the matrix flipped over it's main diagonal, switchi
# ng the row and column indices of the matrix.
# Given a matrix A, return the transpose of A.
# Note:
# Example 2:
# Example 1:

# NOTE
# 1 <= A.length <= 1000
# 1 <= A[0].length <= 1000

# EXAMPLE
# Input: [[1,2,3],[4,5,6]]
# Output: [[1,4],[2,5],[3,6]]
# Input: [[1,2,3],[4,5,6],[7,8,9]]
# Output: [[1,4,7],[2,5,8],[3,6,9]]

# Time:  O(r * c)
# Space: O(1)


class Solution(object):
    def transpose(self, A):
        """
        :type A: List[List[int]]
        :rtype: List[List[int]]
        """
        result = [[None] * len(A) for _ in xrange(len(A[0]))]
        for r, row in enumerate(A):
            for c, val in enumerate(row):
                result[c][r] = val
        return result


# Time:  O(r * c)
# Space: O(1)
class Solution2(object):
    def transpose(self, A):
        """
        :type A: List[List[int]]
        :rtype: List[List[int]]
        """
        return zip(*A)
```

the-k-strongest-values-in-an-array.py

```
# the-k-strongest-values-in-an-arra is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def getStrongest(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: List[int]
        """
        arr.sort()
        m = arr[(len(arr)-1)//2]
        result = []
        left, right = 0, len(arr)-1
        while len(result) < k:
            if m-arr[left] > arr[right]-m:
                result.append(arr[left])
                left += 1
            else:
                result.append(arr[right])
                right -= 1
        return result


# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def getStrongest(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: List[int]
        """
        arr.sort()
        m = arr[(len(arr)-1)//2]
        arr.sort(key=lambda x: (-abs(x-m), -x))
        return arr[:k]


# Time:  O(n)
# Space: O(1)
import random


class Solution_TLE(object):
    def getStrongest(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: List[int]
        """
        def nth_element(nums, n, compare=lambda a, b: a < b):
            def partition_around_pivot(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:
        pivot_idx = random.randint(left, right)
        new_pivot_idx = partition_around_pivot(left, right, pivot_idx, nums, compare)
        if new_pivot_idx == n:
            return
        elif new_pivot_idx > n:
            right = new_pivot_idx - 1
        else:  # new_pivot_idx < n
            left = new_pivot_idx + 1

nth_element(arr, (len(arr)-1)//2)
m = arr[(len(arr)-1)//2]
nth_element(arr, k, lambda a, b: abs(a-m) > abs(b-m) if abs(a-m) != abs(b-m) else a > b)
return arr[:k]
```

campus-bikes.py

```
# campus-bikes is not found.
# Time:  O((w * b) * log(w * b))
# Space: O(w * b)

import heapq


class Solution(object):
    def assignBikes(self, workers, bikes):
        """
        :type workers: List[List[int]]
        :type bikes: List[List[int]]
        :rtype: List[int]
        """
        def manhattan(p1, p2):
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        distances = [[] for _ in xrange(len(workers))]
        for i in xrange(len(workers)):
            for j in xrange(len(bikes)):
                distances[i].append((manhattan(workers[i], bikes[j]), i, j))
            distances[i].sort(reverse = True)

        result = [None] * len(workers)
        lookup = set()
        min_heap = []
        for i in xrange(len(workers)):
            heapq.heappush(min_heap, distances[i].pop())
        while len(lookup) < len(workers):
            _, worker, bike = heapq.heappop(min_heap)
            if bike not in lookup:
                result[worker] = bike
                lookup.add(bike)
            else:
                heapq.heappush(min_heap, distances[worker].pop())
        return result
```

range-sum-query-mutable.py

```
# range-sum-query-mutable is not found.
# Time:  ctor:   O(n),
#        update: O(logn),
#        query:  O(logn)
# Space: O(n)

class NumArray(object):
    def __init__(self, nums):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        if not nums:
            return
        self.__nums = nums
        self.__bit = [0] * (len(self.__nums) + 1)
        for i in xrange(1, len(self.__bit)):
            self.__bit[i] = nums[i-1] + self.__bit[i-1]

        for i in reversed(xrange(1, len(self.__bit))):
            last_i = i - (i & -i)
            self.__bit[i] -= self.__bit[last_i]

    def update(self, i, val):
        """
        :type i: int
        :type val: int
        :rtype: int
        """
        if val - self.__nums[i]:
            self.__add(i, val - self.__nums[i])
            self.__nums[i] = val

    def sumRange(self, i, j):
        """
        sum of elements nums[i..j], inclusive.
        :type i: int
        :type j: int
        :rtype: int
        """
        return self.__sum(j) - self.__sum(i-1)

    def __sum(self, i):
        i += 1
        ret = 0
        while i > 0:
            ret += self.__bit[i]
            i -= (i & -i)
        return ret

    def __add(self, i, val):
        i += 1
        while i <= len(self.__nums):
            self.__bit[i] += val
            i += (i & -i)


# Time:  ctor:   O(n),
```

```
#        update: O(logn),
#        query:  O(logn)
# Space: O(n)
# Segment Tree solution.
class NumArray2(object):
    def __init__(self, nums,
                 query_fn=lambda x, y: x+y,
                 update_fn=lambda x, y: y,
                 default_val=0):
        """
        initialize your data structure here.
        :type nums: List[int]
        """
        N = len(nums)
        self.__original_length = N
        self.__tree_length = 2**(N.bit_length() + (N&(N-1) != 0))-1
        self.__query_fn = query_fn
        self.__update_fn = update_fn
        self.__default_val = default_val
        self.__tree = [default_val for _ in range(self.__tree_length)]
        self.__lazy = [None for _ in range(self.__tree_length)]
        self.__constructTree(nums, 0, self.__original_length-1, 0)

    def update(self, i, val):
        self.__updateTree(val, i, i, 0, self.__original_length-1, 0)

    def sumRange(self, i, j):
        return self.__queryRange(i, j, 0, self.__original_length-1, 0)

    def __constructTree(self, nums, left, right, idx):
        if left > right:
            return
        if left == right:
            self.__tree[idx] = self.__update_fn(self.__tree[idx], nums[left])
            return
        mid = left + (right-left)//2
        self.__constructTree(nums, left, mid, idx*2 + 1)
        self.__constructTree(nums, mid+1, right, idx*2 + 2)
        self.__tree[idx] = self.__query_fn(self.__tree[idx*2 + 1], self.__tree[idx*2 + 2])

    def __apply(self, left, right, idx, val):
        self.__tree[idx] = self.__update_fn(self.__tree[idx], val)
        if left != right:
            self.__lazy[idx*2 + 1] = self.__update_fn(self.__lazy[idx*2 + 1], val)
            self.__lazy[idx*2 + 2] = self.__update_fn(self.__lazy[idx*2 + 2], val)

    def __updateTree(self, val, range_left, range_right, left, right, idx):
        if left > right:
            return
        if self.__lazy[idx] is not None:
            self.__apply(left, right, idx, self.__lazy[idx])
            self.__lazy[idx] = None
        if range_left > right or range_right < left:
            return
        if range_left <= left and right <= range_right:
            self.__apply(left, right, idx, val)
            return
        mid = left + (right-left)//2
        self.__updateTree(val, range_left, range_right, left, mid, idx*2 + 1)
        self.__updateTree(val, range_left, range_right, mid+1, right, idx*2 + 2)
```

```
        self.__tree[idx] = self.__query_fn(self.__tree[idx*2 + 1],
                                           self.__tree[idx*2 + 2])

    def __queryRange(self, range_left, range_right, left, right, idx):
        if left > right:
            return self.__default_val
        if self.__lazy[idx] is not None:
            self.__apply(left, right, idx, self.__lazy[idx])
            self.__lazy[idx] = None
        if right < range_left or left > range_right:
            return self.__default_val
        if range_left <= left and right <= range_right:
            return self.__tree[idx]
        mid = left + (right-left)//2
        return self.__query_fn(self.__queryRange(range_left, range_right, left, mid, idx*2 + 1),
                               self.__queryRange(range_left, range_right, mid + 1, right, idx*2 + 2))
```

house-robber-ii.py

```
# DESC
# Example 2:
# You are a professional robber planning to rob houses along a street. Each house
# has a certain amount of money stashed. All houses at this place are arranged in
# a circle. That means the first house is the neighbor of the last one. Meanwhile,
#  adjacent houses have security system connected and it will automatically contac
# t the police if two adjacent houses were broken into on the same night.
# Given a list of non-negative integers representing the amount of money of each h
# ouse, determine the maximum amount of money you can rob tonight without alerting
#  the police.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: [1,2,3,1]
# Output: 4
# Explanation: Rob house 1 (money = 1) and then rob hou
# se 3 (money = 3).
#              Total amount you can rob = 1 + 3 = 4.
# Input: [2,3,2]
# Output: 3
# Explanation: You cannot rob house 1 (money = 2) and the
# n rob house 3 (money = 2),
#              because they are adjacent houses.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {integer[]} nums
    # @return {integer}
    def rob(self, nums):
        if len(nums) == 0:
            return 0

        if len(nums) == 1:
            return nums[0]

        return max(self.robRange(nums, 0, len(nums) - 1),\
                   self.robRange(nums, 1, len(nums)))

    def robRange(self, nums, start, end):
        num_i, num_i_1 = nums[start], 0
        for i in xrange(start + 1, end):
            num_i_1, num_i_2 = num_i, num_i_1
            num_i = max(nums[i] + num_i_2, num_i_1)

        return num_i
```

reverse-words-in-a-string-ii.py

```
# reverse-words-in-a-string-ii is not found.
# Time: O(n)
# Space:O(1)

class Solution(object):
    def reverseWords(self, s):
        """
        :type s: a list of 1 length strings (List[str])
        :rtype: nothing
        """
        def reverse(s, begin, end):
            for i in xrange((end - begin) / 2):
                s[begin + i], s[end - 1 - i] = s[end - 1 - i], s[begin + i]

        reverse(s, 0, len(s))
        i = 0
        for j in xrange(len(s) + 1):
            if j == len(s) or s[j] == ' ':
                reverse(s, i, j)
                i = j + 1
```

maximum-score-words-formed-by-letters.py

```python
# maximum-score-words-formed-by-letters is not found.
# Time:  O(n * 2^n)
# Space: O(n)

import collections


class Solution(object):
    def maxScoreWords(self, words, letters, score):
        """
        :type words: List[str]
        :type letters: List[str]
        :type score: List[int]
        :rtype: int
        """
        def backtracking(words, word_scores, word_counts, curr, curr_score, letter_count, result):
            result[0] = max(result[0], curr_score)
            for i in xrange(curr, len(words)):
                if any(letter_count[c] < word_counts[i][c] for c in word_counts[i]):
                    continue
                backtracking(words, word_scores, word_counts, i+1,
                             curr_score+word_scores[i], letter_count-word_counts[i],
                             result)

        letter_count = collections.Counter(letters)
        word_counts = map(collections.Counter, words)
        word_scores = [sum(score[ord(c)-ord('a')] for c in words[i])
                       for i in xrange(len(words))]
        result = [0]
        backtracking(words, word_scores, word_counts, 0, 0, letter_count, result)
        return result[0]
```

delete-n-nodes-after-m-nodes-of-a-linked-list.py

```python
# delete-n-nodes-after-m-nodes-of-a-linked-list is not found.
# Time:  O(n)
# Space: O(1)

# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next


class Solution(object):
    def deleteNodes(self, head, m, n):
        """
        :type head: ListNode
        :type m: int
        :type n: int
        :rtype: ListNode
        """
        head = dummy = ListNode(next=head)
        while head:
            for _ in xrange(m):
                if not head.next:
                    return dummy.next
                head = head.next
            prev = head
            for _ in xrange(n):
                if not head.next:
                    prev.next = None
                    return dummy.next
                head = head.next
            prev.next = head.next
        return dummy.next
```

design-phone-directory.py

```
# design-phone-director is not found.
# init:     Time: O(n), Space: O(n)
# get:      Time: O(1), Space: O(1)
# check:    Time: O(1), Space: O(1)
# release:  Time: O(1), Space: O(1)

class PhoneDirectory(object):

    def __init__(self, maxNumbers):
        """
        Initialize your data structure here
        @param maxNumbers - The maximum numbers that can be stored in the phone directory.
        :type maxNumbers: int
        """
        self.__curr = 0
        self.__numbers = range(maxNumbers)
        self.__used = [False] * maxNumbers


    def get(self):
        """
        Provide a number which is not assigned to anyone.
        @return - Return an available number. Return -1 if none is available.
        :rtype: int
        """
        if self.__curr == len(self.__numbers):
            return -1
        number = self.__numbers[self.__curr]
        self.__curr += 1
        self.__used[number] = True
        return number


    def check(self, number):
        """
        Check if a number is available or not.
        :type number: int
        :rtype: bool
        """
        return 0 <= number < len(self.__numbers) and \
               not self.__used[number]


    def release(self, number):
        """
        Recycle or release a number.
        :type number: int
        :rtype: void
        """
        if not 0 <= number < len(self.__numbers) or \
           not self.__used[number]:
            return
        self.__used[number] = False
        self.__curr -= 1
        self.__numbers[self.__curr] = number
```

minimum-distance-to-type-a-word-using-two-fingers.py

```
# minimum-distance-to-type-a-word-using-two-fingers is not found.
# Time:  O(26n)
# Space: O(26)

class Solution(object):
    def minimumDistance(self, word):
        """
        :type word: str
        :rtype: int
        """
        def distance(a, b):
            return abs(a//6 - b//6) + abs(a%6 - b%6)

        dp = [0]*26
        for i in xrange(len(word)-1):
            b, c = ord(word[i])-ord('A'), ord(word[i+1])-ord('A')
            dp[b] = max(dp[a] - distance(a, c) + distance(b, c) for a in xrange(26))
        return sum(distance(ord(word[i])-ord('A'), ord(word[i+1])-ord('A')) for i in xrange(len(word)-1)) - ma


# Time:  O(52n)
# Space: O(52)
class Solution2(object):
    def minimumDistance(self, word):
        """
        :type word: str
        :rtype: int
        """
        def distance(a, b):
            if -1 in [a, b]:
                return 0
            return abs(a//6 - b//6) + abs(a%6 - b%6)

        dp = {(-1, -1): 0}
        for c in word:
            c = ord(c)-ord('A')
            new_dp = {}
            for a, b in dp:
                new_dp[c, b] = min(new_dp.get((c, b), float("inf")), dp[a, b] + distance(a, c))
                new_dp[a, c] = min(new_dp.get((a, c), float("inf")), dp[a, b] + distance(b, c))
            dp = new_dp
        return min(dp.itervalues())
```

remove-duplicates-from-sorted-list.py

```
# DESC
# Example 1:
# Example 2:
# Given a sorted linked list, delete all duplicates such that each element appear
# only once.

# NOTE
#

# EXAMPLE
# Input: 1->1->2
# Output: 1->2
# Input: 1->1->2->3->3
# Output: 1->2->3

# Time:  O(n)
# Space: O(1)

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


class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        cur = head
        while cur:
            runner = cur.next
            while runner and runner.val == cur.val:
                runner = runner.next
            cur.next = runner
            cur = runner
        return head

    def deleteDuplicates2(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head: return head
        if head.next:
            if head.val == head.next.val:
                head = self.deleteDuplicates2(head.next)
            else:
                head.next = self.deleteDuplicates2(head.next)
        return head
```

unique-substrings-in-wraparound-string.py

```
# DESC
# Example 2:
# Consider the string s to be the infinite wraparound string of "abcdefghijklmnopq
# rstuvwxyz", so s will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijk
# lmnopqrstuvwxyzabcd....".
# Note: p consists of only lowercase English letters and the size of p might be ov
# er 10000.
# Example 1:
# Example 3:
# 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 stri
# ng s.

# NOTE
#

# EXAMPLE
# Input: "zab"
# Output: 6
# Explanation: There are six substrings "z", "a", "b", "za"
# , "ab", "zab" of string "zab" in the string s.
# Input: "cac"
# Output: 2
# Explanation: There are two substrings "a", "c" of string
# "cac" in the string s.
# Input: "a"
# Output: 1
#
# Explanation: Only the substring "a" of string "a" is in th
# e 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
```

search-for-a-range.py

```python
# search-for-a-range is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def searchRange(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        # Find the first idx where nums[idx] >= target
        left = self.binarySearch(lambda x, y: x >= y, nums, target)
        if left >= len(nums) or nums[left] != target:
            return [-1, -1]
        # Find the first idx where nums[idx] > target
        right = self.binarySearch(lambda x, y: x > y, nums, target)
        return [left, right - 1]

    def binarySearch(self, compare, nums, target):
        left, right = 0, len(nums)
        while left < right:
            mid = left + (right - left) / 2
            if compare(nums[mid], target):
                right = mid
            else:
                left = mid + 1
        return left

    def binarySearch2(self, compare, nums, target):
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = left + (right - left) / 2
            if compare(nums[mid], target):
                right = mid - 1
            else:
                left = mid + 1
        return left

    def binarySearch3(self, compare, nums, target):
        left, right = -1, len(nums)
        while left + 1 < right:
            mid = left + (right - left) / 2
            if compare(nums[mid], target):
                right = mid
            else:
                left = mid
        return left if left != -1 and compare(nums[left], target) else right
```

find-root-of-n-ary-tree.py

```python
# find-root-of-n-ary-tree is not found.
# Time:  O(n)
# Space: O(1)

# Definition for a Node.
class Node(object):
    def __init__(self, val=None, children=None):
        pass


class Solution(object):
    def findRoot(self, tree):
        """
        :type tree: List['Node']
        :rtype: 'Node'
        """
        root = 0
        for node in tree:
            root ^= id(node)
            for child in node.children:
                root ^= id(child)
        for node in tree:
            if id(node) == root:
                return node
        return None


class Solution2(object):
    def findRoot(self, tree):
        """
        :type tree: List['Node']
        :rtype: 'Node'
        """
        root = 0
        for node in tree:
            root ^= node.val
            for child in node.children:
                root ^= child.val
        for node in tree:
            if node.val == root:
                return node
        return None


class Solution3(object):
    def findRoot(self, tree):
        """
        :type tree: List['Node']
        :rtype: 'Node'
        """
        root = 0
        for node in tree:
            root += node.val-sum(child.val for child in node.children)
        for node in tree:
            if node.val == root:
                return node
        return None
```

shuffle-string.py

```
# shuffle-string is not found.
# Time:  O(n)
# Space: O(1)

# in-place solution
class Solution(object):
    def restoreString(self, s, indices):
        """
        :type s: str
        :type indices: List[int]
        :rtype: str
        """
        result = list(s)
        for i, c in enumerate(result):
            if indices[i] == i:
                continue
            move, j = c, indices[i]
            while j != i:
                result[j], move = move, result[j]
                indices[j], j = j, indices[j]
            result[i] = move
        return "".join(result)


# Time:  O(n)
# Space: O(1)
import itertools


class Solution2(object):
    def restoreString(self, s, indices):
        """
        :type s: str
        :type indices: List[int]
        :rtype: str
        """
        result = ['']*len(s)
        for i, c in itertools.izip(indices, s):
            result[i] = c
        return "".join(result)
```

scramble-string.py

```
# DESC
# For example, if we choose the node "gr" and swap its two children, it produces a
#  scrambled string "rgeat".
# Example 1:
# Similarly, if we continue to swap the children of nodes "eat" and "at", it produ
# ces a scrambled string "rgtae".
# We say that "rgeat" is a scrambled string of "great".
# We say that "rgtae" is a scrambled string of "great".
# Given two strings s1 and s2 of the same length, determine if s2 is a scrambled s
# tring of s1.
# Below is one possible representation of s1 = "great":
# Example 2:
# "great"
# Given a string s1, we may represent it as a binary tree by partitioning it to tw
# o non-empty substrings recursively.
# To scramble the string, we may choose any non-leaf node and swap its two children.

# NOTE
#

# EXAMPLE
# rgtae
#    /    \
#   rg    tae
#  / \    /  \
# r   g  ta  e
#       / \
#      t   a
# Input: s1 = "abcde", s2 = "caebd"
# Output: false
# rgeat
#    /    \
#   rg    eat
#  / \    /  \
# r   g  e   at
#           / \
#          a   t
# great
#    /    \
#   gr    eat
#  / \    /  \
# g   r  e   at
#           / \
#          a   t
# Input: s1 = "great", s2 = "rgeat"
# Output: true

# Time:  O(n^4)
# Space: O(n^3)

class Solution(object):
    # @return a boolean
    def isScramble(self, s1, s2):
        if not s1 or not s2 or len(s1) != len(s2):
            return False
        if s1 == s2:
            return True
        result = [[[False for j in xrange(len(s2))] for i in xrange(len(s1))] for n in xrange(len(s1) + 1)]
```

```
    for i in xrange(len(s1)):
        for j in xrange(len(s2)):
            if s1[i] == s2[j]:
                result[1][i][j] = True

    for n in xrange(2, len(s1) + 1):
        for i in xrange(len(s1) - n + 1):
            for j in xrange(len(s2) - n + 1):
                for k in xrange(1, n):
                    if result[k][i][j] and result[n - k][i + k][j + k] or\
                       result[k][i][j + n - k] and result[n - k][i + k][j]:
                        result[n][i][j] = True
                        break

    return result[n][0][0]
```

palindrome-partitioning-iii.py

```
# palindrome-partitioning-iii is not found.
# Time:  O(k * n^2)
# Space: O(n^2)

class Solution(object):
    def palindromePartition(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        # dp1[i][j]: minimum number of changes to make s[i, j] palindrome
        dp1 = [[0]*len(s) for _ in xrange(len(s))]
        for l in xrange(1, len(s)+1):
            for i in xrange(len(s)-l+1):
                j = i+l-1
                if i == j-1:
                    dp1[i][j] = 0 if s[i] == s[j] else 1
                elif i != j:
                    dp1[i][j] = dp1[i+1][j-1] if s[i] == s[j] else dp1[i+1][j-1]+1

        # dp2[d][i]: minimum number of changes to divide s[0, i] into d palindromes
        dp2 = [[float("inf")]*len(s) for _ in xrange(2)]
        dp2[1] = dp1[0][:]
        for d in xrange(2, k+1):
            dp2[d%2] = [float("inf")]*len(s)
            for i in xrange(d-1, len(s)):
                for j in xrange(d-2, i):
                    dp2[d%2][i] = min(dp2[d%2][i], dp2[(d-1)%2][j]+dp1[j+1][i])
        return dp2[k%2][len(s)-1]
```

next-greater-element-i.py

```
# DESC
# nums1
# The Next Greater Number of a number x in nums1 is the first greater number to it
# s right in nums2. If it does not exist, output -1 for this number.
# Note:
# Example 2:
# Example 1:
# You are given two arrays (without duplicates) nums1 and nums2 where nums1's elem
# ents are subset of nums2. Find all the next greater numbers for nums1's elements
#  in the corresponding places of nums2.

# NOTE
# All elements in nums1 and nums2 are unique.
# The length of both nums1 and nums2 would not exceed 1000.

# EXAMPLE
# Input: nums1 = [4,1,2], nums2 = [1,3,4,2].
# Output: [-1,3,-1]
# Explanation:
#     Fo
# r number 4 in the first array, you cannot find the next greater number for it in
#  the second array, so output -1.
#     For number 1 in the first array, the next g
# reater number for it in the second array is 3.
#     For number 2 in the first arr
# ay, there is no next greater number for it in the second array, so output -1.
# Input: nums1 = [2,4], nums2 = [1,2,3,4].
# Output: [3,-1]
# Explanation:
#     For num
# ber 2 in the first array, the next greater number for it in the second array is
# 3.
#     For number 4 in the first array, there is no next greater number for it i
# n the second array, so output -1.

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

class Solution(object):
    def nextGreaterElement(self, findNums, nums):
        """
        :type findNums: List[int]
        :type nums: List[int]
        :rtype: List[int]
        """
        stk, lookup = [], {}
        for num in nums:
            while stk and num > stk[-1]:
                lookup[stk.pop()] = num
            stk.append(num)
        while stk:
            lookup[stk.pop()] = -1
        return map(lambda x : lookup[x], findNums)
```

kth-smallest-number-in-multiplication-table.py

```
# DESC
# Example 1:
# Nearly every one have used the Multiplication Table. But could you find out the
# k-th smallest number quickly from the multiplication table?
# Given the height m and the length n of a m * n Multiplication Table, and a posit
# ive integer k, you need to return the k-th smallest number in this table.
# Note:
# Example 2:

# NOTE
# The k will be in the range [1, m * n]
# The m and n will be in the range [1, 30000].

# EXAMPLE
# Input: m = 3, n = 3, k = 5
# Output:
# Explanation:
# The Multiplication Table:
# 1        2
# 3
# 2        4        6
# 3        6        9
#
# The 5-th smallest number is 3 (1, 2, 2, 3, 3).
# Input: m = 2, n = 3, k = 6
# Output:
# Explanation:
# The Multiplication Table:
# 1        2
# 3
# 2        4        6
#
# The 6-th smallest number is 6 (1, 2, 2, 3, 4, 6).

# Time:  O(m * log(m * n))
# Space: O(1)

class Solution(object):
    def findKthNumber(self, m, n, k):
        """
        :type m: int
        :type n: int
        :type k: int
        :rtype: int
        """
        def count(target, m, n):
            return sum(min(target//i, n) for i in xrange(1, m+1))

        left, right = 1, m*n
        while left <= right:
            mid = left + (right-left)/2
            if count(mid, m, n) >= k:
                right = mid-1
            else:
                left = mid+1
        return left
```

exclusive-time-of-functions.py

```
# DESC
# Return the exclusive time of each function, sorted by their function id.
# Each log is a string with this format: "{function_id}:{"start" | "end"}:{timesta
# mp}".  For example, "0:start:3" means the function with id 0 started at the begi
# nning of timestamp 3.  "1:end:2" means the function with id 1 ended at the end o
# f timestamp 2.
# On a single threaded CPU, we execute some functions.  Each function has a unique
#  id between 0 and N-1.
# A function's exclusive time is the number of units of time spent in this functio
# n.  Note that this does not include any recursive calls to child functions.
# Example 1:
# The CPU is single threaded which means that only one function is being executed
# at a given time unit.
# We store logs in timestamp order that describe when a function is entered or exited.
# Note:

# NOTE
# 1 <= n <= 100
# Functions will always log when they exit.
# Two functions won't start or end at the same time.

# EXAMPLE
# 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 u
# nits of time and reaches the end of time 1.
# Now function 1 starts at the beginni
# ng 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 exec
# uting for 1 unit of time.
# So function 0 spends 2 + 1 = 3 units of total time ex
# ecuting, and function 1 spends 4 units of total time executing.

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

class Solution(object):
    def exclusiveTime(self, n, logs):
        """
        :type n: int
        :type logs: List[str]
        :rtype: List[int]
        """
        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
```

minimize-rounding-error-to-meet-target.py

```python
# minimize-rounding-error-to-meet-target is not found.
# Time:  O(n) on average
# Space: O(n)

import math
import random


class Solution(object):
    def minimizeError(self, prices, target):
        """
        :type prices: List[str]
        :type target: int
        :rtype: str
        """
        def kthElement(nums, k, compare=lambda a, b: a < b):
            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:
                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

        errors = []
        lower, upper = 0, 0
        for i, p in enumerate(map(float, prices)):
            lower += int(math.floor(p))
            upper += int(math.ceil(p))
            if p != math.floor(p):
                errors.append(p-math.floor(p))
        if not lower <= target <= upper:
            return "-1"

        lower_round_count = upper-target
        kthElement(errors, lower_round_count)
        result = 0.0
        for i in xrange(len(errors)):
            if i < lower_round_count:
                result += errors[i]
            else:
                result += 1.0-errors[i]
        return "{:.3f}".format(result)
```

repeated-dna-sequences.py

```
# DESC
# Example:
# All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, fo
# r example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify r
# epeated sequences within the DNA.
# Write a function to find all the 10-letter-long sequences (substrings) that occu
# r more than once in a DNA molecule.

# NOTE
#

# EXAMPLE
# Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
#
# Output: ["AAAAACCCCC", "CCCCCAAAAA"]

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

import collections

class Solution(object):
    def findRepeatedDnaSequences(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        dict, rolling_hash, res = {}, 0, []

        for i in xrange(len(s)):
            rolling_hash = ((rolling_hash << 3) & 0x3fffffff) | (ord(s[i]) & 7)
            if rolling_hash not in dict:
                dict[rolling_hash] = True
            elif dict[rolling_hash]:
                res.append(s[i - 9: i + 1])
                dict[rolling_hash] = False
        return res

    def findRepeatedDnaSequences2(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        l, r = [], []
        if len(s) < 10: return []
        for i in range(len(s) - 9):
            l.extend([s[i:i + 10]])
        return [k for k, v in collections.Counter(l).items() if v > 1]
```

preimage-size-of-factorial-zeroes-function.py

```python
# DESC
# For example, f(3) = 0 because 3! = 6 has no zeroes at the end, while f(11) = 2 b
# ecause 11! = 39916800 has 2 zeroes at the end. Given K, find how many non-negati
# ve integers x have the property that f(x) = K.
# Note:
# Let f(x) be the number of zeroes at the end of x!. (Recall that x! = 1 * 2 * 3 *
#  ... * x, and by convention, 0! = 1.)
# f(3) = 0

# NOTE
# K will be an integer in the range [0, 10^9].

# EXAMPLE
# Example 1:
# Input: K = 0
# Output: 5
# Explanation: 0!, 1!, 2!, 3!, and 4! end with K
#  = 0 zeroes.
#
# Example 2:
# Input: K = 5
# Output: 0
# Explanation: There is no x such
# that x! ends in K = 5 zeroes.

# Time:  O((logn)^2)
# Space: O(1)

class Solution(object):
    def preimageSizeFZF(self, K):
        """
        :type K: int
        :rtype: int
        """
        def count_of_factorial_primes(n, p):
            cnt = 0
            while n > 0:
                cnt += n//p
                n //= p
            return cnt

        p = 5
        left, right = 0, p*K
        while left <= right:
            mid = left + (right-left)//2
            if count_of_factorial_primes(mid, p) >= K:
                right = mid-1
            else:
                left = mid+1
        return p if count_of_factorial_primes(left, p) == K else 0
```

flip-equivalent-binary-trees.py

```python
# DESC
# Write a function that determines whether two binary trees are flip equivalent.
# The trees are given by root nodes root1 and root2.
# Example 1:
# For a binary tree T, we can define a flip operation as follows: choose any node,
#  and swap the left and right child subtrees.
# Note:
# 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.

# NOTE
# Each value in each tree will be a unique integer in the range [0, 99].
# Each tree will have at most 100 nodes.

# EXAMPLE
# 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.

# 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 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))
```

first-bad-version.py

```python
# DESC
# You are a product manager and currently leading a team to develop a new product.
#  Unfortunately, the latest version of your product fails the quality check. Sinc
# e each version is developed based on the previous version, all the versions afte
# r a bad version are also bad.
# Example:
# Suppose you have n versions [1, 2, ..., n] and you want to find out the first ba
# d one, which causes all the following ones to be bad.
# You are given an API bool isBadVersion(version) which will return whether versio
# n is bad. Implement a function to find the first bad version. You should minimiz
# e the number of calls to the API.

# NOTE
#

# EXAMPLE
# Given n = 5, and version = 4 is the first bad version.
#
# call isBadVersion(3) ->
# false
# call isBadVersion(5) -> true
# call isBadVersion(4) -> true
#
# Then 4 is the f
# irst bad version.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        left, right = 1, n
        while left <= right:
            mid = left + (right - left) / 2
            if isBadVersion(mid): # noqa
                right = mid - 1
            else:
                left = mid + 1
        return left
```

maximum-frequency-stack.py

```
# DESC
# Note:
# Example 1:
# FreqStack has two functions:
# FreqStack
# Implement FreqStack, a class which simulates the operation of a stack-like data
# structure.

# NOTE
# The total number of FreqStack.pop calls will not exceed 10000 in a single test case.
# If there is a tie for most frequent element, the element closest to the top of t
# he stack is removed and returned.
# The total number of FreqStack.push and FreqStack.pop calls will not exceed 15000
# 0 across all test cases.
# push(int x), which pushes an integer x onto the stack.
# pop(), which removes and returns the most frequent element in the stack.
#
#                 If
# there is a tie for most frequent element, the element closest to the top of the
# stack is removed and returned.
# It is guaranteed that FreqStack.pop() won't be called if the stack has zero elements.
# The total number of FreqStack.push calls will not exceed 10000 in a single test case.
# Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.

# EXAMPLE
# Input:
# ["FreqStack","push","push","push","push","push","push","pop","pop","pop"
# ,"pop"],
# [[],[5],[7],[5],[7],[4],[5],[],[],[],[]]
# Output: [null,null,null,null,n
# ull,null,null,5,7,5,4]
# Explanation:
# After making six .push operations, the stack
#  is [5,7,5,7,4,5] from bottom to top.  Then:
#
# pop() -> returns 5, as 5 is the mo
# st frequent.
# The stack becomes [5,7,5,7,4].
#
# pop() -> returns 7, as 5 and 7 is t
# he most frequent, but 7 is closest to the top.
# The stack becomes [5,7,5,4].
#
# pop
# () -> returns 5.
# The stack becomes [5,7,4].
#
# pop() -> returns 4.
# The stack becom
# es [5,7].

# Time:  O(1)
# Space: O(n)

import collections


class FreqStack(object):
```

```
def __init__(self):
    self.__freq = collections.Counter()
    self.__group = collections.defaultdict(list)
    self.__maxfreq = 0

def push(self, x):
    """
    :type x: int
    :rtype: void
    """
    self.__freq[x] += 1
    if self.__freq[x] > self.__maxfreq:
        self.__maxfreq = self.__freq[x]
    self.__group[self.__freq[x]].append(x)

def pop(self):
    """
    :rtype: int
    """
    x = self.__group[self.__maxfreq].pop()
    if not self.__group[self.__maxfreq]:
        self.__group.pop(self.__maxfreq)
        self.__maxfreq -= 1
    self.__freq[x] -= 1
    if not self.__freq[x]:
        self.__freq.pop(x)
    return x
```

moving-average-from-data-stream.py

```python
# moving-average-from-data-stream is not found.
# Time:  O(1)
# Space: O(w)

from collections import deque

class MovingAverage(object):

    def __init__(self, size):
        """
        Initialize your data structure here.
        :type size: int
        """
        self.__size = size
        self.__sum = 0
        self.__q = deque()

    def next(self, val):
        """
        :type val: int
        :rtype: float
        """
        if len(self.__q) == self.__size:
            self.__sum -= self.__q.popleft()
        self.__sum += val
        self.__q.append(val)
        return 1.0 * self.__sum / len(self.__q)
```

3sum-smaller.py

```
# 3sum-smaller is not found.
# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    # @param {integer[]} nums
    # @param {integer} target
    # @return {integer}
    def threeSumSmaller(self, nums, target):
        nums.sort()
        n = len(nums)

        count, k = 0, 2
        while k < n:
            i, j = 0, k - 1
            while i < j:  # Two Pointers, linear time.
                if nums[i] + nums[j] + nums[k] >= target:
                    j -= 1
                else:
                    count += j - i
                    i += 1
            k += 1

        return count
```

plus-one.py

```
# DESC
# The digits are stored such that the most significant digit is at the head of the
#  list, and each element in the array contains a single digit.
# Given a non-empty array of digits representing a non-negative integer, increment
#  one to the integer.
# Example 2:
# Example 1:
# You may assume the integer does not contain any leading zero, except the number
# 0 itself.

# NOTE
#

# EXAMPLE
# Input: [1,2,3]
# Output: [1,2,4]
# Explanation: The array represents the integer 123.
# Input: [4,3,2,1]
# Output: [4,3,2,2]
# Explanation: The array represents the integer
#  4321.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def plusOne(self, digits):
        """
        :type digits: List[int]
        :rtype: List[int]
        """
        for i in reversed(xrange(len(digits))):
            if digits[i] == 9:
                digits[i] = 0
            else:
                digits[i] += 1
                return digits
        digits[0] = 1
        digits.append(0)
        return digits


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def plusOne(self, digits):
        """
        :type digits: List[int]
        :rtype: List[int]
        """
        result = digits[::-1]
        carry = 1
        for i in xrange(len(result)):
            result[i] += carry
            carry, result[i] = divmod(result[i], 10)
        if carry:
            result.append(carry)
        return result[::-1]
```

bus-routes.py

```
# DESC
# We have a list of bus routes. Each routes[i] is a bus route that the i-th bus re
# peats forever. For example if routes[0] = [1, 5, 7], this means that the first b
# us (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.
# We start at bus stop S (initially not on a bus), and we want to go to bus stop T
# . Travelling by buses only, what is the least number of buses we must take to re
# ach our destination? Return -1 if it is not possible.
# Constraints:

# NOTE
# 0 <= routes[i][j] < 10 ^ 6.
# 1 <= routes.length <= 500.
# 1 <= routes[i].length <= 10^5.

# EXAMPLE
# Example:
# Input:
# routes = [[1, 2, 7], [3, 6, 7]]
# S = 1
# T = 6
# Output: 2
# Explanati
# on:
# The best strategy is take the first bus to the bus stop 7, then take the se
# cond bus to the bus stop 6.

# Time:  O(|V| + |E|)
# Space: O(|V| + |E|)

import collections


class Solution(object):
    def numBusesToDestination(self, routes, S, T):
        """
        :type routes: List[List[int]]
        :type S: int
        :type T: int
        :rtype: int
        """
        if S == T:
            return 0

        to_route = collections.defaultdict(set)
        for i, route in enumerate(routes):
            for stop in route:
                to_route[stop].add(i)

        result = 1
        q = [S]
        lookup = set([S])
        while q:
            next_q = []
            for stop in q:
                for i in to_route[stop]:
                    for next_stop in routes[i]:
                        if next_stop in lookup:
                            continue
```

```
                if next_stop == T:
                    return result
                next_q.append(next_stop)
                to_route[next_stop].remove(i)
                lookup.add(next_stop)
    q = next_q
    result += 1

return -1
```

number-of-burgers-with-no-waste-of-ingredients.py

```python
# number-of-burgers-with-no-waste-of-ingredients is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def numOfBurgers(self, tomatoSlices, cheeseSlices):
        """
        :type tomatoSlices: int
        :type cheeseSlices: int
        :rtype: List[int]
        """
        # let the number of jumbo burger be x, the number of small burger be y:
        #   4x + 2y = t
        #   x + y = c
        #   =>
        #   x = t/2-c
        #   y = 2c-t/2
        # since x, y are natural numbers
        #   => t/2 is integer, t/2-c >= 0, 2c-t/2 >= 0
        #   => t%2 == 0, 2c <= t <= 4c
        return [tomatoSlices//2-cheeseSlices, 2*cheeseSlices - tomatoSlices//2] \
               if tomatoSlices%2 == 0 and 2*cheeseSlices <= tomatoSlices <= 4*cheeseSlices \
               else []
```

tag-validator.py

```
# DESC
# Valid Code Examples:
# Given a string representing a code snippet, you need to implement a tag validato
# r to parse the code and return whether it is valid. A code snippet is valid if a
# ll the following rules hold:
# Invalid Code Examples:
# Note:

# NOTE
# A valid TAG_NAME only contain upper-case letters, and has length in range [1,9].
#  Otherwise, the TAG_NAME is invalid.
# The code must be wrapped in a valid closed tag. Otherwise, the code is invalid.
# For simplicity, you could assume the input code (including the any characters me
# ntioned above) only contain letters, digits, '<','>','/','!','[',']' and ' '.
# A < is unmatched if you cannot find a subsequent >. And when you find a < or </,
#  all the subsequent characters until the next > should be parsed as TAG_NAME  (n
# ot necessarily valid).
# A start tag is unmatched if no end tag exists with the same TAG_NAME, and vice v
# ersa. However, you also need to consider the issue of unbalanced when tags are n
# ested.
# The cdata has the following format : <![CDATA[CDATA_CONTENT]]>. The range of CDA
# TA_CONTENT is defined as the characters between <![CDATA[ and the first subseque
# nt ]]>.
# A closed tag (not necessarily valid) has exactly the following format : <TAG_NAM
# E>TAG_CONTENT</TAG_NAME>. Among them, <TAG_NAME> is the start tag, and </TAG_NAM
# E> is the end tag. The TAG_NAME in start and end tags should be the same. A clos
# ed tag is valid if and only if the TAG_NAME and TAG_CONTENT are valid.
# A valid TAG_CONTENT may contain other valid closed tags, cdata and any character
# s (see note1) EXCEPT unmatched <, unmatched start and end tag, and unmatched or
# closed tags with invalid TAG_NAME. Otherwise, the TAG_CONTENT is invalid.
# CDATA_CONTENT may contain any characters. The function of cdata is to forbid the
#  validator to parse CDATA_CONTENT, so even it has some characters that can be pa
# rsed as tag (no matter valid or invalid), you should treat it as regular charact
# ers.

# EXAMPLE
# Input: "<DIV>This is the first line <![CDATA[<div>]]></DIV>"
#
# Output: True
#
# Expl
# anation:
#
# The code is wrapped in a closed tag : <DIV> and </DIV>.
#
# The TAG_NAM
# E is valid, the TAG_CONTENT consists of some characters and cdata.
#
# Although CD
# ATA_CONTENT has unmatched start tag with invalid TAG_NAME, it should be consider
# ed as plain text, not parsed as tag.
#
# So TAG_CONTENT is valid, and then the code
#  is valid. Thus return true.
#
#
# Input: "<DIV>>>  ![cdata[]] <![CDATA[<div>]>]]>]]
# >>]</DIV>"
```

```
#
# Output: True
#
# Explanation:
#
# We first separate the code into : start_
# tag|tag_content|end_tag.
#
# start_tag -> "<DIV>"
#
# end_tag -> "</DIV>"
#
# tag_content
#  could also be separated into : text1|cdata|text2.
#
# text1 -> ">>  ![cdata[]] "
#
#
# cdata -> "<![CDATA[<div>]>]]>", where the CDATA_CONTENT is "<div>]>"
#
# text2 -> "
# ]]>>]"
#
#
# The reason why start_tag is NOT "<DIV>>>" is because of the rule 6.
# The
#  reason why cdata is NOT "<![CDATA[<div>]>]]>]]>" is because of the rule 7.
# Input: "<A>  <B> </A>   </B>"
# Output: False
# Explanation: Unbalanced. If "<A>" is
#  closed, then "<B>" must be unmatched, and vice versa.
#
# Input: "<DIV>  div tag i
# s not closed  <DIV>"
# Output: False
#
# Input: "<DIV>  unmatched <  </DIV>"
# Output:
# False
#
# Input: "<DIV> closed tags with invalid tag name  <b>123</b> </DIV>"
# Outpu
# t: False
#
# Input: "<DIV> unmatched tags with invalid tag name  </1234567890> and
# <CDATA[[]>  </DIV>"
# Output: False
#
# Input: "<DIV>  unmatched start tag <B>  and
# unmatched end tag </C>  </DIV>"
# Output: False

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

class Solution(object):
    def isValid(self, code):
        """
        :type code: str
        :rtype: bool
```

```
    """
    def validText(s, i):
        j = i
        i = s.find("<", i)
        return i != j, i

    def validCData(s, i):
        if s.find("<![CDATA[", i) != i:
            return False, i
        j = s.find("]]>", i)
        if j == -1:
            return False, i
        return True, j+3

    def parseTagName(s, i):
        if s[i] != '<':
            return "", i
        j = s.find('>', i)
        if j == -1 or not (1 <= (j-1-i) <= 9):
            return "", i
        tag = s[i+1:j]
        for c in tag:
            if not (ord('A') <= ord(c) <= ord('Z')):
                return "", i
        return tag, j+1

    def parseContent(s, i):
        while i < len(s):
            result, i = validText(s, i)
            if result:
                continue
            result, i = validCData(s, i)
            if result:
                continue
            result, i = validTag(s, i)
            if result:
                continue
            break
        return i

    def validTag(s, i):
        tag, j = parseTagName(s, i)
        if not tag:
            return False, i
        j = parseContent(s, j)
        k = j + len(tag) + 2
        if k >= len(s) or s[j:k+1] != "</" + tag + ">":
            return False, i
        return True, k+1

    result, i = validTag(code, 0)
    return result and i == len(code)
```

reshape-the-matrix.py

```
# DESC
# Example 1:
# The reshaped matrix need to be filled with all the elements of the original matr
# ix in the same row-traversing order as they were.
# If the 'reshape' operation with given parameters is possible and legal, output t
# he new reshaped matrix; Otherwise, output the original matrix.
# In MATLAB, there is a very useful function called 'reshape', which can reshape a
#  matrix into a new one with different size but keep its original data.
# You're given a matrix represented by a two-dimensional array, and two positive i
# ntegers r and c representing the row number and column number of the wanted resh
# aped matrix, respectively.
# Example 2:
# Note:

# NOTE
# The height and width of the given matrix is in range [1, 100].
# The given r and c are all positive.

# EXAMPLE
# Input:
# nums =
# [[1,2],
#  [3,4]]
# r = 2, c = 4
# Output:
# [[1,2],
#  [3,4]]
# Explanatio
# n:
# There is no way to reshape a 2 * 2 matrix to a 2 * 4 matrix. So output the or
# iginal matrix.
# Input:
# nums =
# [[1,2],
#  [3,4]]
# r = 1, c = 4
# Output:
# [[1,2,3,4]]
# Explanation:
# T
# he row-traversing of nums is [1,2,3,4]. The new reshaped matrix is a 1 * 4 matri
# x, fill it row by row by using the previous list.

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

class Solution(object):
    def matrixReshape(self, nums, r, c):
        """
        :type nums: List[List[int]]
        :type r: int
        :type c: int
        :rtype: List[List[int]]
        """
        if not nums or \
           r*c != len(nums) * len(nums[0]):
            return nums
```

```
result = [[0 for _ in xrange(c)] for _ in xrange(r)]
count = 0
for i in xrange(len(nums)):
    for j in xrange(len(nums[0])):
        result[count/c][count%c] = nums[i][j]
        count += 1
return result
```

pour-water.py

```
# pour-water is not found.
# Time:  O(v * n)
# Space: O(1)

class Solution(object):
    def pourWater(self, heights, V, K):
        """
        :type heights: List[int]
        :type V: int
        :type K: int
        :rtype: List[int]
        """
        for _ in xrange(V):
            best = K
            for d in (-1, 1):
                i = K
                while 0 <= i+d < len(heights) and \
                      heights[i+d] <= heights[i]:
                    if heights[i+d] < heights[i]: best = i+d
                    i += d
                if best != K:
                    break
            heights[best] += 1
        return heights
```

minimum-add-to-make-parentheses-valid.py

```
# DESC
# Note:
# Example 1:
# Example 4:
# Example 2:
# Given a parentheses string, return the minimum number of parentheses we must add
#  to make the resulting string valid.
# Formally, a parentheses string is valid if and only if:
# Given a string S of '(' and ')' parentheses, we add the minimum number of parent
# heses ( '(' or ')', and in any positions ) so that the resulting parentheses str
# ing is valid.
# Example 3:

# NOTE
# S.length <= 1000
# It is the empty string, or
# It can be written as (A), where A is a valid string.
# S only consists of '(' and ')' characters.
# It can be written as AB (A concatenated with B), where A and B are valid strings, or

# EXAMPLE
# Input: "())"
# Output: 1
# Input: "()))(("
# Output: 4
# Input: "()"
# Output: 0
# Input: "((("
# Output: 3

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minAddToMakeValid(self, S):
        """
        :type S: str
        :rtype: int
        """
        add, bal, = 0, 0
        for c in S:
            bal += 1 if c == '(' else -1
            if bal == -1:
                add += 1
                bal += 1
        return add + bal
```

missing-number.py

```
# DESC
# Example 1:
# Example 2:
# Note:
#
# Your algorithm should run in linear runtime complexity. Could you impleme
# nt it using only constant extra space complexity?
# Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find th
# e one that is missing from the array.

# NOTE
#

# EXAMPLE
# Input: [9,6,4,2,3,5,7,0,1]
# Output: 8
# Input: [3,0,1]
# Output: 2

# Time:  O(n)
# Space: O(1)

import operator


class Solution(object):
    def missingNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return reduce(operator.xor, nums, \
                      reduce(operator.xor, xrange(len(nums) + 1)))


class Solution2(object):
    def missingNumber(self, nums):
        return sum(xrange(len(nums)+1)) - sum(nums)
```

linked-list-cycle-ii.py

```
# DESC
# Example 2:
# Follow-up:
#
# Can you solve it without using extra space?
# Example 3:
# Given a linked list, return the node where the cycle begins. If there is no cycl
# e, return null.
# Note: Do not modify the linked list.
# Example 1:
# To represent a cycle in the given linked list, we use an integer pos which repre
# sents 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
#

# EXAMPLE
# 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.
# Input: head = [1], pos = -1
# Output: no cycle
# Explanation: There is no cycle in t
# he linked list.
# Input: head = [3,2,0,-4], pos = 1
# Output: tail connects to node index 1
# Explanat
# ion: There is a cycle in the linked list, where tail connects to the second node
# .

# 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

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
```

max-chunks-to-make-sorted.py

```
# DESC
# What is the most number of chunks we could have made?
# Note:
# Example 2:
# Given an array arr that is a permutation of [0, 1, ..., arr.length - 1], we spli
# t the array into some number of "chunks" (partitions), and individually sort eac
# h chunk.  After concatenating them, the result equals the sorted array.
# Example 1:

# NOTE
# arr will have length in range [1, 10].
# arr[i] will be a permutation of [0, 1, ..., arr.length - 1].

# EXAMPLE
# Input: arr = [1,0,2,3,4]
# Output: 4
# Explanation:
# We can split into two chunks, su
# ch as [1, 0], [2, 3, 4].
# However, splitting into [1, 0], [2], [3], [4] is the hi
# ghest number of chunks possible.
# Input: arr = [4,3,2,1,0]
# Output: 1
# Explanation:
# Splitting into two or more chunk
# s will not return the required result.
# For example, splitting into [4, 3], [2, 1
# , 0] will result in [3, 4, 0, 1, 2], which isn't sorted.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxChunksToSorted(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        result, max_i = 0, 0
        for i, v in enumerate(arr):
            max_i = max(max_i, v)
            if max_i == i:
                result += 1
        return result
```

maximum-subarray-sum-with-one-deletion.py

```python
# maximum-subarray-sum-with-one-deletion is not found.
class 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
```

pizza-with-3n-slices.py

```
# pizza-with-3n-slices is not found.
# Time:  O(n^2)
# Space: O(n)

# [observation]
# 1. we can never take two adjacent slices
# 2. if we want some set of N / 3 non-adjacent slices, there is always a way to take
#
# [proof]
# - for N = 3, it is obviously true.
# - for N' = N + 3,
#   - because it's impossible to have only one unwanted slices between all wanted slices.
#     if it's true, there will be 3N'/2 unwanted slices rather than 2N' unwanted ones, ->&lt;-
#   - so we can always find a sequence of two unwanted slices with one wanted slice
#     to take firstly, then we can find a way to take the remaining N ones by induction, QED

# better optimized space
class Solution(object):
    def maxSizeSlices(self, slices):
        """
        :type slices: List[int]
        :rtype: int
        """
        def maxSizeSlicesLinear(slices, start, end):
            dp = [[0]*(len(slices)//3+1) for _ in xrange(2)]
            for i in xrange(start, end):
                for j in reversed(xrange(1, min(((i-start+1)-1)//2+1, len(slices)//3)+1)):
                    dp[i%2][j] = max(dp[(i-1)%2][j], dp[(i-2)%2][j-1] + slices[i])
            return dp[(end-1)%2][len(slices)//3]

        return max(maxSizeSlicesLinear(slices, 0, len(slices)-1),
                   maxSizeSlicesLinear(slices, 1, len(slices)))


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def maxSizeSlices(self, slices):
        """
        :type slices: List[int]
        :rtype: int
        """
        def maxSizeSlicesLinear(slices, start, end):
            dp = [[0]*(len(slices)//3+1) for _ in xrange(3)]
            for i in xrange(start, end):
                for j in xrange(1, min(((i-start+1)-1)//2+1, len(slices)//3)+1):
                    dp[i%3][j] = max(dp[(i-1)%3][j], dp[(i-2)%3][j-1] + slices[i])
            return dp[(end-1)%3][len(slices)//3]

        return max(maxSizeSlicesLinear(slices, 0, len(slices)-1),
                   maxSizeSlicesLinear(slices, 1, len(slices)))
```

largest-time-for-given-digits.py

```
# DESC
# Note:
# Return the answer as a string of length 5.  If no valid time can be made, return
#  an empty string.
# Example 2:
# Given an array of 4 digits, return the largest 24 hour time that can be made.
# Example 1:
# The smallest 24 hour time is 00:00, and the largest is 23:59.  Starting from 00:
# 00, a time is larger if more time has elapsed since midnight.

# NOTE
# 0 <= A[i] <= 9
# A.length == 4

# EXAMPLE
# Input: [5,5,5,5]
# Output: ""
# Input: [1,2,3,4]
# Output: "23:41"

# Time:  O(1)
# Space: O(1)

import itertools


class Solution(object):
    def largestTimeFromDigits(self, A):
        """
        :type A: List[int]
        :rtype: str
        """
        result = ""
        for i in xrange(len(A)):
            A[i] *= -1
        A.sort()
        for h1, h2, m1, m2 in itertools.permutations(A):
            hours = -(10*h1 + h2)
            mins = -(10*m1 + m2)
            if 0 <= hours < 24 and 0 <= mins < 60:
                result = "{:02}:{:02}".format(hours, mins)
                break
        return result
```

orderly-queue.py

```
# DESC
# Return the lexicographically smallest string we could have after any number of moves.
# A string S of lowercase letters is given.  Then, we may make any number of moves.
# Note:
# Example 2:
# Example 1:
# In each move, we choose one of the first K letters (starting from the left), rem
# ove it, and place it at the end of the string.

# NOTE
# S consists of lowercase letters only.
# 1 <= K <= S.length <= 1000

# EXAMPLE
# Input: S = "cba", K = 1
# Output: "acb"
# Explanation:
# In the first move, we move t
# he 1st character ("c") to the end, obtaining the string "bac".
# In the second mov
# e, we move the 1st character ("b") to the end, obtaining the final result "acb".
# Input: S = "baaca", K = 3
# Output: "aaabc"
# Explanation:
# In the first move, we mo
# ve the 1st character ("b") to the end, obtaining the string "aacab".
# In the seco
# nd move, we move the 3rd character ("c") to the end, obtaining the final result
# "aaabc".

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

class Solution(object):
    def orderlyQueue(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: str
        """
        if K == 1:
            return min(S[i:] + S[:i] for i in xrange(len(S)))
        return "".join(sorted(S))
```

optimal-account-balancing.py

```python
# optimal-account-balancing is not found.
# Time:  O(n * 2^n), n is the size of the debt.
# Space: O(2^n)

import collections


class Solution(object):
    def minTransfers(self, transactions):
        """
        :type transactions: List[List[int]]
        :rtype: int
        """
        accounts = collections.defaultdict(int)
        for transaction in transactions:
            accounts[transaction[0]] += transaction[2]
            accounts[transaction[1]] -= transaction[2]

        debts = [account for account in accounts.values() if account]

        dp = [0]*(2**len(debts))
        sums = [0]*(2**len(debts))
        for i in xrange(len(dp)):
            for j in xrange(len(debts)):
                if (i & (1<<j)) == 0:
                    nxt = i | (1<<j)
                    sums[nxt] = sums[i]+debts[j]
                    if sums[nxt] == 0:
                        dp[nxt] = max(dp[nxt], dp[i]+1)
                    else:
                        dp[nxt] = max(dp[nxt], dp[i])
        return len(debts)-dp[-1]
```

reorder-routes-to-make-all-paths-lead-to-the-city-zero.py

```python
# reorder-routes-to-make-all-paths-lead-to-the-city-zero is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def minReorder(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: int
        """
        lookup, graph = set(), collections.defaultdict(list)
        for u, v in connections:
            lookup.add(u*n+v)
            graph[v].append(u)
            graph[u].append(v)
        result = 0
        stk = [(-1, 0)]
        while stk:
            parent, u = stk.pop()
            result += (parent*n+u in lookup)
            for v in reversed(graph[u]):
                if v == parent:
                    continue
                stk.append((u, v))
        return result


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


class Solution2(object):
    def minReorder(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: int
        """
        def dfs(n, lookup, graph, parent, u):
            result = (parent*n+u in lookup)
            for v in graph[u]:
                if v == parent:
                    continue
                result += dfs(n, lookup, graph, u, v)
            return result

        lookup, graph = set(), collections.defaultdict(list)
        for u, v in connections:
            lookup.add(u*n+v)
            graph[v].append(u)
            graph[u].append(v)
        return dfs(n, lookup, graph, -1, 0)
```

convex-polygon.py

```python
# convex-polygon is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isConvex(self, points):
        """
        :type points: List[List[int]]
        :rtype: bool
        """
        def det(A):
            return A[0][0]*A[1][1] - A[0][1]*A[1][0]

        n, prev, curr = len(points), 0, None
        for i in xrange(len(points)):
            A = [[points[(i+j) % n][0] - points[i][0], points[(i+j) % n][1] - points[i][1]] for j in (1, 2)]
            curr = det(A)
            if curr:
                if curr * prev < 0:
                    return False
                prev = curr
        return True
```

sort-colors.py

```
# DESC
# Example:
# 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, w
# hite and blue.
# Here, we will use the integers 0, 1, and 2 to represent the color red, white, an
# d blue respectively.
# Follow up:
# Note: You are not suppose to use the library's sort function for this problem.

# NOTE
# 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?

# EXAMPLE
# Input: [2,0,2,1,1,0]
# Output: [0,0,1,1,2,2]

# 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

```
# DESC
# Follow up: How would you extend your design to be generic and work with all type
# s, not just integer?
# 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:

# NOTE
#

# 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 i
# t returns 2, the next element. Calling next() after that still return 2.
# You ca
# ll next() the final time and it returns 3, the last element.
# Calling hasNext()
# after that should return false.

# 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

```
# DESC
# Given string S and a dictionary of words words, find the number of words[i] that
#  is a subsequence of S.
# Note:

# 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].

# EXAMPLE
# Example :
# Input:
# S = "abcde"
# words = ["a", "bb", "acd", "ace"]
# Output: 3
# Explan
# ation: There are three words in words that are a subsequence of S: "a", "acd", "
# ace".

# Time:  O(n + w), n is the size of S, w is the size of words
# 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])
```

make-the-string-great.py

```python
# make-the-string-great is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def makeGood(self, s):
        """
        :type s: str
        :rtype: str
        """
        stk = []
        for ch in s:
            counter_ch = ch.upper() if ch.islower() else ch.lower()
            if stk and stk[-1] == counter_ch:
                stk.pop()
            else:
                stk.append(ch)
        return "".join(stk)
```

find-largest-value-in-each-tree-row.py

```python
# DESC
# Example:
# You need to find the largest value in each row of a binary tree.

# NOTE
#

# EXAMPLE
# Input:
#
#          1
#         / \
#        3   2
#       / \   \
#      5   3   9
#
#
# Output: [1, 3, 9]

# 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)
            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
```

check-if-there-is-a-valid-path-in-a-grid.py

```python
# check-if-there-is-a-valid-path-in-a-grid is not found.
# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def hasValidPath(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: bool
        """
        E, S, W, N = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        directions = [
            [W, E], [N, S],
            [W, S], [S, E],
            [W, N], [N, E]
        ]

        for r, c in directions[grid[0][0]-1]:
            if not (0 <= r < len(grid) and 0 <= c < len(grid[0])):
                continue
            pr, pc = 0, 0
            while r != len(grid)-1 or c != len(grid[0])-1:
                for dx, dy in directions[grid[r][c]-1]:
                    nr, nc = r+dx, c+dy
                    if (nr == pr and nc == pc) or \
                       not(0 <= nr < len(grid) and 0 <= nc < len(grid[0])) or \
                       (-dx, -dy) not in directions[grid[nr][nc]-1]:
                        continue
                    pr, pc, r, c = r, c, nr, nc
                    break
                else:
                    return False
            return True
        return len(grid) == len(grid[0]) == 1
```

longest-uncommon-subsequence-i.py

```python
# DESC
# Example 1:
# Constraints:
# Given two strings, you need to find the longest uncommon subsequence of this two
#  strings. The longest uncommon subsequence is defined as the longest subsequence
#  of one of these strings and this subsequence should not be any subsequence of t
# he other string.
# Example 2:
# The input will be two strings, and the output needs to be the length of the long
# est uncommon subsequence. If the longest uncommon subsequence doesn't exist, ret
# urn -1.
# A subsequence is a sequence that can be derived from one sequence by deleting so
# me characters without changing the order of the remaining elements. Trivially, a
# ny string is a subsequence of itself and an empty string is a subsequence of any
#  string.
# Example 3:

# NOTE
# Only letters from a ~ z will appear in input strings.
# Both strings' lengths will be between [1 - 100].

# EXAMPLE
# Input: a = "aba", b = "cdc"
# Output: 3
# Explanation: The longest uncommon subseque
# nce is "aba",
# because "aba" is a subsequence of "aba",
# but not a subsequence o
# f the other string "cdc".
# Note that "cdc" can be also a longest uncommon subsequ
# ence.
# Input: a = "aaa", b = "bbb"
# Output: 3
# Input: a = "aaa", b = "aaa"
# Output: -1

# Time:  O(min(a, b))
# Space: O(1)

class Solution(object):
    def findLUSlength(self, a, b):
        """
        :type a: str
        :type b: str
        :rtype: int
        """
        if a == b:
            return -1
        return max(len(a), len(b))
```

number-of-corner-rectangles.py

```python
# number-of-corner-rectangles is not found.
# Time:  O(n * m^2), n is the number of rows with 1s, m is the number of cols with 1s
# Space: O(n * m)

class Solution(object):
    def countCornerRectangles(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        rows = [[c for c, val in enumerate(row) if val]
                for row in grid]
        result = 0
        for i in xrange(len(rows)):
            lookup = set(rows[i])
            for j in xrange(i):
                count = sum(1 for c in rows[j] if c in lookup)
                result += count*(count-1)/2
        return result
```

contiguous-array.py

```
# contiguous-arra is not found.
# 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
```

max-value-of-equation.py

```python
# max-value-of-equation is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findMaxValueOfEquation(self, points, k):
        """
        :type points: List[List[int]]
        :type k: int
        :rtype: int
        """
        result = float("-inf")
        dq = collections.deque()
        for i, (x, y) in enumerate(points):
            while dq and points[dq[0]][0] < x-k:
                dq.popleft()
            if dq:
                result = max(result, (points[dq[0]][1]-points[dq[0]][0])+y+x)
            while dq and points[dq[-1]][1]-points[dq[-1]][0] <= y-x:
                dq.pop()
            dq.append(i)
        return result
```

pacific-atlantic-water-flow.py

```
# DESC
# Find the list of grid coordinates where water can flow to both the Pacific and A
# tlantic ocean.
# Given an m x n matrix of non-negative integers representing the height of each u
# nit cell in a continent, the "Pacific ocean" touches the left and top edges of t
# he 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.
# Note:
# Example:

# NOTE
# Both m and n are less than 150.
# The order of returned grid coordinates does not matter.

# 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)
```

```
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
```

largest-multiple-of-three.py

```
# largest-multiple-of-three is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def largestMultipleOfThree(self, digits):
        """
        :type digits: List[int]
        :rtype: str
        """
        lookup = {0: [],
                  1: [(1,), (4,), (7,), (2, 2), (5, 2), (5, 5), (8, 2), (8, 5), (8, 8)],
                  2: [(2,), (5,), (8,), (1, 1), (4, 1), (4, 4), (7, 1), (7, 4), (7, 7)]}
        count = collections.Counter(digits)
        for deletes in lookup[sum(digits)%3]:
            delete_count = collections.Counter(deletes)
            if all(count[k] >= v for k, v in delete_count.iteritems()):
                for k, v in delete_count.iteritems():
                    count[k] -= v
                break
        result = "".join(str(d)*count[d] for d in reversed(xrange(10)))
        return "0" if result and result[0] == '0' else result


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def largestMultipleOfThree(self, digits):
        """
        :type digits: List[int]
        :rtype: str
        """
        def candidates_gen(r):
            if r == 0:
                return
            for i in xrange(10):
                yield [i]
            for i in xrange(10):
                for j in xrange(i+1):
                    yield [i, j]

        count, r = collections.Counter(digits), sum(digits)%3
        for deletes in candidates_gen(r):
            delete_count = collections.Counter(deletes)
            if sum(deletes)%3 == r and \
               all(count[k] >= v for k, v in delete_count.iteritems()):
                for k, v in delete_count.iteritems():
                    count[k] -= v
                break
        result = "".join(str(d)*count[d] for d in reversed(xrange(10)))
        return "0" if result and result[0] == '0' else result
```

remove-linked-list-elements.py

```python
# DESC
# Remove all elements from a linked list of integers that have value val.
# Example:

# NOTE
#

# EXAMPLE
# Input:  1->2->6->3->4->5->6, val = 6
# Output: 1->2->3->4->5

# Time:  O(n)
# Space: O(1)

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


class Solution(object):
    # @param {ListNode} head
    # @param {integer} val
    # @return {ListNode}
    def removeElements(self, head, val):
        dummy = ListNode(float("-inf"))
        dummy.next = head
        prev, curr = dummy, dummy.next

        while curr:
            if curr.val == val:
                prev.next = curr.next
            else:
                prev = curr

            curr = curr.next

        return dummy.next
```

course-schedule-iii.py

```
# DESC
# There are n different online courses numbered from 1 to n. Each course has some
# duration(course length) t and closed on dth day. A course should be taken contin
# uously for t days and must be finished before or on the dth day. You will start
# at the 1st day.
# Given n online courses represented by pairs (t,d), your task is to find the maxi
# mal number of courses that can be taken.
# Note:
# Example:

# NOTE
# You can't take two courses simultaneously.
# The integer 1 <= d, t, n <= 10,000.

# EXAMPLE
# Input: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
# Output: 3
# Explanati
# on:
# There're totally 4 courses, but you can take 3 courses at most:
# First, take
#  the 1st course, it costs 100 days so you will finish it on the 100th day, and r
# eady to take the next course on the 101st day.
# Second, take the 3rd course, it c
# osts 1000 days so you will finish it on the 1100th day, and ready to take the ne
# xt course on the 1101st day.
# Third, take the 2nd course, it costs 200 days so y
# ou will finish it on the 1300th day.
# The 4th course cannot be taken now, since
# you will finish it on the 3300th day, which exceeds the closed date.

# Time:  O(nlogn)
# Space: O(k), k is the number of courses you can take

import collections
import heapq


class Solution(object):
    def scheduleCourse(self, courses):
        """
        :type courses: List[List[int]]
        :rtype: int
        """
        courses.sort(key=lambda t_end: t_end[1])
        max_heap = []
        now = 0
        for t, end in courses:
            now += t
            heapq.heappush(max_heap, -t)
            if now > end:
                now += heapq.heappop(max_heap)
        return len(max_heap)
```

4-keys-keyboard.py

```python
# 4-keys-keyboard is not found.
# Time:  O(1)
# Space: O(1)


class Solution(object):
    def maxA(self, N):
        """
        :type N: int
        :rtype: int
        """
        if N < 7:
            return N
        if N == 10:
            return 20  # the following rule doesn't hold when N = 10

        n = N // 5 + 1  # n3 + n4 increases one every 5 keys
        # (1) n     =     n3 +     n4
        # (2) N + 1 = 4 * n3 + 5 * n4
        #     5 x (1) - (2) => 5*n - N - 1 = n3
        n3 = 5*n - N - 1
        n4 = n - n3
        return 3**n3 * 4**n4


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def maxA(self, N):
        """
        :type N: int
        :rtype: int
        """
        if N < 7:
            return N
        dp = range(N+1)
        for i in xrange(7, N+1):
            dp[i % 6] = max(dp[(i-4) % 6]*3, dp[(i-5) % 6]*4)
        return dp[N % 6]
```

minimum-distance-between-bst-nodes.py

```
# DESC
# Example :
# Given a Binary Search Tree (BST) with the root node root, return the minimum dif
# ference between the values of any two different nodes in the tree.
# Note:

# NOTE
# This question is the same as 530: https://leetcode.com/problems/minimum-absolute
# -difference-in-bst/
# The size of the BST will be between 2 and 100.
# The BST is always valid, each node's value is an integer, and each node's value
# is different.

# EXAMPLE
# Input: root = [4,2,6,1,3,null,null]
# Output: 1
# Explanation:
# Note that root is a T
# reeNode object, not an array.
#
# The given tree [4,2,6,1,3,null,null] is represent
# ed by the following diagram:
#
#           4
#         /   \
#       2      6
#      / \
#
#     1   3
#
# while the minimum difference in this tree is 1, it occurs betwe
# en node 1 and node 2, also between node 3 and node 2.

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

class Solution(object):
    def minDiffInBST(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node):
            if not node:
                return
            dfs(node.left)
            self.result = min(self.result, node.val-self.prev)
            self.prev = node.val
            dfs(node.right)

        self.prev = float('-inf')
        self.result = float('inf')
        dfs(root)
        return self.result
```

card-flipping-game.py

```
# DESC
# Here, fronts[i] and backs[i] represent the number on the front and back of card i.
# A flip swaps the front and back numbers, so the value on the front is now on the
#  back and vice versa.
# What is the smallest number that is good?  If no number is good, output 0.
# Note:
# Example:
# We flip any number of cards, and after we choose one card.
# If the number X on the back of the chosen card is not on the front of any card,
# then this number X is good.
# On a table are N cards, with a positive integer printed on the front and back of
#  each card (possibly different).

# NOTE
# 1 <= fronts[i] <= 2000.
# 1 <= backs[i] <= 2000.
# 1 <= fronts.length == backs.length <= 1000.

# EXAMPLE
# Input: fronts = [1,2,4,4,7], backs = [1,3,4,1,3]
# Output: 2
# Explanation: If we fl
# ip the second card, the fronts are [1,3,4,4,7] and the backs are [1,2,4,1,3].
# We
#  choose the second card, which has number 2 on the back, and it isn't on the fro
# nt of any card, so 2 is good.

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

import itertools


class Solution(object):
    def flipgame(self, fronts, backs):
        """
        :type fronts: List[int]
        :type backs: List[int]
        :rtype: int
        """
        same = {n for i, n in enumerate(fronts) if n == backs[i]}
        result = float("inf")
        for n in itertools.chain(fronts, backs):
            if n not in same:
                result = min(result, n)
        return result if result < float("inf") else 0
```

trapping-rain-water-ii.py

```
# DESC
# Given an m x n matrix of positive integers representing the height of each unit
# cell in a 2D elevation map, compute the volume of water it is able to trap after
#  raining.
# The above image represents the elevation map [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3
# ,2,3,1]] before the rain.
# Constraints:
# Example:
# After the rain, water is trapped between the blocks. The total volume of water t
# rapped is 4.

# NOTE
# 0 <= heightMap[i][j] <= 20000
# 1 <= m, n <= 110

# EXAMPLE
# Given the following 3x6 height map:
# [
#   [1,4,3,1,3,2],
#   [3,2,1,3,2,4],
#   [2,3,3
# ,2,3,1]
# ]
#
# Return 4.

# Time:  O(m * n * log(m + n)) ~ O(m * n * log(m * n))
# Space: O(m * n)

from heapq import heappush, heappop

class Solution(object):
    def trapRainWater(self, heightMap):
        """
        :type heightMap: List[List[int]]
        :rtype: int
        """
        m = len(heightMap)
        if not m:
            return 0
        n = len(heightMap[0])
        if not n:
            return 0

        is_visited = [[False for i in xrange(n)] for j in xrange(m)]

        heap = []
        for i in xrange(m):
            heappush(heap, [heightMap[i][0], i, 0])
            is_visited[i][0] = True
            heappush(heap, [heightMap[i][n-1], i, n-1])
            is_visited[i][n-1] = True
        for j in xrange(n):
            heappush(heap, [heightMap[0][j], 0, j])
            is_visited[0][j] = True
            heappush(heap, [heightMap[m-1][j], m-1, j])
            is_visited[m-1][j] = True
```

```
trap = 0
while heap:
    height, i, j = heappop(heap)
    for (dx, dy) in [(1,0), (-1,0), (0,1), (0,-1)]:
        x, y = i+dx, j+dy
        if 0 <= x < m and 0 <= y < n and not is_visited[x][y]:
            trap += max(0, height - heightMap[x][y])
            heappush(heap, [max(height, heightMap[x][y]), x, y])
            is_visited[x][y] = True

return trap
```

stone-game-iii.py

```
# stone-game-iii is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def stoneGameIII(self, stoneValue):
        """
        :type stoneValue: List[int]
        :rtype: str
        """
        dp = [float("-inf")]*3
        dp[len(stoneValue)%3] = 0
        for i in reversed(xrange(len(stoneValue))):
            max_dp, curr = float("-inf"), 0
            for j in xrange(min(3, len(stoneValue)-i)):
                curr += stoneValue[i+j]
                max_dp = max(max_dp, curr-dp[(i+j+1)%3])
            dp[i%3] = max_dp
        return ["Tie", "Alice", "Bob"][cmp(dp[0], 0)]
```

maximal-square.py

```
# DESC
# Given a 2D binary matrix filled with 0's and 1's, find the largest square contai
# ning only 1's and return its area.
# Example:

# NOTE
#

# EXAMPLE
# Input:
#
# 1 0 1 0 0
# 1 0 1 1 1
# 1 1 1 1 1
# 1 0 0 1 0
#
# Output: 4

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

class Solution(object):
    # @param {character[][]} matrix
    # @return {integer}
    def maximalSquare(self, matrix):
        if not matrix:
            return 0

        m, n = len(matrix), len(matrix[0])
        size = [[0 for j in xrange(n)] for i in xrange(2)]
        max_size = 0

        for j in xrange(n):
            if matrix[0][j] == '1':
                size[0][j] = 1
            max_size = max(max_size, size[0][j])

        for i in xrange(1, m):
            if matrix[i][0] == '1':
                size[i % 2][0] = 1
            else:
                size[i % 2][0] = 0
            for j in xrange(1, n):
                if matrix[i][j] == '1':
                    size[i % 2][j] = min(size[i % 2][j - 1], \
                                         size[(i - 1) % 2][j], \
                                         size[(i - 1) % 2][j - 1]) + 1
                    max_size = max(max_size, size[i % 2][j])
                else:
                    size[i % 2][j] = 0

        return max_size * max_size


# Time:  O(n^2)
# Space: O(n^2)
# DP.
class Solution2(object):
```

```
    # @param {character[][]} matrix
    # @return {integer}
    def maximalSquare(self, matrix):
        if not matrix:
            return 0

        m, n = len(matrix), len(matrix[0])
        size = [[0 for j in xrange(n)] for i in xrange(m)]
        max_size = 0

        for j in xrange(n):
            if matrix[0][j] == '1':
                size[0][j] = 1
            max_size = max(max_size, size[0][j])

        for i in xrange(1, m):
            if matrix[i][0] == '1':
                size[i][0] = 1
            else:
                size[i][0] = 0
            for j in xrange(1, n):
                if matrix[i][j] == '1':
                    size[i][j] = min(size[i][j - 1],  \
                                     size[i - 1][j],  \
                                     size[i - 1][j - 1]) + 1
                    max_size = max(max_size, size[i][j])
                else:
                    size[i][j] = 0

        return max_size * max_size


# Time:  O(n^2)
# Space: O(n^2)
# DP.
class Solution3(object):
    # @param {character[][]} matrix
    # @return {integer}
    def maximalSquare(self, matrix):
        if not matrix:
            return 0

        H, W = 0, 1
        # DP table stores (h, w) for each (i, j).
        table = [[[0, 0] for j in xrange(len(matrix[0]))] \
                         for i in xrange(len(matrix))]
        for i in reversed(xrange(len(matrix))):
            for j in reversed(xrange(len(matrix[i]))):
                # Find the largest h such that (i, j) to (i + h - 1, j) are feasible.
                # Find the largest w such that (i, j) to (i, j + w - 1) are feasible.
                if matrix[i][j] == '1':
                    h, w = 1, 1
                    if i + 1 < len(matrix):
                        h = table[i + 1][j][H] + 1
                    if j + 1 < len(matrix[i]):
                        w = table[i][j + 1][W] + 1
                    table[i][j] = [h, w]

        # A table stores the length of largest square for each (i, j).
        s = [[0 for j in xrange(len(matrix[0]))] \
```

```
        for i in xrange(len(matrix))]
max_square_area = 0
for i in reversed(xrange(len(matrix))):
    for j in reversed(xrange(len(matrix[i]))):
        side = min(table[i][j][H], table[i][j][W])
        if matrix[i][j] == '1':
            # Get the length of largest square with bottom-left corner (i, j).
            if i + 1 < len(matrix) and j + 1 < len(matrix[i + 1]):
                side = min(s[i + 1][j + 1] + 1, side)
            s[i][j] = side
            max_square_area = max(max_square_area, side * side)

return max_square_area
```

minesweeper.py

```
# DESC
# Example 2:
# Let's play the minesweeper game (Wikipedia, online game)!
# Example 1:
# Now given the next click position (row and column indices) among all the unrevea
# led squares ('M' or 'E'), return the board after revealing this position accordi
# ng to the following rules:
# You are given a 2D char matrix representing the game board. 'M' represents an un
# revealed mine, 'E' represents an unrevealed empty square, 'B' represents a revea
# led blank square that has no adjacent (above, below, left, right, and all 4 diag
# onals) mines, digit ('1' to '8') represents how many mines are adjacent to this
# revealed square, and finally 'X' represents a revealed mine.
# Note:

# NOTE
# If an empty square ('E') with no adjacent mines is revealed, then change it to r
# evealed blank ('B') and all of its adjacent unrevealed squares should be reveale
# d recursively.
# The click position will only be an unrevealed square ('M' or 'E'), which also me
# ans 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).
# Return the board when no more squares will be revealed.
# For simplicity, not mentioned rules should be ignored in this problem. For examp
# le, you don't need to reveal all the unrevealed mines when the game is over, con
# sider any cases that you will win the game or flag any squares.
# If a mine ('M') is revealed, then the game is over - change it to 'X'.
# The range of the input matrix's height and width is [1,50].
# If an empty square ('E') with at least one adjacent mine is revealed, then chang
# e it to a digit ('1' to '8') representing the number of adjacent mines.

# EXAMPLE
# Input:
#
# [['B', '1', 'E', '1', 'B'],
#  ['B', '1', 'M', '1', 'B'],
#  ['B', '1', '1'
# , '1', 'B'],
#  ['B', 'B', 'B', 'B', 'B']]
#
# Click : [1,2]
#
# Output:
#
# [['B', '1', '
# E', '1', 'B'],
#  ['B', '1', 'X', '1', 'B'],
#  ['B', '1', '1', '1', 'B'],
#  ['B', 'B
# ', 'B', 'B', 'B']]
#
# Explanation:
# 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:

# 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:
                                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
```

graph-valid-tree.py

```
# graph-valid-tree is not found.
# Time:  O(|V| + |E|)
# Space: O(|V| + |E|)

import collections


# BFS solution. Same complexity but faster version.
class Solution(object):
    # @param {integer} n
    # @param {integer[][]} edges
    # @return {boolean}
    def validTree(self, n, edges):
        if len(edges) != n - 1:  # Check number of edges.
            return False

        # init node's neighbors in dict
        neighbors = collections.defaultdict(list)
        for u, v in edges:
            neighbors[u].append(v)
            neighbors[v].append(u)

        # BFS to check whether the graph is valid tree.
        q = collections.deque([0])
        visited = set([0])
        while q:
            curr = q.popleft()
            for node in neighbors[curr]:
                if node not in visited:
                    visited.add(node)
                    q.append(node)

        return len(visited) == n


# Time:  O(|V| + |E|)
# Space: O(|V| + |E|)
# BFS solution.
class Solution2(object):
    # @param {integer} n
    # @param {integer[][]} edges
    # @return {boolean}
    def validTree(self, n, edges):
        # A structure to track each node's [visited_from, neighbors]
        visited_from = [-1] * n
        neighbors = collections.defaultdict(list)
        for u, v in edges:
            neighbors[u].append(v)
            neighbors[v].append(u)

        # BFS to check whether the graph is valid tree.
        q = collections.deque([0])
        visited = set([0])
        while q:
            i = q.popleft()
            for node in neighbors[i]:
                if node != visited_from[i]:
                    if node in visited:
```

```
                return False
            else:
                visited.add(node)
                visited_from[node] = i
                q.append(node)
    return len(visited) == n
```

maximum-number-of-occurrences-of-a-substring.py

```python
# maximum-number-of-occurrences-of-a-substring is not found.
# Time:  O(n)
# Space: O(n)

import collections


# rolling hash (Rabin-Karp Algorithm)
class Solution(object):
    def maxFreq(self, s, maxLetters, minSize, maxSize):
        """
        :type s: str
        :type maxLetters: int
        :type minSize: int
        :type maxSize: int
        :rtype: int
        """
        M, p = 10**9+7, 113
        power, rolling_hash = pow(p, minSize-1, M), 0

        left = 0
        lookup, count = collections.defaultdict(int), collections.defaultdict(int)
        for right in xrange(len(s)):
            count[s[right]] += 1
            if right-left+1 > minSize:
                count[s[left]] -= 1
                rolling_hash = (rolling_hash - ord(s[left])*power) % M
                if count[s[left]] == 0:
                    count.pop(s[left])
                left += 1
            rolling_hash = (rolling_hash*p + ord(s[right])) % M
            if right-left+1 == minSize and len(count) <= maxLetters:
                lookup[rolling_hash] += 1
        return max(lookup.values() or [0])


# Time:  O(m * n), m = 26
# Space: O(m * n)
class Solution2(object):
    def maxFreq(self, s, maxLetters, minSize, maxSize):
        """
        :type s: str
        :type maxLetters: int
        :type minSize: int
        :type maxSize: int
        :rtype: int
        """
        lookup = {}
        for right in xrange(minSize-1, len(s)):
            word = s[right-minSize+1:right+1]
            if word in lookup:
                lookup[word] += 1
            elif len(collections.Counter(word)) <= maxLetters:
                lookup[word] = 1
        return max(lookup.values() or [0])
```

jump-game-v.py

```
# jump-game-v is not found.
# Time:  O(n)
# Space: O(n)

import collections
import itertools


# sliding window + top-down dp
class Solution(object):
    def maxJumps(self, arr, d):
        """
        :type arr: List[int]
        :type d: int
        :rtype: int
        """
        def dp(arr, d, i, left, right, lookup):
            if lookup[i]:
                return lookup[i]
            lookup[i] = 1
            for j in itertools.chain(left[i], right[i]):
                # each dp[j] will be visited at most twice
                lookup[i] = max(lookup[i], dp(arr, d, j, left, right, lookup)+1)
            return lookup[i]

        left, decreasing_dq = [[] for _ in xrange(len(arr))], collections.deque()
        for i in xrange(len(arr)):
            if decreasing_dq and i - decreasing_dq[0] == d+1:
                decreasing_dq.popleft()
            while decreasing_dq and arr[decreasing_dq[-1]] < arr[i]:
                if left[i] and arr[left[i][-1]] != arr[decreasing_dq[-1]]:
                    left[i] = []
                left[i].append(decreasing_dq.pop())
            decreasing_dq.append(i)
        right, decreasing_dq = [[] for _ in xrange(len(arr))], collections.deque()
        for i in reversed(xrange(len(arr))):
            if decreasing_dq and decreasing_dq[0] - i == d+1:
                decreasing_dq.popleft()
            while decreasing_dq and arr[decreasing_dq[-1]] < arr[i]:
                if right[i] and arr[right[i][-1]] != arr[decreasing_dq[-1]]:
                    right[i] = []
                right[i].append(decreasing_dq.pop())
            decreasing_dq.append(i)

        lookup = [0]*len(arr)
        return max(itertools.imap(lambda x: dp(arr, d, x, left, right, lookup), xrange(len(arr))))


# Time:  O(nlogn)
# Space: O(n)
# mono stack + bottom-up dp
class Solution2(object):
    def maxJumps(self, arr, d):
        """
        :type arr: List[int]
        :type d: int
        :rtype: int
        """
```

```
        left, decreasing_stk = [[] for _ in xrange(len(arr))], []
        for i in xrange(len(arr)):
            while decreasing_stk and arr[decreasing_stk[-1]] < arr[i]:
                if i - decreasing_stk[-1] <= d:
                    if left[i] and arr[left[i][-1]] != arr[decreasing_stk[-1]]:
                        left[i] = []
                    left[i].append(decreasing_stk[-1])
                decreasing_stk.pop()
            decreasing_stk.append(i)
        right, decreasing_stk = [[] for _ in xrange(len(arr))], []
        for i in reversed(xrange(len(arr))):
            while decreasing_stk and arr[decreasing_stk[-1]] < arr[i]:
                if decreasing_stk[-1] - i <= d:
                    if right[i] and arr[right[i][-1]] != arr[decreasing_stk[-1]]:
                        right[i] = []
                    right[i].append(decreasing_stk[-1])
                decreasing_stk.pop()
            decreasing_stk.append(i)

        dp = [0]*len(arr)
        for a, i in sorted([a, i] for i, a in enumerate(arr)):
            dp[i] = 1
            for j in itertools.chain(left[i], right[i]):
                # each dp[j] will be visited at most twice
                dp[i] = max(dp[i], dp[j]+1)
        return max(dp)


# Template:
# https://github.com/kamyu104/FacebookHackerCup-2018/blob/master/Final%20Round/the_claw.py
class SegmentTree(object):
    def __init__(self, N,
                 build_fn=lambda x, y: [y]*(2*x),
                 query_fn=max,
                 update_fn=lambda x, y: y,
                 default_val=0):
        self.N = N
        self.H = (N-1).bit_length()
        self.query_fn = query_fn
        self.update_fn = update_fn
        self.default_val = default_val
        self.tree = build_fn(N, default_val)
        self.lazy = [None]*N

    def __apply(self, x, val):
        self.tree[x] = self.update_fn(self.tree[x], val)
        if x < self.N:
            self.lazy[x] = self.update_fn(self.lazy[x], val)

    def update(self, L, R, h):  # Time: O(logN), Space: O(N)
        def pull(x):
            while x > 1:
                x //= 2
                self.tree[x] = self.query_fn(self.tree[x*2], self.tree[x*2+1])
                if self.lazy[x] is not None:
                    self.tree[x] = self.update_fn(self.tree[x], self.lazy[x])
        L += self.N
        R += self.N
        L0, R0 = L, R
        while L <= R:
```

```
            if L & 1:  # is right child
                self.__apply(L, h)
                L += 1
            if R & 1 == 0:  # is left child
                self.__apply(R, h)
                R -= 1
            L //= 2
            R //= 2
        pull(L0)
        pull(R0)

    def query(self, L, R):  # Time: O(logN), Space: O(N)
        def push(x):
            n = 2**self.H
            while n != 1:
                y = x // n
                if self.lazy[y] is not None:
                    self.__apply(y*2, self.lazy[y])
                    self.__apply(y*2 + 1, self.lazy[y])
                    self.lazy[y] = None
                n //= 2

        result = self.default_val
        if L > R:
            return result

        L += self.N
        R += self.N
        push(L)
        push(R)
        while L <= R:
            if L & 1:  # is right child
                result = self.query_fn(result, self.tree[L])
                L += 1
            if R & 1 == 0:  # is left child
                result = self.query_fn(result, self.tree[R])
                R -= 1
            L //= 2
            R //= 2
        return result

    def __str__(self):
        showList = []
        for i in xrange(self.N):
            showList.append(self.query(i, i))
        return ",".join(map(str, showList))


# Time:  O(nlogn)
# Space: O(n)
# mono stack + bottom-up dp + segment tree
class Solution3(object):
    def maxJumps(self, arr, d):
        """
        :type arr: List[int]
        :type d: int
        :rtype: int
        """
        left, decreasing_stk = range(len(arr)), []
        for i in xrange(len(arr)):
```

```
    while decreasing_stk and arr[decreasing_stk[-1]] < arr[i]:
        if i - decreasing_stk[-1] <= d:
            left[i] = decreasing_stk[-1]
        decreasing_stk.pop()
    decreasing_stk.append(i)
right, decreasing_stk = range(len(arr)), []
for i in reversed(xrange(len(arr))):
    while decreasing_stk and arr[decreasing_stk[-1]] < arr[i]:
        if decreasing_stk[-1] - i <= d:
            right[i] = decreasing_stk[-1]
        decreasing_stk.pop()
    decreasing_stk.append(i)

segment_tree = SegmentTree(len(arr))
for _, i in sorted([x, i] for i, x in enumerate(arr)):
    segment_tree.update(i, i, segment_tree.query(left[i], right[i]) + 1)
return segment_tree.query(0, len(arr)-1)
```

linked-list-cycle.py

```
# DESC
# Example 2:
# Follow up:
# Can you solve it using O(1) (i.e. constant) memory?
# Example 1:
# To represent a cycle in the given linked list, we use an integer pos which repre
# sents 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.
# Example 3:
# Given a linked list, determine if it has a cycle in it.

# NOTE
#

# EXAMPLE
# Input: head = [1,2], pos = 0
# Output: true
# Explanation: There is a cycle in the l
# inked list, where tail connects to the first node.
# Input: head = [1], pos = -1
# Output: false
# Explanation: There is no cycle in the
# linked list.
# Input: head = [3,2,0,-4], pos = 1
# Output: true
# Explanation: There is a cycle in
# the linked list, where tail connects to the second node.

# Time:  O(n)
# Space: O(1)

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

class Solution(object):
    # @param head, a ListNode
    # @return a boolean
    def hasCycle(self, head):
        fast, slow = head, head
        while fast and fast.next:
            fast, slow = fast.next.next, slow.next
            if fast is slow:
                return True
        return False
```

count-numbers-with-unique-digits.py

```
# DESC
# Given a non-negative integer n, count all numbers with unique digits, x, where 0
#   x < 10n.
# Constraints:
# Example:

# NOTE
# 0 <= n <= 8

# 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

# 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
```

parsing-a-boolean-expression.py

```python
# DESC
# An expression can either be:
# Example 4:
# Return the result of evaluating a given boolean expression, represented as a string.
# Example 1:
# Example 2:
# Constraints:
# Example 3:

# NOTE
# "t", evaluating to True;
# "!(expr)", evaluating to the logical NOT of the inner expression expr;
# "|(expr1,expr2,...)", evaluating to the logical OR of 2 or more inner expression
# s expr1, expr2, ...
# "f", evaluating to False;
# 1 <= expression.length <= 20000
# expression[i] consists of characters in {'(', ')', '&', '|', '!', 't', 'f', ','}.
# "&(expr1,expr2,...)", evaluating to the logical AND of 2 or more inner expressio
# ns expr1, expr2, ...;
# expression is a valid expression representing a boolean, as given in the description.

# EXAMPLE
# Input: expression = "!(f)"
# Output: true
# Input: expression = "|(f,t)"
# Output: true
# Input: expression = "&(t,f)"
# Output: false
# Input: expression = "|(&(t,f,t),!(t))"
# Output: false

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

class Solution(object):
    def parseBoolExpr(self, expression):
        """
        :type expression: str
        :rtype: bool
        """
        def parse(expression, i):
            if expression[i[0]] not in "&|!":
                result = expression[i[0]] == 't'
                i[0] += 1
                return result
            op = expression[i[0]]
            i[0] += 2
            stk = []
            while expression[i[0]] != ')':
                if expression[i[0]] == ',':
                    i[0] += 1
                    continue
                stk.append(parse(expression, i))
            i[0] += 1
            if op == '&':
                return all(stk)
            if op == '|':
                return any(stk)
```

```
        return not stk[0]

    return parse(expression, [0])
```

intersection-of-two-arrays-ii.py

```
# DESC
# Given two arrays, write a function to compute their intersection.
# Example 1:
# Example 2:
# Note:
# Follow up:

# NOTE
# What if elements of nums2 are stored on disk, and the memory is limited such tha
# t you cannot load all elements into the memory at once?
# Each element in the result should appear as many times as it shows in both arrays.
# What if nums1's size is small compared to nums2's size? Which algorithm is better?
# What if the given array is already sorted? How would you optimize your algorithm?
# The result can be in any order.

# EXAMPLE
# Input: nums1 = [1,2,2,1], nums2 = [2,2]
# Output: [2,2]
# Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
# Output: [4,9]

# If the given array is not sorted and the memory is unlimited:
#   - Time:  O(m + n)
#   - Space: O(min(m, n))
# elif the given array is already sorted:
#   if m << n or m >> n:
#     - Time:  O(min(m, n) * log(max(m, n)))
#     - Space: O(1)
#   else:
#     - Time:  O(m + n)
#     - Soace: O(1)
# else: (the given array is not sorted and the memory is limited)
#     - Time:  O(max(m, n) * log(max(m, n)))
#     - Space: O(1)

import collections


class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        if len(nums1) > len(nums2):
            return self.intersect(nums2, nums1)

        lookup = collections.defaultdict(int)
        for i in nums1:
            lookup[i] += 1

        res = []
        for i in nums2:
            if lookup[i] > 0:
                res += i,
                lookup[i] -= 1
```

```
        return res

    def intersect2(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        c = collections.Counter(nums1) & collections.Counter(nums2)
        intersect = []
        for i in c:
            intersect.extend([i] * c[i])
        return intersect


# If the given array is already sorted, and the memory is limited, and (m << n or m >> n).
# Time:  O(min(m, n) * log(max(m, n)))
# Space: O(1)
# Binary search solution.
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        if len(nums1) > len(nums2):
            return self.intersect(nums2, nums1)

        def binary_search(compare, nums, left, right, target):
            while left < right:
                mid = left + (right - left) / 2
                if compare(nums[mid], target):
                    right = mid
                else:
                    left = mid + 1
            return left

        nums1.sort(), nums2.sort()  # Make sure it is sorted, doesn't count in time.

        res = []
        left = 0
        for i in nums1:
            left = binary_search(lambda x, y: x >= y, nums2, left, len(nums2), i)
            if left != len(nums2) and nums2[left] == i:
                res += i,
                left += 1

        return res


# If the given array is already sorted, and the memory is limited or m ~ n.
# Time:  O(m + n)
# Space: O(1)
# Two pointers solution.
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
```

```
        :rtype: List[int]
        """
        nums1.sort(), nums2.sort()  # Make sure it is sorted, doesn't count in time.

        res = []

        it1, it2 = 0, 0
        while it1 < len(nums1) and it2 < len(nums2):
            if nums1[it1] < nums2[it2]:
                it1 += 1
            elif nums1[it1] > nums2[it2]:
                it2 += 1
            else:
                res += nums1[it1],
                it1 += 1
                it2 += 1

        return res


# If the given array is not sorted, and the memory is limited.
# Time:  O(max(m, n) * log(max(m, n)))
# Space: O(1)
# Two pointers solution.
class Solution(object):
    def intersect(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        nums1.sort(), nums2.sort()  # O(max(m, n) * log(max(m, n)))

        res = []

        it1, it2 = 0, 0
        while it1 < len(nums1) and it2 < len(nums2):
            if nums1[it1] < nums2[it2]:
                it1 += 1
            elif nums1[it1] > nums2[it2]:
                it2 += 1
            else:
                res += nums1[it1],
                it1 += 1
                it2 += 1

        return res
```

remove-element.py

```
# DESC
# Note that the input array is passed in by reference, which means modification to
#  the input array will be known to the caller as well.
# Confused why the returned value is an integer but your answer is an array?
# Internally you can think of this:
# Example 1:
# Clarification:
# Do not allocate extra space for another array, you must do this by modifying the
#  input array in-place with O(1) extra memory.
# Given an array nums and a value val, remove all instances of that value in-place
#  and return the new length.
# The order of elements can be changed. It doesn't matter what you leave beyond th
# e new length.
# Example 2:

# NOTE
#

# EXAMPLE
# // nums is passed in by reference. (i.e., without making a copy)
# int len = remov
# eElement(nums, val);
#
# // any modification to nums in your function would be know
# n by the caller.
# // using the length returned by your function, it prints the fi
# rst len elements.
# for (int i = 0; i < len; i++) {
#     print(nums[i]);
# }
# Given nums = [3,2,2,3], val = 3,
#
# Your function should return length = 2, with t
# he first two elements of nums being 2.
#
# It doesn't matter what you leave beyond
# the returned length.
# Given nums = [0,1,2,2,3,0,4,2], val = 2,
#
# Your function should return length = 5
# , with the first five elements of nums containing 0, 1, 3, 0, and 4.
#
# Note that
# the order of those five elements can be arbitrary.
#
# It doesn't matter what value
# s are set beyond the returned length.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param    A       a list of integers
    # @param    elem    an integer, value need to be removed
    # @return an integer
    def removeElement(self, A, elem):
        i, last = 0, len(A) - 1
        while i <= last:
```

```
    if A[i] == elem:
        A[i], A[last] = A[last], A[i]
        last -= 1
    else:
        i += 1
return last + 1
```

my-calendar-iii.py

```
# DESC
# Example 1:
# book(int start, int end)
# A K-booking happens when K events have some non-empty intersection (ie., there i
# s some time that is common to all K events.)
# Your class will have one method, book(int start, int end). Formally, this repres
# ents a booking on the half open interval [start, end), the range of real numbers
#  x such that start <= x < end.
# Implement a MyCalendarThree class to store your events. A new event can always b
# e added.
# For each call to the method MyCalendar.book, return an integer K representing th
# e largest integer such that there exists a K-booking in the calendar.
# Note:

# NOTE
# In calls to MyCalendarThree.book(start, end), start and end are integers in the
# range [0, 10^9].
# The number of calls to MyCalendarThree.book per test case will be at most 400.

# EXAMPLE
# MyCalendarThree();
# MyCalendarThree.book(10, 20); // returns 1
# MyCalendarThree.bo
# ok(50, 60); // returns 1
# MyCalendarThree.book(10, 40); // returns 2
# MyCalendarTh
# ree.book(5, 15); // returns 3
# MyCalendarThree.book(5, 10); // returns 3
# MyCalend
# arThree.book(25, 55); // returns 3
# Explanation:
# The first two events can be boo
# ked and are disjoint, so the maximum K-booking is a 1-booking.
# The third event [
# 10, 40) intersects the first event, and the maximum K-booking is a 2-booking.
# Th
# e remaining events cause the maximum K-booking to be only a 3-booking.
# Note that
#  the last event locally causes a 2-booking, but the answer is still 3 because
# eg
# . [10, 20), [10, 40), and [5, 15) are still triple booked.

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

import bisect


class MyCalendarThree(object):

    def __init__(self):
        self.__books = []


    def book(self, start, end):
        """
        :type start: int
        :type end: int
```

```
        :rtype: int
        """
        i = bisect.bisect_left(self.__books, (start, 1))
        if i < len(self.__books) and self.__books[i][0] == start:
            self.__books[i] = (self.__books[i][0], self.__books[i][1]+1)
        else:
            self.__books.insert(i, (start, 1))

        j = bisect.bisect_left(self.__books, (end, 1))
        if j < len(self.__books) and self.__books[j][0] == end:
            self.__books[j] = (self.__books[j][0], self.__books[j][1]-1)
        else:
            self.__books.insert(j, (end, -1))

        result, cnt = 0, 0
        for book in self.__books:
            cnt += book[1]
            result = max(result, cnt)
        return result
```

most-stones-removed-with-same-row-or-column.py

```
# DESC
# Example 3:
# Now, a move consists of removing a stone that shares a column or row with anothe
# r stone on the grid.
# Note:
# On a 2D plane, we place stones at some integer coordinate points.  Each coordina
# te point may have at most one stone.
# What is the largest possible number of moves we can make?
# Example 2:
# Example 1:

# NOTE
# 1 <= stones.length <= 1000
# 0 <= stones[i][j] < 10000

# EXAMPLE
# Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]
# Output: 3
# Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
# Output: 5
# Input: stones = [[0,0]]
# Output: 0

# 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})
```

invert-binary-tree.py

```
# DESC
# Example:
# Input:
# Invert a binary tree.
# Trivia:
#
# This problem was inspired by this original tweet by Max Howell:
# Output:

# NOTE
#

# EXAMPLE
# 4
#    /   \
#   2     7
#  / \   / \
# 1   3 6   9
# 4
#    /   \
#   7     2
#  / \   / \
# 9   6 3   1

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

import collections


# BFS solution.
class Queue(object):
    def __init__(self):
        self.data = collections.deque()

    def push(self, x):
        self.data.append(x)

    def peek(self):
        return self.data[0]

    def pop(self):
        return self.data.popleft()

    def size(self):
        return len(self.data)

    def empty(self):
        return len(self.data) == 0

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

class Solution(object):
```

```
    # @param {TreeNode} root
    # @return {TreeNode}
    def invertTree(self, root):
        if root is not None:
            nodes = Queue()
            nodes.push(root)
            while not nodes.empty():
                node = nodes.pop()
                node.left, node.right = node.right, node.left
                if node.left is not None:
                    nodes.push(node.left)
                if node.right is not None:
                    nodes.push(node.right)

        return root

# Time:  O(n)
# Space: O(h)
# Stack solution.
class Solution2(object):
    # @param {TreeNode} root
    # @return {TreeNode}
    def invertTree(self, root):
        if root is not None:
            nodes = []
            nodes.append(root)
            while nodes:
                node = nodes.pop()
                node.left, node.right = node.right, node.left
                if node.left is not None:
                    nodes.append(node.left)
                if node.right is not None:
                    nodes.append(node.right)

        return root

# Time:  O(n)
# Space: O(h)
# DFS, Recursive solution.
class Solution3(object):
    # @param {TreeNode} root
    # @return {TreeNode}
    def invertTree(self, root):
        if root is not None:
            root.left, root.right = self.invertTree(root.right), \
                                    self.invertTree(root.left)

        return root
```

minimum-subsequence-in-non-increasing-order.py

```python
# minimum-subsequence-in-non-increasing-order is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def minSubsequence(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        result, total, curr = [], sum(nums), 0
        nums.sort(reverse=True)
        for i, x in enumerate(nums):
            curr += x
            if curr > total-curr:
                break
        return nums[:i+1]
```

previous-permutation-with-one-swap.py

```python
# previous-permutation-with-one-swa is not found.
# 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
```

count-square-submatrices-with-all-ones.py

```
# DESC
# Example 2:
# Example 1:
# Given a m * n matrix of ones and zeros, return how many square submatrices have
# all ones.
# Constraints:

# NOTE
# 1 <= arr[0].length <= 300
# 0 <= arr[i][j] <= 1
# 1 <= arr.length <= 300

# EXAMPLE
# Input: matrix =
# [
#   [1,0,1],
#   [1,1,0],
#   [1,1,0]
# ]
# Output: 7
# Explanation:
# The
# re are 6 squares of side 1.
# There is 1 square of side 2.
# Total number of squa
# res = 6 + 1 = 7.
# Input: matrix =
# [
#   [0,1,1,1],
#   [1,1,1,1],
#   [0,1,1,1]
# ]
# Output: 15
# Explanation
# :
# There are 10 squares of side 1.
# There are 4 squares of side 2.
# There is  1 sq
# uare of side 3.
# Total number of squares = 10 + 4 + 1 = 15.

# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def countSquares(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: int
        """
        for i in xrange(1, len(matrix)):
            for j in xrange(1, len(matrix[0])):
                if not matrix[i][j]:
                    continue
                l = min(matrix[i-1][j], matrix[i][j-1])
                matrix[i][j] = l+1 if matrix[i-l][j-l] else l
        return sum(x for row in matrix for x in row)
```

plus-one-linked-list.py

```
# plus-one-linked-list is not found.
# Time:  O(n)
# Space: O(1)

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


# Two pointers solution.
class Solution(object):
    def plusOne(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head:
            return None

        dummy = ListNode(0)
        dummy.next = head

        left, right = dummy, head
        while right.next:
            if right.val != 9:
                left = right
            right = right.next

        if right.val != 9:
            right.val += 1
        else:
            left.val += 1
            right = left.next
            while right:
                right.val = 0
                right = right.next

        return dummy if dummy.val else dummy.next


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def plusOne(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        def reverseList(head):
            dummy = ListNode(0)
            curr = head
            while curr:
                dummy.next, curr.next, curr = curr, dummy.next, curr.next
            return dummy.next

        rev_head = reverseList(head)
        curr, carry = rev_head, 1
```

```
while curr and carry:
    curr.val += carry
    carry = curr.val / 10
    curr.val %= 10
    if carry and curr.next is None:
        curr.next = ListNode(0)
    curr = curr.next

return reverseList(rev_head)
```

palindrome-removal.py

```python
# palindrome-removal is not found.
# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def minimumMoves(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        dp = [[0 for _ in xrange(len(arr)+1)] for _ in xrange(len(arr)+1)]
        for l in xrange(1, len(arr)+1):
            for i in xrange(len(arr)-l+1):
                j = i+l-1
                if l == 1:
                    dp[i][j] = 1
                else:
                    dp[i][j] = 1+dp[i+1][j]
                    if arr[i] == arr[i+1]:
                        dp[i][j] = min(dp[i][j], 1+dp[i+2][j])
                    for k in xrange(i+2, j+1):
                        if arr[i] == arr[k]:
                            dp[i][j] = min(dp[i][j], dp[i+1][k-1] + dp[k+1][j])
        return dp[0][len(arr)-1]
```

read-n-characters-given-read4.py

```python
# read-n-characters-given-read4 is not found.
# Time:  O(n)
# Space: O(1)

def read4(buf):
    global file_content
    i = 0
    while i < len(file_content) and i < 4:
        buf[i] = file_content[i]
        i += 1

    if len(file_content) > 4:
        file_content = file_content[4:]
    else:
        file_content = ""
    return i

class Solution(object):
    def read(self, buf, n):
        """
        :type buf: Destination buffer (List[str])
        :type n: Maximum number of characters to read (int)
        :rtype: The number of characters read (int)
        """
        read_bytes = 0
        buffer = [''] * 4
        for i in xrange(n / 4 + 1):
            size = read4(buffer)
            if size:
                size = min(size, n-read_bytes)
                buf[read_bytes:read_bytes+size] = buffer[:size]
                read_bytes += size
            else:
                break
        return read_bytes
```

convert-bst-to-greater-tree.py

```
# DESC
# Given a Binary Search Tree (BST), convert it to a Greater Tree such that every k
# ey of the original BST is changed to the original key plus sum of all keys great
# er than the original key in BST.
# Note: This question is the same as 1038: https://leetcode.com/problems/binary-se
# arch-tree-to-greater-sum-tree/
# Example:

# NOTE
#

# EXAMPLE
# Input: The root of a Binary Search Tree like this:
#               5
#             /
#    \
#            2     13
#
# Output: The root of a Greater Tree like this:
#
#      18
#              /   \
#           20     13

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

class Solution(object):
    def convertBST(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        def convertBSTHelper(root, cur_sum):
            if not root:
                return cur_sum

            if root.right:
                cur_sum = convertBSTHelper(root.right, cur_sum)
            cur_sum += root.val
            root.val = cur_sum
            if root.left:
                cur_sum = convertBSTHelper(root.left, cur_sum)
            return cur_sum

        convertBSTHelper(root, 0)
        return root
```

balanced-binary-tree.py

```
# DESC
# Return false.
# Given a binary tree, determine if it is height-balanced.
# Given the following tree [1,2,2,3,3,null,null,4,4]:
# Given the following tree [3,9,20,null,null,15,7]:
# Return true.
#
#
#
# Example 2:
# Example 1:
# a binary tree in which the left and right subtrees of every node differ in heigh
# t by no more than 1.
# For this problem, a height-balanced binary tree is defined as:

# NOTE
#

# EXAMPLE
# 1
#       / \
#      2   2
#     / \
#    3   3
#   / \
#  4   4
# 3
#    / \
#   9  20
#     /  \
#    15   7

# 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 a boolean
    def isBalanced(self, root):
        def getHeight(root):
            if root is None:
                return 0
            left_height, right_height = \
                getHeight(root.left), getHeight(root.right)
            if left_height < 0 or right_height < 0 or \
               abs(left_height - right_height) > 1:
                return -1
            return max(left_height, right_height) + 1
        return (getHeight(root) >= 0)
```

spiral-matrix.py

```
# DESC
# Example 2:
# Example 1:
# Given a matrix of m x n elements (m rows, n columns), return all elements of the
#  matrix in spiral order.

# NOTE
#

# EXAMPLE
# 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]
# Input:
# [
#  [ 1, 2, 3 ],
#  [ 4, 5, 6 ],
#  [ 7, 8, 9 ]
# ]
# Output: [1,2,3,6,9,8,7,4,5]

# 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
```

long-pressed-name.py

```
# DESC
# Example 4:
# Example 2:
# Your friend is typing his name into a keyboard.  Sometimes, when typing a charac
# ter c, the key might get long pressed, and the character will be typed 1 or more
#  times.
# Constraints:
# typed
# Example 1:
# Example 3:
# You examine the typed characters of the keyboard.  Return True if it is possible
#  that it was your friends name, with some characters (possibly none) being long
# pressed.

# NOTE
# 1 <= name.length <= 1000
# The characters of name and typed are lowercase letters.
# 1 <= typed.length <= 1000

# EXAMPLE
# Input: name = "leelee", typed = "lleeelee"
# Output: true
# Input: name = "laiden", typed = "laiden"
# Output: true
# Explanation: It's not nece
# ssary to long press any character.
# Input: name = "saeed", typed = "ssaaedd"
# Output: false
# Explanation: 'e' must hav
# e been pressed twice, but it wasn't in the typed output.
# Input: name = "alex", typed = "aaleex"
# Output: true
# Explanation: 'a' and 'e' in
# 'alex' were long pressed.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isLongPressedName(self, name, typed):
        """
        :type name: str
        :type typed: str
        :rtype: bool
        """
        i = 0
        for j in xrange(len(typed)):
            if i < len(name) and name[i] == typed[j]:
                i += 1
            elif j == 0 or typed[j] != typed[j-1]:
                return False
        return i == len(name)
```

longest-univalue-path.py

```
# DESC
# Output: 2
# Example 2:
# Note: The given binary tree has not more than 10000 nodes. The height of the tre
# e is not more than 1000.
# The length of path between two nodes is represented by the number of edges betwe
# en them.
# Example 1:
# Output: 2
# Input:
# Given a binary tree, find the length of the longest path where each node in the
# path has the same value. This path may or may not pass through the root.
# Input:

# NOTE
#

# EXAMPLE
# 5
#              / \
#             4   5
#            / \   \
#           1   1   5
# 1
#              / \
#             4   5
#            / \   \
#           4   4   5

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

class Solution(object):
    def longestUnivaluePath(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result = [0]
        def dfs(node):
            if not node:
                return 0
            left, right = dfs(node.left), dfs(node.right)
            left = (left+1) if node.left and node.left.val == node.val else 0
            right = (right+1) if node.right and node.right.val == node.val else 0
            result[0] = max(result[0], left+right)
            return max(left, right)

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

invalid-transactions.py

```python
# invalid-transactions is not found.
# 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))
        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
```

encode-string-with-shortest-length.py

```python
# encode-string-with-shortest-length is not found.
# Time:  O(n^3) on average
# Space: O(n^2)

class Solution(object):
    def encode(self, s):
        """
        :type s: str
        :rtype: str
        """
        def encode_substr(dp, s, i, j):
            temp = s[i:j+1]
            pos = (temp + temp).find(temp, 1)  # O(n) on average
            if pos >= len(temp):
                return temp
            return str(len(temp)/pos) + '[' + dp[i][i + pos - 1] + ']'

        dp = [["" for _ in xrange(len(s))] for _ in xrange(len(s))]
        for length in xrange(1, len(s)+1):
            for i in xrange(len(s)+1-length):
                j = i+length-1
                dp[i][j] = s[i:i+length]
                for k in xrange(i, j):
                    if len(dp[i][k]) + len(dp[k+1][j]) < len(dp[i][j]):
                        dp[i][j] = dp[i][k] + dp[k+1][j]
                encoded_string = encode_substr(dp, s, i, j)
                if len(encoded_string) < len(dp[i][j]):
                    dp[i][j] = encoded_string
        return dp[0][len(s) - 1]
```

string-compression.py

```
# DESC
# Given an array of characters, compress it in-place.
# Follow up:
#
# Could you solve it using only O(1) extra space?
# After you are done modifying the input array in-place, return the new length of
# the array.
# Note:
# Every element of the array should be a character (not int) of length 1.
# Example 2:
# Example 3:
# The length after compression must always be smaller than or equal to the origina
# l array.
# Example 1:

# NOTE
# 1 <= len(chars) <= 1000.
# All characters have an ASCII value in [35, 126].

# EXAMPLE
# Input:
# ["a","a","b","b","c","c","c"]
#
# Output:
# Return 6, and the first 6 characte
# rs of the input array should be: ["a","2","b","2","c","3"]
#
# Explanation:
# "aa" is
#  replaced by "a2". "bb" is replaced by "b2". "ccc" is replaced by "c3".
# Input:
# ["a","b","b","b","b","b","b","b","b","b","b","b","b"]
#
# Output:
# Return 4,
# and the first 4 characters of the input array should be: ["a","b","1","2"].
#
# Exp
# lanation:
# Since the character "a" does not repeat, it is not compressed. "bbbbbb
# bbbbbb" is replaced by "b12".
# Notice each digit has it's own entry in the array.
# Input:
# ["a"]
#
# Output:
# Return 1, and the first 1 characters of the input array sh
# ould be: ["a"]
#
# Explanation:
# Nothing is replaced.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def compress(self, chars):
        """
```

```
        :type chars: List[str]
        :rtype: int
        """
        anchor, write = 0, 0
        for read, c in enumerate(chars):
            if read+1 == len(chars) or chars[read+1] != c:
                chars[write] = chars[anchor]
                write += 1
                if read > anchor:
                    n, left = read-anchor+1, write
                    while n > 0:
                        chars[write] = chr(n%10+ord('0'))
                        write += 1
                        n /= 10
                    right = write-1
                    while left < right:
                        chars[left], chars[right] = chars[right], chars[left]
                        left += 1
                        right -= 1
                anchor = read+1
        return write
```

alien-dictionary.py

```
# alien-dictionar is not found.
# Time:  O(n)
# Space: O(|V|+|E|) = O(26 + 26^2) = O(1)

import collections


# BFS solution.
class Solution(object):
    def alienOrder(self, words):
        """
        :type words: List[str]
        :rtype: str
        """
        result, in_degree, out_degree = [], {}, {}
        zero_in_degree_queue = collections.deque()
        nodes = set()
        for word in words:
            for c in word:
                nodes.add(c)

        for i in xrange(1, len(words)):
            if (len(words[i-1]) > len(words[i]) and
                    words[i-1][:len(words[i])] == words[i]):
                return ""
            self.findEdges(words[i - 1], words[i], in_degree, out_degree)

        for node in nodes:
            if node not in in_degree:
                zero_in_degree_queue.append(node)

        while zero_in_degree_queue:
            precedence = zero_in_degree_queue.popleft()
            result.append(precedence)

            if precedence in out_degree:
                for c in out_degree[precedence]:
                    in_degree[c].discard(precedence)
                    if not in_degree[c]:
                        zero_in_degree_queue.append(c)

                del out_degree[precedence]

        if out_degree:
            return ""

        return "".join(result)

    # Construct the graph.
    def findEdges(self, word1, word2, in_degree, out_degree):
        str_len = min(len(word1), len(word2))
        for i in xrange(str_len):
            if word1[i] != word2[i]:
                if word2[i] not in in_degree:
                    in_degree[word2[i]] = set()
                if word1[i] not in out_degree:
                    out_degree[word1[i]] = set()
                in_degree[word2[i]].add(word1[i])
```

```
                out_degree[word1[i]].add(word2[i])
                break


# DFS solution.
class Solution2(object):
    def alienOrder(self, words):
        """
        :type words: List[str]
        :rtype: str
        """
        # Find ancestors of each node by DFS.
        nodes, ancestors = set(), {}
        for i in xrange(len(words)):
            for c in words[i]:
                nodes.add(c)
        for node in nodes:
            ancestors[node] = []
        for i in xrange(1, len(words)):
            if (len(words[i-1]) > len(words[i]) and
                    words[i-1][:len(words[i])] == words[i]):
                return ""
            self.findEdges(words[i - 1], words[i], ancestors)

        # Output topological order by DFS.
        result = []
        visited = {}
        for node in nodes:
            if self.topSortDFS(node, node, ancestors, visited, result):
                return ""

        return "".join(result)

    # Construct the graph.
    def findEdges(self, word1, word2, ancestors):
        min_len = min(len(word1), len(word2))
        for i in xrange(min_len):
            if word1[i] != word2[i]:
                ancestors[word2[i]].append(word1[i])
                break

    # Topological sort, return whether there is a cycle.
    def topSortDFS(self, root, node, ancestors, visited, result):
        if node not in visited:
            visited[node] = root
            for ancestor in ancestors[node]:
                if self.topSortDFS(root, ancestor, ancestors, visited, result):
                    return True
            result.append(node)
        elif visited[node] == root:
            # Visited from the same root in the DFS path.
            # So it is cyclic.
            return True
        return False
```

counting-bits.py

```python
# DESC
# 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:
# Follow up:
# Example 2:

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

# EXAMPLE
# Input: 2
# Output: [0,1,1]
# Input: 5
# Output: [0,1,1,2,1,2]

# 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

```
# DESC
# M = 1
# Example 1:
# Alex and Lee continue their games with piles of stones.  There are a number of p
# iles arranged in a row, and each pile has a positive integer number of stones pi
# les[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 remain
# ing piles, where 1 <= X <= 2M.  Then, we set M = max(M, X).
# Constraints:
# The game continues until all the stones have been taken.
# Assuming Alex and Lee play optimally, return the maximum number of stones Alex c
# an get.

# NOTE
# 1 <= piles[i] <= 10 ^ 4
# 1 <= piles.length <= 100

# EXAMPLE
# Input: piles = [2,7,9,4,4]
# Output: 10
# Explanation:  If Alex takes one pile at th
# e 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.

# 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)
```

merge-two-sorted-lists.py

```
# DESC
# Merge two sorted linked lists and return it as a new sorted list. The new list s
# hould be made by splicing together the nodes of the first two lists.
# Example:

# NOTE
#

# EXAMPLE
# Input: 1->2->4, 1->3->4
# Output: 1->1->2->3->4->4

# 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):
    def mergeTwoLists(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        curr = dummy = ListNode(0)
        while l1 and l2:
            if l1.val < l2.val:
                curr.next = l1
                l1 = l1.next
            else:
                curr.next = l2
                l2 = l2.next
            curr = curr.next
        curr.next = l1 or l2
        return dummy.next
```

minimize-malware-spread.py

```
# DESC
# We will remove one node from the initial list.  Return the node that if removed,
#  would minimize M(initial).  If multiple nodes could be removed to minimize M(in
# itial), return such a node with the smallest index.
# In a network of nodes, each node i is directly connected to another node j if an
# d only if graph[i][j] = 1.
# Example 2:
# initial
# Note:
# Some nodes initial are initially infected by malware.  Whenever two nodes are di
# rectly connected and at least one of those two nodes is infected by malware, bot
# h nodes will be infected by malware.  This spread of malware will continue until
#  no more nodes can be infected in this manner.
# Note that if a node was removed from the initial list of infected nodes, it may
# still be infected later as a result of the malware spread.
# Example 3:
# Suppose M(initial) is the final number of nodes infected with malware in the ent
# ire network, after the spread of malware stops.
# Example 1:

# NOTE
# 1 <= initial.length <= graph.length
# 0 <= graph[i][j] == graph[j][i] <= 1
# graph[i][i] == 1
# 1 < graph.length = graph[0].length <= 300
# 0 <= initial[i] < graph.length

# EXAMPLE
# Input: graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]
# Output: 1
# Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]
# Output: 0
# Input: graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]
# Output: 0

# Time:  O(n^2)
# 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:
            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 minMalwareSpread(self, graph, initial):
        """
        :type graph: List[List[int]]
        :type initial: List[int]
        :rtype: int
        """
        union_find = UnionFind(len(graph))
        for i in xrange(len(graph)):
            for j in xrange(i+1, len(graph)):
                if graph[i][j] == 1:
                    union_find.union_set(i, j)
        union_size = collections.Counter(union_find.find_set(i) for i in xrange(len(graph)))
        malware_count = collections.Counter(union_find.find_set(i) for i in initial)
        return min(initial, key=lambda x: [malware_count[union_find.find_set(x)] > 1,
                                           -union_size[union_find.find_set(x)],
                                           x])
```

lru-cache.py

```
# lru-cache is not found.
# 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:
            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

```
# DESC
# Constraints:
# Example 3:
# Example 4:
# 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.
# Example 2:
# Example 1:
# Since the answer may be too large, return it modulo 10^9 + 7.

# NOTE
# Each node's value is between [1, 10000].
# Each tree has at most 50000 nodes and at least 2 nodes.

# EXAMPLE
# 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)
# Input: root = [1,1]
# Output: 1
# Input: root = [1,null,2,3,4,null,null,5,6]
# Output: 90
# Explanation:  Remove the r
# ed edge and get 2 binary trees with sum 15 and 6.Their product is 90 (15*6)
# Input: root = [2,3,9,10,7,8,6,5,4,11,1]
# Output: 1025

# 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
        :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
```

largest-triangle-area.py

```
# DESC
# Notes:
# You have a list of points in the plane. Return the area of the largest triangle
# that can be formed by any 3 of the points.

# NOTE
# 3 <= points.length <= 50.
# No points will be duplicated.
# -50 <= points[i][j] <= 50.
# Answers within 10^-6 of the true value will be accepted as correct.

# EXAMPLE
# Example:
# Input: points = [[0,0],[0,1],[1,0],[0,2],[2,0]]
# Output: 2
# Explanation:
#
# The five points are show in the figure below. The red triangle is the largest.

# Time:  O(n^3)
# Space: O(1)


class Solution(object):
    def largestTriangleArea(self, points):
        """
        :type points: List[List[int]]
        :rtype: float
        """
        result = 0
        for i in xrange(len(points)-2):
            for j in xrange(i+1, len(points)-1):
                for k in xrange(j+1, len(points)):
                    result = max(result,
                                 0.5 * abs(points[i][0] * points[j][1] +
                                           points[j][0] * points[k][1] +
                                           points[k][0] * points[i][1] -
                                           points[j][0] * points[i][1] -
                                           points[k][0] * points[j][1] -
                                           points[i][0] * points[k][1]))
        return result
```

valid-tic-tac-toe-state.py

```
# DESC
# board
# 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-ta
# c-toe game.
# Note:
# Here are the rules of Tic-Tac-Toe:
# The board is a 3 x 3 array, and consists of characters " ", "X", and "O".  The "
#  " character represents an empty square.

# NOTE
# No more moves can be played if the game is over.
# The game ends when there are 3 of the same (non-empty) character filling any row
# , column, or diagonal.
# "X" and "O" characters are always placed into empty squares, never filled ones.
# The first player always places "X" characters, while the second player always pl
# aces "O" characters.
# The game also ends if all squares are non-empty.
# Each board[i][j] is a character in the set {" ", "X", "O"}.
# board is a length-3 array of strings, where each string board[i] has length 3.
# Players take turns placing characters into empty squares (" ").

# EXAMPLE
# Example 1:
# Input: board = ["O  ", "   ", "   "]
# Output: false
# Explanation: The f
# irst player always plays "X".
#
# Example 2:
# Input: board = ["XOX", " X ", "   "]
# O
# utput: false
# Explanation: Players take turns making moves.
#
# Example 3:
# Input: bo
# ard = ["XXX", "   ", "OOO"]
# Output: false
#
# Example 4:
# Input: board = ["XOX", "O
# O", "XOX"]
# Output: true

# 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
```

friends-of-appropriate-ages.py

```
# DESC
# Notes:
# Note that if A requests B, B does not necessarily request A.  Also, people will
# not friend request themselves.
# Example 3:
# Otherwise, A will friend request B.
# Person A will NOT friend request person B (B != A) if any of the following condi
# tions are true:
# How many total friend requests are made?
# Some people will make friend requests. The list of their ages is given and ages[
# i] is the age of the ith person.
# Example 2:
# Example 1:

# NOTE
# 1 <= ages[i] <= 120.
# 1 <= ages.length <= 20000.
# age[B] > 100 && age[A] < 100
# age[B] <= 0.5 * age[A] + 7
# age[B] > age[A]

# EXAMPLE
# Input: [16,17,18]
# Output: 2
# Explanation: Friend requests are made 17 -> 16, 18 -> 17.
# Input: [16,16]
# Output: 2
# Explanation: 2 people friend request each other.
# Input: [20,30,100,110,120]
# Output: 3
# Explanation: Friend requests are made 110 -
# > 100, 120 -> 110, 120 -> 100.

# Time:  O(a^2 + n), a is the number of ages,
#                    n is the number of people
# Space: O(a)

import collections


class Solution(object):
    def numFriendRequests(self, ages):
        """
        :type ages: List[int]
        :rtype: int
        """
        def request(a, b):
            return 0.5*a+7 < b <= a

        c = collections.Counter(ages)
        return sum(int(request(a, b)) * c[a]*(c[b]-int(a == b))
                   for a in c
                   for b in c)
```

decrease-elements-to-make-array-zigzag.py

```
# DESC
# Example 2:
# Given an array nums of integers, a move consists of choosing any element and dec
# reasing it by 1.
# Example 1:
# An array A is a zigzag array if either:
# Return the minimum number of moves to transform the given array nums into a zigz
# ag array.
# Constraints:

# NOTE
# OR, every odd-indexed element is greater than adjacent elements, ie. A[0] < A[1]
#  > A[2] < A[3] > A[4] < ...
# Every even-indexed element is greater than adjacent elements, ie. A[0] > A[1] <
# A[2] > A[3] < A[4] > ...
# 1 <= nums[i] <= 1000
# 1 <= nums.length <= 1000

# EXAMPLE
# Input: nums = [9,6,1,6,2]
# Output: 4
# Input: nums = [1,2,3]
# Output: 2
# Explanation: We can decrease 2 to 0 or 3 to 1.

# 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)
```

word-squares.py

```
# word-squares is not found.
# Time:  O(n^2 * n!)
# Space: O(n^2)

class TrieNode(object):
    def __init__(self):
        self.indices = []
        self.children = [None] * 26

    def insert(self, words, i):
        cur = self
        for c in words[i]:
            if not cur.children[ord(c)-ord('a')]:
                cur.children[ord(c)-ord('a')] = TrieNode()
            cur = cur.children[ord(c)-ord('a')]
            cur.indices.append(i)


class Solution(object):
    def wordSquares(self, words):
        """
        :type words: List[str]
        :rtype: List[List[str]]
        """
        result = []

        trie = TrieNode()
        for i in xrange(len(words)):
            trie.insert(words, i)

        curr = []
        for s in words:
            curr.append(s)
            self.wordSquaresHelper(words, trie, curr, result)
            curr.pop()

        return result

    def wordSquaresHelper(self, words, trie, curr, result):
        if len(curr) >= len(words[0]):
            return result.append(list(curr))

        node = trie
        for s in curr:
            node = node.children[ord(s[len(curr)]) - ord('a')]
            if not node:
                return

        for i in node.indices:
            curr.append(words[i])
            self.wordSquaresHelper(words, trie, curr, result)
            curr.pop()
```

is-graph-bipartite.py

```
# is-graph-bipartite is not found.
# Time:  O(|V| + |E|)
# Space: O(|V|)

class Solution(object):
    def isBipartite(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: bool
        """
        color = {}
        for node in xrange(len(graph)):
            if node in color:
                continue
            stack = [node]
            color[node] = 0
            while stack:
                curr = stack.pop()
                for neighbor in graph[curr]:
                    if neighbor not in color:
                        stack.append(neighbor)
                        color[neighbor] = color[curr] ^ 1
                    elif color[neighbor] == color[curr]:
                        return False
        return True
```

maximum-xor-of-two-numbers-in-an-array.py

```
# maximum-xor-of-two-numbers-in-an-arra is not found.
# 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
```

shift-2d-grid.py

```python
# DESC
# Example 2:
# In one shift operation:
# Constraints:
# Return the 2D grid after applying shift operation k times.
# Given a 2D grid of size m x n and an integer k. You need to shift the grid k times.
# Example 3:
# Example 1:

# NOTE
# 1 <= m <= 50
# -1000 <= grid[i][j] <= 1000
# Element at grid[m - 1][n - 1] moves to grid[0][0].
# Element at grid[i][j] moves to grid[i][j + 1].
# 1 <= n <= 50
# n == grid[i].length
# Element at grid[i][n - 1] moves to grid[i + 1][0].
# 0 <= k <= 100
# m == grid.length

# EXAMPLE
# Input: grid = [[3,8,1,9],[19,7,2,5],[4,6,11,10],[12,0,21,13]], k = 4
# Output: [[1
# 2,0,21,13],[3,8,1,9],[19,7,2,5],[4,6,11,10]]
# Input: grid = [[1,2,3],[4,5,6],[7,8,9]], k = 9
# Output: [[1,2,3],[4,5,6],[7,8,9]]
# Input: grid = [[1,2,3],[4,5,6],[7,8,9]], k = 1
# Output: [[9,1,2],[3,4,5],[6,7,8]]

# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def shiftGrid(self, grid, k):
        """
        :type grid: List[List[int]]
        :type k: int
        :rtype: List[List[int]]
        """
        def rotate(grids, k):
            def reverse(grid, start, end):
                while start < end:
                    start_r, start_c = divmod(start, len(grid[0]))
                    end_r, end_c = divmod(end-1, len(grid[0]))
                    grid[start_r][start_c], grid[end_r][end_c] = grid[end_r][end_c], grid[start_r][start_c]
                    start += 1
                    end -= 1

            k %= len(grid)*len(grid[0])
            reverse(grid, 0, len(grid)*len(grid[0]))
            reverse(grid, 0, k)
            reverse(grid, k, len(grid)*len(grid[0]))

        rotate(grid, k)
        return grid
```

largest-bst-subtree.py

```
# largest-bst-subtree is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def largestBSTSubtree(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root is None:
            return 0

        max_size = [1]
        def largestBSTSubtreeHelper(root):
            if root.left is None and root.right is None:
                return 1, root.val, root.val

            left_size, left_min, left_max = 0, root.val, root.val
            if root.left is not None:
                left_size, left_min, left_max = largestBSTSubtreeHelper(root.left)

            right_size, right_min, right_max = 0, root.val, root.val
            if root.right is not None:
                right_size, right_min, right_max = largestBSTSubtreeHelper(root.right)

            size = 0
            if (root.left is None or left_size > 0) and \
               (root.right is None or right_size > 0) and \
               left_max <= root.val <= right_min:
                size = 1 + left_size + right_size
                max_size[0] = max(max_size[0], size)

            return size, left_min, right_max

        largestBSTSubtreeHelper(root)
        return max_size[0]
```

accounts-merge.py

```
# DESC
# Example 1:
# Given a list accounts, each element accounts[i] is a list of strings, where the
# first element accounts[i][0] is a name, and the rest of the elements are emails
# representing emails of the account.
# Now, we would like to merge these accounts.  Two accounts definitely belong to t
# he same person if there is some email that is common to both accounts.  Note tha
# t even if two accounts have the same name, they may belong to different people a
# s people could have the same name.  A person can have any number of accounts ini
# tially, but all of their accounts definitely have the same name.
# Note:
# After merging the accounts, return the accounts in the following format: the fir
# st element of each account is the name, and the rest of the elements are emails
# in sorted order.  The accounts themselves can be returned in any order.

# NOTE
# The length of accounts[i] will be in the range [1, 10].
# The length of accounts will be in the range [1, 1000].
# The length of accounts[i][j] will be in the range [1, 30].

# EXAMPLE
# Input:
# accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John",
# "johnnybravo@mail.com"], ["John", "johnsmith@mail.com", "john_newyork@mail.com"]
# , ["Mary", "mary@mail.com"]]
# Output: [["John", 'john00@mail.com', 'john_newyork@
# mail.com', 'johnsmith@mail.com'],  ["John", "johnnybravo@mail.com"], ["Mary", "m
# ary@mail.com"]]
# Explanation:
# The first and third John's are the same person as
# they have the common email "johnsmith@mail.com".
# The second John and Mary are di
# fferent people as none of their email addresses are used by other accounts.
# We c
# ould return these lists in any order, for example the answer [['Mary', 'mary@mai
# l.com'], ['John', 'johnnybravo@mail.com'],
# ['John', 'john00@mail.com', 'john_ne
# wyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

# Time:  O(nlogn), n is the number of total emails,
#                  and the max length ofemail is 320, p.s. {64}@{255}
# Space: O(n)

import collections


class UnionFind(object):
    def __init__(self):
        self.set = []

    def get_id(self):
        self.set.append(len(self.set))
        return len(self.set)-1

    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)


class Solution(object):
    def accountsMerge(self, accounts):
        """
        :type accounts: List[List[str]]
        :rtype: List[List[str]]
        """
        union_find = UnionFind()
        email_to_name = {}
        email_to_id = {}
        for account in accounts:
            name = account[0]
            for i in xrange(1, len(account)):
                if account[i] not in email_to_id:
                    email_to_name[account[i]] = name
                    email_to_id[account[i]] = union_find.get_id()
                union_find.union_set(email_to_id[account[1]],
                                     email_to_id[account[i]])

        result = collections.defaultdict(list)
        for email in email_to_name.keys():
            result[union_find.find_set(email_to_id[email])].append(email)
        for emails in result.values():
            emails.sort()
        return [[email_to_name[emails[0]]] + emails
                for emails in result.values()]
```

implement-queue-using-stacks.py

```
# DESC
# Example:
# Notes:
# Implement the following operations of a queue using stacks.

# NOTE
# You may assume that all operations are valid (for example, no pop or peek operat
# ions will be called on an empty queue).
# empty() -- Return whether the queue is empty.
# peek() -- Get the front element.
# pop() -- Removes the element from in front of queue.
# Depending on your language, stack may not be supported natively. You may simulat
# e a stack by using a list or deque (double-ended queue), as long as you use only
#  standard operations of a stack.
# You must use only standard operations of a stack -- which means only push to top
# , peek/pop from top, size, and is empty operations are valid.
# push(x) -- Push element x to the back of queue.

# EXAMPLE
# MyQueue queue = new MyQueue();
#
# queue.push(1);
# queue.push(2);
# queue.peek();  /
# / returns 1
# queue.pop();   // returns 1
# queue.empty(); // returns false

# Time:  O(1), amortized
# Space: O(n)

class Queue(object):
    # initialize your data structure here.
    def __init__(self):
        self.A, self.B = [], []

    # @param x, an integer
    # @return nothing
    def push(self, x):
        self.A.append(x)

    # @return an integer
    def pop(self):
        self.peek()
        return self.B.pop()

    # @return an integer
    def peek(self):
        if not self.B:
            while self.A:
                self.B.append(self.A.pop())
        return self.B[-1]

    # @return an boolean
    def empty(self):
        return not self.A and not self.B
```

arithmetic-slices-ii-subsequence.py

```
# arithmetic-slices-ii-subsequence is not found.
# Time:  O(n^2)
# Space: O(n * d)

import collections


class Solution(object):
    def numberOfArithmeticSlices(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result = 0
        dp = [collections.defaultdict(int) for i in xrange(len(A))]
        for i in xrange(1, len(A)):
            for j in xrange(i):
                diff = A[i]-A[j]
                dp[i][diff] += 1
                if diff in dp[j]:
                    dp[i][diff] += dp[j][diff]
                    result += dp[j][diff]
        return result
```

longest-common-subsequence.py

```
# DESC
# A subsequence of a string is a new string generated from the original string wit
# h some characters(can be none) deleted without changing the relative order of th
# e remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is no
# t). A common subsequence of two strings is a subsequence that is common to both
# strings.
# Constraints:
# If there is no common subsequence, return 0.
# Example 2:
# Example 1:
# Given two strings text1 and text2, return the length of their longest common sub
# sequence.
# Example 3:

# NOTE
# 1 <= text2.length <= 1000
# 1 <= text1.length <= 1000
# The input strings consist of lowercase English characters only.

# EXAMPLE
# Input: text1 = "abc", text2 = "def"
# Output: 0
# Explanation: There is no such comm
# on subsequence, so the result is 0.
# Input: text1 = "abcde", text2 = "ace"
# Output: 3
# Explanation: The longest comm
# on subsequence is "ace" and its length is 3.
# Input: text1 = "abc", text2 = "abc"
# Output: 3
# Explanation: The longest common su
# bsequence is "abc" and its length is 3.

# 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

```
# DESC
# Given a binary tree, return the inorder traversal of its nodes' values.
# Follow up: Recursive solution is trivial, could you do it iteratively?
# Example:

# NOTE
#

# EXAMPLE
# Input: [1,null,2,3]
#    1
#     \
#      2
#     /
#    3
#
# Output: [1,3,2]

# 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)
```

```python
# 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

```
# sort-the-matrix-diagonall is not found.
# 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
```

binary-tree-paths.py

```
# DESC
# Example:
# Note: A leaf is a node with no children.
# Given a binary tree, return all root-to-leaf paths.

# NOTE
#

# EXAMPLE
# Input:
#
#    1
#  /   \
# 2     3
#  \
#   5
#
# Output: ["1->2->5", "1->3"]
#
# Explanation: Al
# l root-to-leaf paths are: 1->2->5, 1->3

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

class Solution(object):
    # @param {TreeNode} root
    # @return {string[]}
    def binaryTreePaths(self, root):
        result, path = [], []
        self.binaryTreePathsRecu(root, path, result)
        return result

    def binaryTreePathsRecu(self, node, path, result):
        if node is None:
            return

        if node.left is node.right is None:
            ans = ""
            for n in path:
                ans += str(n.val) + "->"
            result.append(ans + str(node.val))

        if node.left:
            path.append(node)
            self.binaryTreePathsRecu(node.left, path, result)
            path.pop()

        if node.right:
            path.append(node)
            self.binaryTreePathsRecu(node.right, path, result)
            path.pop()
```

delete-tree-nodes.py

```python
# delete-tree-nodes is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def deleteTreeNodes(self, nodes, parent, value):
        """
        :type nodes: int
        :type parent: List[int]
        :type value: List[int]
        :rtype: int
        """
        def dfs(value, children, x):
            total, count = value[x], 1
            for y in children[x]:
                t, c = dfs(value, children, y)
                total += t
                count += c if t else 0
            return total, count if total else 0

        children = collections.defaultdict(list)
        for i, p in enumerate(parent):
            if i:
                children[p].append(i)
        return dfs(value, children, 0)[1]


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def deleteTreeNodes(self, nodes, parent, value):
        """
        :type nodes: int
        :type parent: List[int]
        :type value: List[int]
        :rtype: int
        """
        # assuming parent[i] < i for all i > 0
        result = [1]*nodes
        for i in reversed(xrange(1, nodes)):
            value[parent[i]] += value[i]
            result[parent[i]] += result[i] if value[i] else 0
        return result[0]
```

closest-binary-search-tree-value.py

```python
# closest-binary-search-tree-value is not found.
# Time:  O(h)
# Space: O(1)

class Solution(object):
    def closestValue(self, root, target):
        """
        :type root: TreeNode
        :type target: float
        :rtype: int
        """
        gap = float("inf")
        closest = float("inf")
        while root:
            if abs(root.val - target) < gap:
                gap = abs(root.val - target)
                closest = root.val
            if target == root.val:
                break
            elif target < root.val:
                root = root.left
            else:
                root = root.right
        return closest
```

find-the-smallest-divisor-given-a-threshold.py

```python
# find-the-smallest-divisor-given-a-threshold is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def smallestDivisor(self, nums, threshold):
        """
        :type nums: List[int]
        :type threshold: int
        :rtype: int
        """
        def check(A, d, threshold):
            return sum((i-1)//d+1 for i in nums) <= threshold

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

smallest-good-base.py

```
# DESC
# Now given a string representing n, you should return the smallest good base of n
#  in string format.
# Example 1:
# Example 2:
# Example 3:
# Note:
# For an integer n, we call k>=2 a good base of n, if all digits of n base k are 1.

# NOTE
# The range of n is [3, 10^18].
# The string representing n is always valid and will not have leading zeros.

# EXAMPLE
# Input: "13"
# Output: "3"
# Explanation: 13 base 3 is 111.
# Input: "1000000000000000000"
# Output: "999999999999999999"
# Explanation: 100000000
# 0000000000 base 999999999999999999 is 11.
# Input: "4681"
# Output: "8"
# Explanation: 4681 base 8 is 11111.

# Time:  O(logn * log(logn))
# Space: O(1)

import math


class Solution(object):
    def smallestGoodBase(self, n):
        """
        :type n: str
        :rtype: str
        """
        num = int(n)
        max_len = int(math.log(num,2))
        for l in xrange(max_len, 1, -1):
            b = int(num ** (l**-1))
            if (b**(l+1)-1) // (b-1) == num:
                return str(b)
        return str(num-1)
```

count-servers-that-communicate.py

```python
# count-servers-that-communicate is not found.
# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    def countServers(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        rows, cols = [0]*len(grid), [0]*len(grid[0])
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                if grid[i][j]:
                    rows[i] += 1
                    cols[j] += 1
        result = 0
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                if grid[i][j] and (rows[i] > 1 or cols[j] > 1):
                    result += 1
        return result
```

count-of-range-sum.py

```
# DESC
# Given an integer array nums, return the number of range sums that lie in [lower,
#  upper] inclusive.
#
# Range sum S(i, j) is defined as the sum of the elements in n
# ums between indices i and j (i  j), inclusive.
# Constraints:
# Note:
#
# A naive algorithm of O(n2) is trivial. You MUST do better than that.
# Example:

# NOTE
# 0 <= nums.length <= 10^4

# EXAMPLE
# Input: nums = [-2,5,-1], lower = -2, upper = 2,
# Output: 3
# Explanation: The thre
# e ranges are : [0,0], [2,2], [0,2] and their respective sums are: -2, -1, 2.

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

class Solution(object):
    def countRangeSum(self, nums, lower, upper):
        """
        :type nums: List[int]
        :type lower: int
        :type upper: int
        :rtype: int
        """
        def countAndMergeSort(sums, start, end, lower, upper):
            if end - start <= 1:  # The size of range [start, end) less than 2 is always with count 0.
                return 0
            mid = start + (end - start) / 2
            count = countAndMergeSort(sums, start, mid, lower, upper) + \
                    countAndMergeSort(sums, mid, end, lower, upper)
            j, k, r = mid, mid, mid
            tmp = []
            for i in xrange(start, mid):
                # Count the number of range sums that lie in [lower, upper].
                while k < end and sums[k] - sums[i] < lower:
                    k += 1
                while j < end and sums[j] - sums[i] <= upper:
                    j += 1
                count += j - k

                # Merge the two sorted arrays into tmp.
                while r < end and sums[r] < sums[i]:
                    tmp.append(sums[r])
                    r += 1
                tmp.append(sums[i])
            # Copy tmp back to sums.
            sums[start:start+len(tmp)] = tmp
            return count

        sums = [0] * (len(nums) + 1)
```

```
        for i in xrange(len(nums)):
            sums[i + 1] = sums[i] + nums[i]
        return countAndMergeSort(sums, 0, len(sums), lower, upper)


# Divide and Conquer solution.
class Solution2(object):
    def countRangeSum(self, nums, lower, upper):
        """
        :type nums: List[int]
        :type lower: int
        :type upper: int
        :rtype: int
        """
        def countAndMergeSort(sums, start, end, lower, upper):
            if end - start <= 0:  # The size of range [start, end] less than 2 is always with count 0.
                return 0

            mid = start + (end - start) / 2
            count = countAndMergeSort(sums, start, mid, lower, upper) + \
                    countAndMergeSort(sums, mid + 1, end, lower, upper)
            j, k, r = mid + 1, mid + 1, mid + 1
            tmp = []
            for i in xrange(start, mid + 1):
                # Count the number of range sums that lie in [lower, upper].
                while k <= end and sums[k] - sums[i] < lower:
                    k += 1
                while j <= end and sums[j] - sums[i] <= upper:
                    j += 1
                count += j - k

                # Merge the two sorted arrays into tmp.
                while r <= end and sums[r] < sums[i]:
                    tmp.append(sums[r])
                    r += 1
                tmp.append(sums[i])

            # Copy tmp back to sums
            sums[start:start+len(tmp)] = tmp
            return count

        sums = [0] * (len(nums) + 1)
        for i in xrange(len(nums)):
            sums[i + 1] = sums[i] + nums[i]
        return countAndMergeSort(sums, 0, len(sums) - 1, lower, upper)
```

largest-number-at-least-twice-of-others.py

```python
# DESC
# In a given integer array nums, there is always exactly one largest element.
# If it is, return the index of the largest element, otherwise return -1.
# Find whether the largest element in the array is at least twice as much as every
#  other number in the array.
# Note:
# Example 2:
# Example 1:

# NOTE
# Every nums[i] will be an integer in the range [0, 99].
# nums will have a length in the range [1, 50].

# EXAMPLE
# Input: nums = [3, 6, 1, 0]
# Output: 1
# Explanation: 6 is the largest integer, and
# for every other number in the array x,
# 6 is more than twice as big as x.  The in
# dex of value 6 is 1, so we return 1.
# Input: nums = [1, 2, 3, 4]
# Output: -1
# Explanation: 4 isn't at least as big as tw
# ice the value of 3, so we return -1.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def dominantIndex(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        m = max(nums)
        if all(m >= 2*x for x in nums if x != m):
            return nums.index(m)
        return -1
```

buddy-strings.py

```
# DESC
# Example 2:
# Example 3:
# Example 4:
# Constraints:
# Given two strings A and B of lowercase letters, return true if and only if we ca
# n swap two letters in A so that the result equals B.
# Example 1:
# Example 5:

# NOTE
# 0 <= B.length <= 20000
# 0 <= A.length <= 20000
# A and B consist only of lowercase letters.

# EXAMPLE
# Input: A = "", B = "aa"
# Output: false
# Input: A = "ab", B = "ab"
# Output: false
# Input: A = "aaaaaaabc", B = "aaaaaaacb"
# Output: true
# Input: A = "aa", B = "aa"
# Output: true
# Input: A = "ab", B = "ba"
# Output: true

# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def buddyStrings(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: bool
        """
        if len(A) != len(B):
            return False
        diff = []
        for a, b in itertools.izip(A, B):
            if a != b:
                diff.append((a, b))
                if len(diff) > 2:
                    return False
        return (not diff and len(set(A)) < len(A)) or \
               (len(diff) == 2 and diff[0] == diff[1][::-1])
```

insert-delete-getrandom-o1-duplicates-allowed.py

```
# insert-delete-getrandom-o1-duplicates-allowed is not found.
# Time:  O(1)
# Space: O(n)

from random import randint
from collections import defaultdict

class RandomizedCollection(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__list = []
        self.__used = defaultdict(list)


    def insert(self, val):
        """
        Inserts a value to the collection. Returns true if the collection did not already contain the specifie
        :type val: int
        :rtype: bool
        """
        has = val in self.__used

        self.__list += (val, len(self.__used[val])),
        self.__used[val] += len(self.__list)-1,

        return not has


    def remove(self, val):
        """
        Removes a value from the collection. Returns true if the collection contained the specified element.
        :type val: int
        :rtype: bool
        """
        if val not in self.__used:
            return False

        self.__used[self.__list[-1][0]][self.__list[-1][1]] = self.__used[val][-1]
        self.__list[self.__used[val][-1]], self.__list[-1] = self.__list[-1], self.__list[self.__used[val][-1]]

        self.__used[val].pop()
        if not self.__used[val]:
            self.__used.pop(val)
        self.__list.pop()

        return True

    def getRandom(self):
        """
        Get a random element from the collection.
        :rtype: int
        """
        return self.__list[randint(0, len(self.__list)-1)][0]
```

the-k-th-lexicographical-string-of-all-happy-strings-of-length-n.py

```
# the-k-th-lexicographical-string-of-all-happy-strings-of-length-n is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def getHappyString(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        base = 2**(n-1)
        if k > 3*base:
            return ""
        result = [chr(ord('a')+(k-1)//base)]
        while base > 1:
            k -= (k-1)//base*base
            base //= 2
            result.append(('a' if result[-1] != 'a' else 'b') if (k-1)//base == 0 else
                          ('c' if result[-1] != 'c' else 'b'))
        return "".join(result)
```

dice-roll-simulation.py

```python
# dice-roll-simulation is not found.
# Time:  O(m * n), m is the max of rollMax
# Space: O(m)

class Solution(object):
    def dieSimulator(self, n, rollMax):
        """
        :type n: int
        :type rollMax: List[int]
        :rtype: int
        """
        MOD = 10**9+7
        def sum_mod(array):
            return reduce(lambda x, y: (x+y)%MOD, array)

        dp = [[1] + [0]*(rollMax[i]-1) for i in xrange(6)]  # 0-indexed
        for _ in xrange(n-1):
            new_dp = [[0]*rollMax[i] for i in xrange(6)]
            for i in xrange(6):
                for k in xrange(rollMax[i]):
                    for j in xrange(6):
                        if i == j:
                            if k < rollMax[i]-1:  # 0-indexed
                                new_dp[j][k+1] = (new_dp[j][k+1]+dp[i][k])%MOD
                        else:
                            new_dp[j][0] = (new_dp[j][0]+dp[i][k])%MOD
            dp = new_dp
        return sum_mod(sum_mod(row) for row in dp)
```

flip-binary-tree-to-match-preorder-traversal.py

```
# DESC
# Our goal is to flip the least number of nodes in the tree so that the voyage of
# the tree matches the voyage we are given.
# Consider the sequence of N values reported by a preorder traversal starting from
#  the root.  Call such a sequence of N values the voyage of the tree.
# If we cannot do so, then return the list [-1].
# Given a binary tree with N nodes, each node has a different value from {1, ..., N}.
# Example 3:
# (Recall that a preorder traversal of a node means we report the current node's v
# alue, then preorder-traverse the left child, then preorder-traverse the right ch
# ild.)
# Example 2:
# If we can do so, then return a list of the values of all nodes flipped.  You may
#  return the answer in any order.
# Example 1:
# A node in this binary tree can be flipped by swapping the left child and the rig
# ht child of that node.
# Note:

# NOTE
# 1 <= N <= 100

# EXAMPLE
# Input: root = [1,2,3], voyage = [1,3,2]
# Output: [1]
# Input: root = [1,2], voyage = [2,1]
# Output: [-1]
# Input: root = [1,2,3], voyage = [1,2,3]
# Output: []

# 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]
```

complement-of-base-10-integer.py

```python
# DESC
# Example 1:
# Note:
# Every non-negative integer N has a binary representation.  For example, 5 can be
#  represented as "101" in binary, 11 as "1011" in binary, and so on.  Note that e
# xcept for N = 0, there are no leading zeroes in any binary representation.
# Example 3:
# For a given number N in base-10, return the complement of it's binary representa
# tion as a base-10 integer.
# The complement of a binary representation is the number in binary you get when c
# hanging every 1 to a 0 and 0 to a 1.  For example, the complement of "101" in bi
# nary is "010" in binary.
# Example 2:

# NOTE
# 0 <= N < 10^9
# This question is the same as 476: https://leetcode.com/problems/number-complement/

# EXAMPLE
# Input: 7
# Output: 0
# Explanation: 7 is "111" in binary, with complement "000" in b
# inary, which is 0 in base-10.
# Input: 5
# Output: 2
# Explanation: 5 is "101" in binary, with complement "010" in b
# inary, which is 2 in base-10.
# Input: 10
# Output: 5
# Explanation: 10 is "1010" in binary, with complement "0101"
# in binary, which is 5 in base-10.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def bitwiseComplement(self, N):
        """
        :type N: int
        :rtype: int
        """
        mask = 1
        while N > mask:
            mask = mask*2+1
        return mask-N
```

duplicate-zeros.py

```python
# DESC
# Example 1:
# Given a fixed length array arr of integers, duplicate each occurrence of zero, s
# hifting the remaining elements to the right.
# Note that elements beyond the length of the original array are not written.
# Do the above modifications to the input array in place, do not return anything f
# rom your function.
# Example 2:
# Note:

# NOTE
# 0 <= arr[i] <= 9
# 1 <= arr.length <= 10000

# EXAMPLE
# Input: [1,0,2,3,0,4,5,0]
# Output: null
# Explanation: After calling your function,
# the input array is modified to: [1,0,0,2,3,0,0,4]
# Input: [1,2,3]
# Output: null
# Explanation: After calling your function, the input
# array is modified to: [1,2,3]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def duplicateZeros(self, arr):
        """
        :type arr: List[int]
        :rtype: None Do not return anything, modify arr in-place instead.
        """
        shift, i = 0, 0
        while i+shift < len(arr):
            shift += int(arr[i] == 0)
            i += 1
        i -= 1
        while shift:
            if i+shift < len(arr):
                arr[i+shift] = arr[i]
            if arr[i] == 0:
                shift -= 1
                arr[i+shift] = arr[i]
            i -= 1
```

kth-largest-element-in-a-stream.py

```
# DESC
# Design a class to find the kth largest element in a stream. Note that it is the
# kth largest element in the sorted order, not the kth distinct element.
# Your KthLargest class will have a constructor which accepts an integer k and an
# integer array nums, which contains initial elements from the stream. For each ca
# ll to the method KthLargest.add, return the element representing the kth largest
#  element in the stream.
# Note:
#
# You may assume that nums' length  k-1 and k  1.
# KthLargest
# Example:

# NOTE
#

# EXAMPLE
# int k = 3;
# int[] arr = [4,5,8,2];
# KthLargest kthLargest = new KthLargest(3, arr)
# ;
# kthLargest.add(3);   // returns 4
# kthLargest.add(5);   // returns 5
# kthLargest
# .add(10);  // returns 5
# kthLargest.add(9);   // returns 8
# kthLargest.add(4);   /
# / returns 8

# Time:  O(nlogk)
# Space: O(k)

import heapq


class KthLargest(object):

    def __init__(self, k, nums):
        """
        :type k: int
        :type nums: List[int]
        """
        self.__k = k
        self.__min_heap = []
        for n in nums:
            self.add(n)


    def add(self, val):
        """
        :type val: int
        :rtype: int
        """
        heapq.heappush(self.__min_heap, val)
        if len(self.__min_heap) > self.__k:
            heapq.heappop(self.__min_heap)
        return self.__min_heap[0]
```

design-tic-tac-toe.py

```
# design-tic-tac-toe is not found.
# Time:  O(1), per move.
# Space: O(n^2)


class TicTacToe(object):

    def __init__(self, n):
        """
        Initialize your data structure here.
        :type n: int
        """
        self.__size = n
        self.__rows = [[0, 0] for _ in xrange(n)]
        self.__cols = [[0, 0] for _ in xrange(n)]
        self.__diagonal = [0, 0]
        self.__anti_diagonal = [0, 0]

    def move(self, row, col, player):
        """
        Player {player} makes a move at ({row}, {col}).
        @param row The row of the board.
        @param col The column of the board.
        @param player The player, can be either 1 or 2.
        @return The current winning condition, can be either:
                0: No one wins.
                1: Player 1 wins.
                2: Player 2 wins.
        :type row: int
        :type col: int
        :type player: int
        :rtype: int
        """
        i = player - 1
        self.__rows[row][i] += 1
        self.__cols[col][i] += 1
        if row == col:
            self.__diagonal[i] += 1
        if col == len(self.__rows) - row - 1:
            self.__anti_diagonal[i] += 1
        if any(self.__rows[row][i] == self.__size,
               self.__cols[col][i] == self.__size,
               self.__diagonal[i] == self.__size,
               self.__anti_diagonal[i] == self.__size):
            return player

        return 0
```

bulb-switcher-ii.py

```python
# DESC
# Suppose n lights are labeled as number [1, 2, 3 ..., n], function of these 4 but
# tons are given below:
# Example 1:
# Example 3:
# 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 t
# o return how many different kinds of status of the n lights could be.
# Note: n and m both fit in range [0, 1000].
# Example 2:

# NOTE
# Flip lights with odd numbers.
# Flip all the lights.
# Flip lights with (3k + 1) numbers, k = 0, 1, 2, ...
# Flip lights with even numbers.

# EXAMPLE
# Input: n = 1, m = 1.
# Output: 2
# Explanation: Status can be: [on], [off]
# Input: n = 3, m = 1.
# Output: 4
# Explanation: Status can be: [off, on, off], [on,
# off, on], [off, off, off], [off, on, on].
# Input: n = 2, m = 1.
# Output: 3
# Explanation: Status can be: [on, off], [off, on],
#  [off, off]

# 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
```

employee-importance.py

```
# DESC
# You are given a data structure of employee information, which includes the emplo
# yee's unique id, their importance value and their direct subordinates' id.
# Example 1:
# Note:
# Now given the employee information of a company, and an employee id, you need to
#  return the total importance value of this employee and all their subordinates.
# For example, employee 1 is the leader of employee 2, and employee 2 is the leade
# r of employee 3. They have importance value 15, 10 and 5, respectively. Then emp
# loyee 1 has a data structure like [1, 15, [2]], and employee 2 has [2, 10, [3]],
#  and employee 3 has [3, 5, []]. Note that although employee 3 is also a subordin
# ate of employee 1, the relationship is not direct.

# NOTE
# The maximum number of employees won't exceed 2000.
# One employee has at most one direct leader and may have several subordinates.

# EXAMPLE
# Input: [[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1
# Output: 11
# Explanation:
# Emplo
# yee 1 has importance value 5, and he has two direct subordinates: employee 2 and
#  employee 3. They both have importance value 3. So the total importance value of
#  employee 1 is 5 + 3 + 3 = 11.

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

import collections


"""
# Employee info
class Employee(object):
    def __init__(self, id, importance, subordinates):
        # It's the unique id of each node.
        # unique id of this employee
        self.id = id
        # the importance value of this employee
        self.importance = importance
        # the id of direct subordinates
        self.subordinates = subordinates
"""
class Solution(object):
    def getImportance(self, employees, id):
        """
        :type employees: Employee
        :type id: int
        :rtype: int
        """
        if employees[id-1] is None:
            return 0
        result = employees[id-1].importance
        for id in employees[id-1].subordinates:
            result += self.getImportance(employees, id)
        return result
```

```python
# Time:  O(n)
# Space: O(w), w is the max number of nodes in the levels of the tree
class Solution2(object):
    def getImportance(self, employees, id):
        """
        :type employees: Employee
        :type id: int
        :rtype: int
        """
        result, q = 0, collections.deque([id])
        while q:
            curr = q.popleft()
            employee = employees[curr-1]
            result += employee.importance
            for id in employee.subordinates:
                q.append(id)
        return result
```

reorder-log-files.py

```python
# reorder-log-files is not found.
# Time:  O(nlogn * l), n is the length of files, l is the average length of strings
# Space: O(l)

class Solution(object):
    def reorderLogFiles(self, logs):
        """
        :type logs: List[str]
        :rtype: List[str]
        """
        def f(log):
            i, content = log.split(" ", 1)
            return (0, content, i) if content[0].isalpha() else (1,)

        logs.sort(key=f)
        return logs
```

digit-count-in-range.py

```python
# digit-count-in-range is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def digitsCount(self, d, low, high):
        """
        :type d: int
        :type low: int
        :type high: int
        :rtype: int
        """
        def digitsCount(n, k):
            pivot, result = 1, 0
            while n >= pivot:
                result += (n//(10*pivot))*pivot + \
                          min(pivot, max(n%(10*pivot) - k*pivot + 1, 0))
                if k == 0:
                    result -= pivot
                pivot *= 10
            return result+1

        return digitsCount(high, d) - digitsCount(low-1, d)
```

design-a-stack-with-increment-operation.py

```python
# design-a-stack-with-increment-operation is not found.
# Time:  cotr:      O(1)
#        push:      O(1)
#        pop:       O(1)
#        increment: O(1)
# Space: O(n)

class CustomStack(object):

    def __init__(self, maxSize):
        """
        :type maxSize: int
        """
        self.__max_size = maxSize
        self.__stk = []

    def push(self, x):
        """
        :type x: int
        :rtype: None
        """
        if len(self.__stk) == self.__max_size:
            return
        self.__stk.append([x, 0])

    def pop(self):
        """
        :rtype: int
        """
        if not self.__stk:
            return -1
        x, inc = self.__stk.pop()
        if self.__stk:
            self.__stk[-1][1] += inc
        return x + inc

    def increment(self, k, val):
        """
        :type k: int
        :type val: int
        :rtype: None
        """
        i = min(len(self.__stk), k)-1
        if i >= 0:
            self.__stk[i][1] += val
```

arithmetic-slices.py

```python
# DESC
# A sequence of numbers is called arithmetic if it consists of at least three elem
# ents and if the difference between any two consecutive elements is the same.
# A zero-indexed array A consisting of N numbers is given. A slice of that array i
# s 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
# .
# For example, these are arithmetic sequences:
# Example:
# The function should return the number of arithmetic slices in the array A.
# The following sequence is not arithmetic.

# NOTE
#

# EXAMPLE
# 1, 3, 5, 7, 9
# 7, 7, 7, 7
# 3, -1, -5, -9
# 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.
# 1, 1, 2, 5, 7

# 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

```
# DESC
# How many possible unique paths are there?
# 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 diag
# ram below).
# Example 2:
# Constraints:
# Above is a 7 x 3 grid. How many possible unique paths are there?
# Example 1:
# A robot is located at the top-left corner of a m x n grid (marked 'Start' in the
#  diagram below).

# NOTE
# It's guaranteed that the answer will be less than or equal to 2 * 10 ^ 9.
# 1 <= m, n <= 100

# EXAMPLE
# Input: m = 7, n = 3
# Output: 28
# 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

# 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]
```

ugly-number.py

```python
# DESC
# Example 3:
# Note:
# Example 2:
# 2, 3, 5
# Write a program to check whether a given number is an ugly number.
# Example 1:
# Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.

# NOTE
# 1 is typically treated as an ugly number.
# Input is within the 32-bit signed integer range: [231,  231  1].

# EXAMPLE
# Input: 14
# Output: false
# Explanation: 14 is not ugly since it includes another p
# rime factor 7.
# Input: 6
# Output: true
# Explanation: 6 = 2 × 3
# Input: 8
# Output: true
# Explanation: 8 = 2 × 2 × 2

# Time:  O(logn) = O(1)
# Space: O(1)

class Solution(object):
    # @param {integer} num
    # @return {boolean}
    def isUgly(self, num):
        if num == 0:
            return False
        for i in [2, 3, 5]:
            while num % i == 0:
                num /= i
        return num == 1
```

search-insert-position.py

```
# DESC
# Example 1:
# Example 4:
# Example 2:
# Example 3:
# You may assume no duplicates in the array.
# Given a sorted array and a target value, return the index if the target is found
# . If not, return the index where it would be if it were inserted in order.

# NOTE
#

# EXAMPLE
# Input: [1,3,5,6], 5
# Output: 2
# Input: [1,3,5,6], 2
# Output: 1
# Input: [1,3,5,6], 0
# Output: 0
# Input: [1,3,5,6], 7
# Output: 4

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def searchInsert(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:
                right = mid - 1
            else:
                left = mid + 1

        return left
```

bag-of-tokens.py

```
# DESC
# Note:
# token[i]
# Example 1:
# You have an initial power P, an initial score of 0 points, and a bag of tokens.
# Each token can be used at most once, has a value token[i], and has potentially t
# wo ways to use it.
# Example 2:
# Return the largest number of points we can have after playing any number of tokens.
# Example 3:

# NOTE
# If we have at least token[i] power, we may play the token face up, losing token[
# i] power, and gaining 1 point.
# 0 <= P < 10000
# If we have at least 1 point, we may play the token face down, gaining token[i] p
# ower, and losing 1 point.
# 0 <= tokens[i] < 10000
# tokens.length <= 1000

# EXAMPLE
# Input: tokens = [100], P = 50
# Output: 0
# Input: tokens = [100,200], P = 150
# Output: 1
# Input: tokens = [100,200,300,400], P = 200
# Output: 2

# 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
```

delete-columns-to-make-sorted-iii.py

```
# DESC
# Return the minimum possible value of D.length.
# We are given an array A of N lowercase letter strings, all of the same length.
# For clarity, A[0] is in lexicographic order (ie. A[0][0] <= A[0][1] <= ... <= A[
# 0][A[0].length - 1]), A[1] is in lexicographic order (ie. A[1][0] <= A[1][1] <=
# ... <= A[1][A[1].length - 1]), and so on.
# Suppose we chose a set of deletion indices D such that after deletions, the fina
# l array has every element (row) in lexicographic order.
# Note:
# For example, if we have an array A = ["babca","bbazb"] and deletion indices {0,
# 1, 4}, then the final array after deletions is ["bc","az"].
# Example 1:
# Example 2:
# Now, we may choose any set of deletion indices, and for each string, we delete a
# ll the characters in those indices.
# Example 3:

# NOTE
# 1 <= A.length <= 100
# 1 <= A[i].length <= 100

# EXAMPLE
# Input: ["babca","bbazb"]
# Output: 3
# Explanation: After deleting columns 0, 1, and
#  4, the final array is A = ["bc", "az"].
# Both these rows are individually in lex
# icographic order (ie. A[0][0] <= A[0][1] and A[1][0] <= A[1][1]).
# Note that A[0]
#  > A[1] - the array A isn't necessarily in lexicographic order.
# Input: ["ghi","def","abc"]
# Output: 0
# Explanation: All rows are already lexicogra
# phically sorted.
# Input: ["edcba"]
# Output: 4
# Explanation: If we delete less than 4 columns, the on
# ly row won't be lexicographically sorted.

# Time:  O(n * l^2)
# Space: O(l)

class Solution(object):
    def minDeletionSize(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        dp = [1] * len(A[0])
        for j in xrange(1, len(A[0])):
            for i in xrange(j):
                if all(A[k][i] <= A[k][j] for k in xrange(len(A))):
                    dp[j] = max(dp[j], dp[i]+1)
        return len(A[0]) - max(dp)
```

decode-ways.py

```
# DESC
# Example 1:
# Given a non-empty string containing only digits, determine the total number of w
# ays to decode it.
# Example 2:
# A message containing letters from A-Z is being encoded to numbers using the foll
# owing mapping:

# NOTE
#

# EXAMPLE
# Input: "226"
# Output: 3
# Explanation: It could be decoded as "BZ" (2 26), "VF" (22
#  6), or "BBF" (2 2 6).
# 'A' -> 1
# 'B' -> 2
# ...
# 'Z' -> 26
# Input: "12"
# Output: 2
# Explanation: It could be decoded as "AB" (1 2) or "L" (12).

# 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
```

maximum-number-of-non-overlapping-subarrays-with-sum-equals-target.py

```python
# maximum-number-of-non-overlapping-subarrays-with-sum-equals-target is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def maxNonOverlapping(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        lookup = {0:-1}
        result, accu, right = 0, 0, -1
        for i, num in enumerate(nums):
            accu += num
            if accu-target in lookup and lookup[accu-target] >= right:
                right = i
                result += 1  # greedy
            lookup[accu] = i
        return result
```

armstrong-number.py

```python
# armstrong-number is not found.
# Time:  O(klogk)
# Space: O(k)

class Solution(object):
    def isArmstrong(self, N):
        """
        :type N: int
        :rtype: bool
        """
        n_str = str(N)
        return sum(int(i)**len(n_str) for i in n_str) == N
```

tiling-a-rectangle-with-the-fewest-squares.py

```
# tiling-a-rectangle-with-the-fewest-squares is not found.
# Time:  O(n^2 * m^2 * m^(n * m)), given m < n
# Space: O(n * m)

class Solution(object):
    def tilingRectangle(self, n, m):
        """
        :type n: int
        :type m: int
        :rtype: int
        """
        def find_next(board):
            for i in xrange(len(board)):
                for j in xrange(len(board[0])):
                    if not board[i][j]:
                        return i, j
            return -1, -1

        def find_max_length(board, i, j):
            max_length = 1
            while i+max_length-1 < len(board) and \
                  j+max_length-1 < len(board[0]):
                for r in xrange(i, i+max_length-1):
                    if board[r][j+max_length-1]:
                        return max_length-1
                for c in xrange(j, j+max_length):
                    if board[i+max_length-1][c]:
                        return max_length-1
                max_length += 1
            return max_length-1

        def fill(board, i, j, length, val):
            for r in xrange(i, i+length):
                for c in xrange(j, j+length):
                    board[r][c] = val

        def backtracking(board, count, result):
            if count >= result[0]:  # pruning
                return
            i, j = find_next(board)
            if (i, j) == (-1, -1):  # finished
                result[0] = min(result[0], count)
                return
            max_length = find_max_length(board, i, j)
            for k in reversed(xrange(1, max_length+1)):
                fill(board, i, j, k, 1)
                backtracking(board, count+1, result)
                fill(board, i, j, k, 0)

        if m > n:
            return self.tilingRectangle(m, n)
        board = [[0]*m for _ in xrange(n)]
        result = [float("inf")]
        backtracking(board, 0, result)
        return result[0]
```

statistics-from-a-large-sample.py

```python
# DESC
# Example 1:
# 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.
# (Recall that the median of a sample is:
# 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.
# Constraints:
# Example 2:

# NOTE
# The mode of the sample that count represents is unique.
# The middle element, if the elements of the sample were sorted and the number of
# elements is odd;
# 1 <= sum(count) <= 10^9
# The average of the middle two elements, if the elements of the sample were sorte
# d and the number of elements is even.)
# Answers within 10^-5 of the true value will be accepted as correct.
# count.length == 256

# EXAMPLE
# 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: [1.00000,4.00000,2.1818
# 2,2.00000,1.00000]
# 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: [1.00000,3.00000,2.3750
# 0,2.50000,3.00000]

# 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]
```

count-good-nodes-in-binary-tree.py

```python
# count-good-nodes-in-binary-tree is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution(object):
    def goodNodes(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result = 0
        stk = [(root, root.val)]
        while stk:
            node, curr_max = stk.pop()
            if not node:
                continue
            curr_max = max(curr_max, node.val)
            result += int(curr_max <= node.val)
            stk.append((node.right, curr_max))
            stk.append((node.left, curr_max))
        return result


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def goodNodes(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, curr_max):
            if not node:
                return 0
            curr_max = max(curr_max, node.val)
            return (int(curr_max <= node.val) +
                    dfs(node.left, curr_max) + dfs(node.right, curr_max))

        return dfs(root, root.val)
```

minimum-swaps-to-group-all-1s-together.py

```python
# minimum-swaps-to-group-all-1s-together is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minSwaps(self, data):
        """
        :type data: List[int]
        :rtype: int
        """
        total_count = sum(data)
        result, count, left = 0, 0, 0
        for i in xrange(len(data)):
            count += data[i]
            if i-left+1 > total_count:
                count -= data[left]
                left += 1
            result = max(result, count)
        return total_count-result
```

remove-vowels-from-a-string.py

```python
# remove-vowels-from-a-string is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def removeVowels(self, S):
        """
        :type S: str
        :rtype: str
        """
        lookup = set("aeiou")
        return "".join(c for c in S if c not in lookup)
```

stream-of-characters.py

```
# DESC
# Implement the StreamChecker class as follows:
# Note:
# Example:

# NOTE
# The number of queries is at most 40000.
# StreamChecker(words): Constructor, init the data structure with the given words.
# 1 <= words.length <= 2000
# Queries will only consist of lowercase English letters.
# Words will only consist of lowercase English letters.
# query(letter): returns true if and only if for some k >= 1, the last k character
# s queried (in order from oldest to newest, including this letter just queried) s
# pell one of the words in the given list.
# 1 <= words[i].length <= 2000

# EXAMPLE
# StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]); // init the di
# ctionary.
# streamChecker.query('a');          // return false
# streamChecker.query
# ('b');          // return false
# streamChecker.query('c');          // return fal
# se
# streamChecker.query('d');          // return true, because 'cd' is in the wor
# dlist
# streamChecker.query('e');          // return false
# streamChecker.query('f'
# );          // return true, because 'f' is in the wordlist
# streamChecker.query('
# g');          // return false
# streamChecker.query('h');          // return false
#
# streamChecker.query('i');          // return false
# streamChecker.query('j');
#        // return false
# streamChecker.query('k');          // return false
# stream
# Checker.query('l');          // return true, because 'kl' is in the wordlist

# Time:  ctor:  O(n)    , n is the total size of patterns
#        query: O(m + z), m is the total size of query string
#                       , z is the number of all matched strings
#                       , query time could be further improved to O(m) if we don't return all matched patterns
# Space: O(t), t is the total size of ac automata trie
#            , space could be further improved by DAT (double-array trie)

# Aho-Corasick automata
# reference:
# 1. http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/02/Small02.pdf
# 2. http://algo.pw/algo/64/python

import collections


class AhoNode(object):
    def __init__(self):
        self.children = collections.defaultdict(AhoNode)
```

```
        self.indices = []
        self.suffix = None
        self.output = None


class AhoTrie(object):

    def step(self, letter):
        while self.__node and letter not in self.__node.children:
            self.__node = self.__node.suffix
        self.__node = self.__node.children[letter] if self.__node else self.__root
        return self.__get_ac_node_outputs(self.__node)

    def __init__(self, patterns):
        self.__root = self.__create_ac_trie(patterns)
        self.__node = self.__create_ac_suffix_and_output_links(self.__root)

    def __create_ac_trie(self, patterns):  # Time:  O(n), Space: O(t)
        root = AhoNode()
        for i, pattern in enumerate(patterns):
            node = root
            for c in pattern:
                node = node.children[c]
            node.indices.append(i)
        return root

    def __create_ac_suffix_and_output_links(self, root):  # Time:  O(n), Space: O(t)
        queue = collections.deque()
        for node in root.children.itervalues():
            queue.append(node)
            node.suffix = root

        while queue:
            node = queue.popleft()
            for c, child in node.children.iteritems():
                queue.append(child)
                suffix = node.suffix
                while suffix and c not in suffix.children:
                    suffix = suffix.suffix
                child.suffix = suffix.children[c] if suffix else root
                child.output = child.suffix if child.suffix.indices else child.suffix.output

        return root

    def __get_ac_node_outputs(self, node):  # Time:  O(z), in this question, it could be improved to O(1)
                                            # if we only return a matched pattern without all matched ones
        result = []
        for i in node.indices:
            result.append(i)
            # return result
        output = node.output
        while output:
            for i in output.indices:
                result.append(i)
                # return result
            output = output.output
        return result


class StreamChecker(object):
```

```
    def __init__(self, words):
        """
        :type words: List[str]
        """
        self.__trie = AhoTrie(words)

    def query(self, letter):  # O(m) times
        """
        :type letter: str
        :rtype: bool
        """
        return len(self.__trie.step(letter)) > 0


# Your StreamChecker object will be instantiated and called as such:
# obj = StreamChecker(words)
# param_1 = obj.query(letter)
```

array-nesting.py

```
# DESC
# 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.
# Example 1:
# Note:
# Suppose the first element in S starts with the selection of element A[i] of inde
# x = i, the next element in S should be A[A[i]], and then A[A[A[i]]]... By that ana
# logy, we stop adding right before a duplicate element occurs in S.

# 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].

# EXAMPLE
# 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}

# 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

```
# DESC
# Example 2:
# Given two integers dividend and divisor, divide two integers without using multi
# plication, division and mod operator.
# dividend
# Return the quotient after dividing dividend by divisor.
# The integer division should truncate toward zero, which means losing its fractio
# nal part. For example, truncate(8.345) = 8 and truncate(-2.7335) = -2.
# Note:
# Example 1:

# NOTE
# 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 prob
# lem, assume that your function returns 231  1 when the division result overflow
# s.
# Both dividend and divisor will be 32-bit signed integers.

# EXAMPLE
# Input: dividend = 7, divisor = -3
# Output: -2
# Explanation: 7/-3 = truncate(-2.333
# 33..) = -2.
# Input: dividend = 10, divisor = 3
# Output: 3
# Explanation: 10/3 = truncate(3.33333
# ..) = 3.

# 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
        :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)
```

factorial-trailing-zeroes.py

```
# DESC
# Given an integer n, return the number of trailing zeroes in n!.
# Note: Your solution should be in logarithmic time complexity.
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: 3
# Output: 0
# Explanation: 3! = 6, no trailing zero.
# Input: 5
# Output: 1
# Explanation: 5! = 120, one trailing zero.

# Time:  O(logn) = O(1)
# Space: O(1)

class Solution(object):
    # @return an integer
    def trailingZeroes(self, n):
        result = 0
        while n > 0:
            result += n / 5
            n /= 5
        return result
```

rotate-string.py

```
# DESC
# A shift on A consists of taking string A and moving the leftmost character to th
# e rightmost position. For example, if A = 'abcde', then it will be 'bcdea' after
#  one shift on A. Return True if and only if A can become B after some number of
# shifts on A.
# We are given two strings, A and B.
# Note:

# NOTE
# A and B will have length at most 100.

# EXAMPLE
# Example 1:
# Input: A = 'abcde', B = 'cdeab'
# Output: true
#
# Example 2:
# Input: A = '
# abcde', B = 'abced'
# Output: false

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def rotateString(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: bool
        """
        def check(index):
            return all(A[(i+index) % len(A)] == c
                       for i, c in enumerate(B))

        if len(A) != len(B):
            return False

        M, p = 10**9+7, 113
        p_inv = pow(p, M-2, M)

        b_hash, power = 0, 1
        for c in B:
            b_hash += power * ord(c)
            b_hash %= M
            power = (power*p) % M

        a_hash, power = 0, 1
        for i in xrange(len(B)):
            a_hash += power * ord(A[i%len(A)])
            a_hash %= M
            power = (power*p) % M

        if a_hash == b_hash and check(0): return True

        power = (power*p_inv) % M
        for i in xrange(len(B), 2*len(A)):
            a_hash = (a_hash-ord(A[(i-len(B))%len(A)])) * p_inv
```

```
            a_hash += power * ord(A[i%len(A)])
            a_hash %= M
            if a_hash == b_hash and check(i-len(B)+1):
                return True

        return False


# Time:  O(n)
# Space: O(n)
# KMP algorithm
class Solution2(object):
    def rotateString(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: bool
        """
        def strStr(haystack, needle):
            def KMP(text, pattern):
                prefix = getPrefix(pattern)
                j = -1
                for i in xrange(len(text)):
                    while j > -1 and pattern[j + 1] != text[i]:
                        j = prefix[j]
                    if pattern[j + 1] == text[i]:
                        j += 1
                    if j == len(pattern) - 1:
                        return i - j
                return -1

            def getPrefix(pattern):
                prefix = [-1] * len(pattern)
                j = -1
                for i in xrange(1, len(pattern)):
                    while j > -1 and pattern[j + 1] != pattern[i]:
                        j = prefix[j]
                    if pattern[j + 1] == pattern[i]:
                        j += 1
                    prefix[i] = j
                return prefix

            if not needle:
                return 0
            return KMP(haystack, needle)

        if len(A) != len(B):
            return False
        return strStr(A*2, B) != -1


# Time:  O(n^2)
# Space: O(n)
class Solution3(object):
    def rotateString(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: bool
        """
```

```
        return len(A) == len(B) and B in A*2
```

coin-change.py

```python
# DESC
# Example 1:
# You are given coins of different denominations and a total amount of money amoun
# t. 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 2:
# Note:
#
# You may assume that you have an infinite number of each kind of coin.

# NOTE
#

# EXAMPLE
# Input: coins = [2], amount = 3
# Output: -1
# Input: coins = [1, 2, 5], amount = 11
# Output: 3
# Explanation: 11 = 5 + 5 + 1

# Time:  O(n * k), n is the number of coins, k is the amount of money
# 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

```
# DESC
# Example 5:
# Example 4:
# Note:
# (Note that backslash characters are escaped, so a \ is represented as "\\".)
# In a N x N grid composed of 1 x 1 squares, each 1 x 1 square consists of a /, \,
#  or blank space.  These characters divide the square into contiguous regions.
# Example 1:
# Return the number of regions.
# Example 2:
# Example 3:

# NOTE
# 1 <= grid.length == grid[0].length <= 30
# grid[i][j] is either '/', '\', or ' '.

# EXAMPLE
# Input:
# [
#   "/\\",
#   "\\/"
# ]
# Output: 5
# Explanation: (Recall that because \ charac
# ters are escaped, "/\\" refers to /\, and "\\/" refers to \/.)
# The 2x2 grid is a
# s follows:
# Input:
# [
#   " /",
#   "  "
# ]
# Output: 1
# Explanation: The 2x2 grid is as follows:
# Input:
# [
#   "\\/",
#   "/\\"
# ]
# Output: 4
# Explanation: (Recall that because \ charac
# ters are escaped, "\\/" refers to \/, and "/\\" refers to /\.)
# The 2x2 grid is a
# s follows:
# Input:
# [
#   " /",
#   "/ "
# ]
# Output: 2
# Explanation: The 2x2 grid is as follows:
# Input:
# [
#   "//",
#   "/ "
# ]
# Output: 3
# Explanation: The 2x2 grid is as follows:
```

```python
# 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:
           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
```

moving-stones-until-consecutive.py

```
# DESC
# Three stones are on a number line at positions a, b, and c.
# x, y, z
# The game ends when you cannot make any more moves, ie. the stones are in consecu
# tive positions.
# When the game ends, what is the minimum and maximum number of moves that you cou
# ld have made?  Return the answer as an length 2 array: answer = [minimum_moves,
# maximum_moves]
# Example 2:
# Example 1:
# Each turn, you pick up a stone at an endpoint (ie., either the lowest or highest
#  position stone), and move it to an unoccupied position between those endpoints.
#   Formally, let's say the stones are currently at positions x, y, z with x < y <
#  z.  You pick up the stone at either position x or position z, and move that sto
# ne to an integer position k, with x < k < z and k != y.
# Example 3:
# Note:

# NOTE
# 1 <= b <= 100
# 1 <= a <= 100
# 1 <= c <= 100
# a != b, b != c, c != a

# EXAMPLE
# Input: a = 1, b = 2, c = 5
# Output: [1,2]
# Explanation: Move the stone from 5 to 3
# , or move the stone from 5 to 4 to 3.
# Input: a = 4, b = 3, c = 2
# Output: [0,0]
# Explanation: We cannot make any moves.
# Input: a = 3, b = 5, c = 1
# Output: [1,2]
# Explanation: Move the stone from 1 to 4
# ; or move the stone from 1 to 2 to 4.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def numMovesStones(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: List[int]
        """
        s = [a, b, c]
        s.sort()
        if s[0]+1 == s[1] and s[1]+1 == s[2]:
            return [0, 0]
        return [1 if s[0]+2 >= s[1] or s[1]+2 >= s[2] else 2, s[2]-s[0]-2]


# Time:  O(1)
# Space: O(1)
class Solution2(object):
```

```
def numMovesStones(self, a, b, c):
    """
    :type a: int
    :type b: int
    :type c: int
    :rtype: List[int]
    """
    stones = [a, b, c]
    stones.sort()
    left, min_moves = 0, float("inf")
    max_moves = (stones[-1]-stones[0]) - (len(stones)-1)
    for right in xrange(len(stones)):
        while stones[right]-stones[left]+1 > len(stones): # find window size <= len(stones)
            left += 1
        min_moves = min(min_moves, len(stones)-(right-left+1))  # move stones not in this window
    return [min_moves, max_moves]
```

maximum-gap.py

```
# maximum-ga is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def maximumGap(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if len(nums) < 2:
            return 0

        # Init bucket.
        max_val, min_val = max(nums), min(nums)
        gap = max(1, (max_val - min_val) / (len(nums) - 1))
        bucket_size = (max_val - min_val) / gap + 1
        bucket = [{'min':float("inf"), 'max':float("-inf")} \
                    for _ in xrange(bucket_size)]

        # Find the bucket where the n should be put.
        for n in nums:
            # min_val / max_val is in the first / last bucket.
            if n in (max_val, min_val):
                continue
            i = (n - min_val) / gap
            bucket[i]['min'] = min(bucket[i]['min'], n)
            bucket[i]['max'] = max(bucket[i]['max'], n)

        # Count each bucket gap between the first and the last bucket.
        max_gap, pre_bucket_max = 0, min_val
        for i in xrange(bucket_size):
            # Skip the bucket it empty.
            if bucket[i]['min'] == float("inf") and \
                bucket[i]['max'] == float("-inf"):
                continue
            max_gap = max(max_gap, bucket[i]['min'] - pre_bucket_max)
            pre_bucket_max = bucket[i]['max']
        # Count the last bucket.
        max_gap = max(max_gap, max_val - pre_bucket_max)

        return max_gap


# Time:  O(nlogn)
# Space: O(n)
class Solution2(object):
    def maximumGap(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if len(nums) < 2:
            return 0

        nums.sort()
        pre = nums[0]
```

```
max_gap = float("-inf")

for i in nums:
    max_gap = max(max_gap, i - pre)
    pre = i
return max_gap
```

letter-combinations-of-a-phone-number.py

```
# DESC
# A mapping of digit to letters (just like on the telephone buttons) is given belo
# w. Note that 1 does not map to any letters.
# Example:
# Note:
# Although the above answer is in lexicographical order, your answer could be in a
# ny order you want.
# Given a string containing digits from 2-9 inclusive, return all possible letter
# combinations that the number could represent.

# NOTE
#

# EXAMPLE
# Input: "23"
# Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

# 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])]:
```

```
            self.letterCombinationsRecu(result, digits, lookup, cur + choice, n + 1)
```

longest-repeating-character-replacement.py

```
# DESC
# Note:
#
# Both the string's length and k will not exceed 104.
# Find the length of the longest sub-string containing all repeating letters you c
# an get after performing the above operations.
# Given a string s that consists of only uppercase English letters, you can perfor
# m at most k operations on that string.
# In one operation, you can choose any character of the string and change it to an
# y other uppercase English character.
# Example 2:
# Example 1:

# NOTE
#

# EXAMPLE
# 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 repeati
# ng letters, which is 4.
# Input:
# s = "ABAB", k = 2
#
# Output:
# 4
#
# Explanation:
# Replace the two 'A's with two
# 'B's or vice versa.

# 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

```
# search-in-rotated-sorted-arra is not found.
# 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
```

number-of-steps-to-reduce-a-number-to-zero.py

```python
# DESC
# Example 1:
# Example 2:
# Example 3:
# Given a non-negative integer num, return the number of steps to reduce it to zer
# o. If the current number is even, you have to divide it by 2, otherwise, you hav
# e to subtract 1 from it.
# Constraints:

# NOTE
# 0 <= num <= 10^6

# EXAMPLE
# Input: num = 123
# Output: 12
# Input: num = 8
# Output: 4
# Explanation:
# Step 1) 8 is even; divide by 2 and obtain
#  4.
# Step 2) 4 is even; divide by 2 and obtain 2.
# Step 3) 2 is even; divide by
# 2 and obtain 1.
# Step 4) 1 is odd; subtract 1 and obtain 0.
# Input: num = 14
# Output: 6
# Explanation:
# Step 1) 14 is even; divide by 2 and obta
# in 7.
# Step 2) 7 is odd; subtract 1 and obtain 6.
# Step 3) 6 is even; divide by 2
#  and obtain 3.
# Step 4) 3 is odd; subtract 1 and obtain 2.
# Step 5) 2 is even; d
# ivide by 2 and obtain 1.
# Step 6) 1 is odd; subtract 1 and obtain 0.

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def numberOfSteps (self, num):
        """
        :type num: int
        :rtype: int
        """
        result = 0
        while num:
            result += 2 if num%2 else 1
            num //= 2
        return result-1
```

increasing-triplet-subsequence.py

```python
# DESC
# Example 1:
# Given an unsorted array return whether an increasing subsequence of length 3 exi
# sts or not in the array.
# Note: Your algorithm should run in O(n) time complexity and O(1) space complexity.
# Example 2:
# Formally the function should:

# NOTE
#

# EXAMPLE
# Input: [5,4,3,2,1]
# Output: false
# Input: [1,2,3,4,5]
# Output: true

# 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)
```

filter-restaurants-by-vegan-friendly-price-and-distance.py

```python
# filter-restaurants-by-vegan-friendly-price-and-distance is not found.
# Time:  O(rlogr), r is the number of result
# Space: O(r)

class Solution(object):
    def filterRestaurants(self, restaurants, veganFriendly, maxPrice, maxDistance):
        """
        :type restaurants: List[List[int]]
        :type veganFriendly: int
        :type maxPrice: int
        :type maxDistance: int
        :rtype: List[int]
        """
        result, lookup = [], {}
        for j, (i, _, v, p, d) in enumerate(restaurants):
            if v >= veganFriendly and p <= maxPrice and d <= maxDistance:
                lookup[i] = j
                result.append(i)
        result.sort(key=lambda i: (-restaurants[lookup[i]][1], -restaurants[lookup[i]][0]))
        return result
```

remove-interval.py

```python
# remove-interval is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def removeInterval(self, intervals, toBeRemoved):
        """
        :type intervals: List[List[int]]
        :type toBeRemoved: List[int]
        :rtype: List[List[int]]
        """
        A, B = toBeRemoved
        return [[x, y] for a, b in intervals
                for x, y in ((a, min(A, b)), (max(a, B), b))
                if x < y]
```

longest-palindrome.py

```python
# DESC
# Given a string which consists of lowercase or uppercase letters, find the length
#  of the longest palindromes that can be built with those letters.
# Example:
# This is case sensitive, for example "Aa" is not considered a palindrome here.
# Note:
#
# Assume the length of given string will not exceed 1,010.

# NOTE
#

# EXAMPLE
# Input:
# "abcccdd"
#
# Output:
# 7
#
# Explanation:
# One longest palindrome that can be bu
# ilt is "dccaccd", whose length is 7.

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: int
        """
        odds = 0
        for k, v in collections.Counter(s).iteritems():
            odds += v & 1
        return len(s) - odds + int(odds > 0)

    def longestPalindrome2(self, s):
        """
        :type s: str
        :rtype: int
        """
        odd = sum(map(lambda x: x & 1, collections.Counter(s).values()))
        return len(s) - odd + int(odd > 0)
```

univalued-binary-tree.py

```
# DESC
# Return true if and only if the given tree is univalued.
# Example 2:
# Example 1:
# Note:
# A binary tree is univalued if every node in the tree has the same value.

# NOTE
# Each node's value will be an integer in the range [0, 99].
# The number of nodes in the given tree will be in the range [1, 100].

# EXAMPLE
# Input: [1,1,1,1,1,null,1]
# Output: true
# Input: [2,2,2,5,2]
# Output: false

# 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 isUnivalTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        s = [root]
        while s:
            node = s.pop()
            if not node:
                continue
            if node.val != root.val:
                return False
            s.append(node.left)
            s.append(node.right)
        return True


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def isUnivalTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        return (not root.left or (root.left.val == root.val and self.isUnivalTree(root.left))) and \
               (not root.right or (root.right.val == root.val and self.isUnivalTree(root.right)))
```

defanging-an-ip-address.py

```python
# DESC
# Example 1:
# Constraints:
# Example 2:
# A defanged IP address replaces every period "." with "[.]".
# Given a valid (IPv4) IP address, return a defanged version of that IP address.

# NOTE
# The given address is a valid IPv4 address.

# EXAMPLE
# Input: address = "1.1.1.1"
# Output: "1[.]1[.]1[.]1"
# Input: address = "255.100.50.0"
# Output: "255[.]100[.]50[.]0"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def defangIPaddr(self, address):
        """
        :type address: str
        :rtype: str
        """
        result = []
        for c in address:
            if c == '.':
                result.append("[.]")
            else:
                result.append(c)
        return "".join(result)
```

distribute-candies.py

```
# DESC
# Example 2:
# Note:
# Example 1:

# NOTE
# The number in given array is in range [-100,000, 100,000].
# The length of the given array is in range [2, 10,000], and will be even.

# EXAMPLE
# Input: candies = [1,1,2,2,3,3]
# Output: 3
# Explanation:
# There are three different
# kinds of candies (1, 2 and 3), and two candies for each kind.
# Optimal distributi
# on: The sister has candies [1,2,3] and the brother has candies [1,2,3], too.
# Th
# e sister has three different kinds of candies.
# Input: candies = [1,1,2,3]
# Output: 2
# Explanation: For example, the sister has ca
# ndies [2,3] and the brother has candies [1,1].
# The sister has two different kin
# ds of candies, the brother has only one kind of candies.

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

class Solution(object):

    def distributeCandies(self, candies):
        """
        :type candies: List[int]
        :rtype: int
        """
        lookup = set(candies)
        return min(len(lookup), len(candies)/2)
```

permutations-ii.py

```
# DESC
# Example:
# Given a collection of numbers that might contain duplicates, return all possible
#  unique permutations.

# NOTE
#

# 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
```

maximum-performance-of-a-team.py

```python
# maximum-performance-of-a-team is not found.
# Time:  O(nlogn)
# Space: O(n)

import itertools
import heapq


class Solution(object):
    def maxPerformance(self, n, speed, efficiency, k):
        """
        :type n: int
        :type speed: List[int]
        :type efficiency: List[int]
        :type k: int
        :rtype: int
        """
        MOD = 10**9 + 7
        result, s_sum = 0, 0
        min_heap = []
        for e, s in sorted(itertools.izip(efficiency, speed), reverse=True):
            s_sum += s
            heapq.heappush(min_heap, s)
            if len(min_heap) > k:
                s_sum -= heapq.heappop(min_heap)
            result = max(result,s_sum*e)
        return result % MOD
```

detect-capital.py

```
# DESC
# Example 2:
# We define the usage of capitals in a word to be right when one of the following
# cases holds:
# Example 1:
# Given a word, you need to judge whether the usage of capitals in it is right or not.
# Note: The input will be a non-empty word consisting of uppercase and lowercase l
# atin letters.

# NOTE
# All letters in this word are capitals, like "USA".
# Only the first letter in this word is capital, like "Google".
# All letters in this word are not capitals, like "leetcode".

# EXAMPLE
# Input: "USA"
# Output: True
# Input: "FlaG"
# Output: False

# Time:  O(l)
# Space: O(1)

class Solution(object):
    def detectCapitalUse(self, word):
        """
        :type word: str
        :rtype: bool
        """
        return word.isupper() or word.islower() or word.istitle()
```

minimum-size-subarray-sum.py

```
# DESC
# Given an array of n positive integers and a positive integer s, find the minimal
#  length of a contiguous subarray of which the sum  s. If there isn't one, retur
# n 0 instead.
# Example:

# NOTE
#

# EXAMPLE
# Input: s = 7, nums = [2,3,1,2,4,3]
# Output: 2
# Explanation: the subarray [4,3] has
#  the minimal length under the problem constraint.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {integer} s
    # @param {integer[]} nums
    # @return {integer}
    def minSubArrayLen(self, s, nums):
        start = 0
        sum = 0
        min_size = float("inf")
        for i in xrange(len(nums)):
            sum += nums[i]
            while sum >= s:
                min_size = min(min_size, i - start + 1)
                sum -= nums[start]
                start += 1

        return min_size if min_size != float("inf") else 0

# Time:  O(nlogn)
# Space: O(n)
# Binary search solution.
class Solution2(object):
    # @param {integer} s
    # @param {integer[]} nums
    # @return {integer}
    def minSubArrayLen(self, s, nums):
        min_size = float("inf")
        sum_from_start = [n for n in nums]
        for i in xrange(len(sum_from_start) - 1):
            sum_from_start[i + 1] += sum_from_start[i]
        for i in xrange(len(sum_from_start)):
            end = self.binarySearch(lambda x, y: x <= y, sum_from_start, \
                                    i, len(sum_from_start), \
                                    sum_from_start[i] - nums[i] + s)
            if end < len(sum_from_start):
                min_size = min(min_size, end - i + 1)

        return min_size if min_size != float("inf") else 0

    def binarySearch(self, compare, A, start, end, target):
        while start < end:
```

```
    mid = start + (end - start) / 2
    if compare(target, A[mid]):
        end = mid
    else:
        start = mid + 1
return start
```

shortest-word-distance-ii.py

```
# shortest-word-distance-ii is not found.
# Time:  init: O(n), lookup: O(a + b), a, b is occurences of word1, word2
# Space: O(n)

import collections


class WordDistance(object):
    # initialize your data structure here.
    # @param {string[]} words
    def __init__(self, words):
        self.wordIndex = collections.defaultdict(list)
        for i in xrange(len(words)):
            self.wordIndex[words[i]].append(i)

    # @param {string} word1
    # @param {string} word2
    # @return {integer}
    # Adds a word into the data structure.
    def shortest(self, word1, word2):
        indexes1 = self.wordIndex[word1]
        indexes2 = self.wordIndex[word2]

        i, j, dist = 0, 0, float("inf")
        while i < len(indexes1) and j < len(indexes2):
            dist = min(dist, abs(indexes1[i] - indexes2[j]))
            if indexes1[i] < indexes2[j]:
                i += 1
            else:
                j += 1

        return dist
```

three-equal-parts.py

```
# DESC
# If it is possible, return any [i, j] with i+1 < j, such that:
# [i, j]
# Note that the entire part is used when considering what binary value it represen
# ts.  For example, [1,1,0] represents 6 in decimal, not 3.  Also, leading zeros a
# re allowed, so [0,1,1] and [1,1] represent the same value.
# Example 1:
# Note:
# Example 2:
# If it is not possible, return [-1, -1].
# Given an array A of 0s and 1s, divide the array into 3 non-empty parts such that
#  all of these parts represent the same binary value.

# NOTE
# All three parts have equal binary value.
# 3 <= A.length <= 30000
# A[0], A[1], ..., A[i] is the first part;
# A[i+1], A[i+2], ..., A[j-1] is the second part, and
# A[j], A[j+1], ..., A[A.length - 1] is the third part.
# A[i] == 0 or A[i] == 1

# EXAMPLE
# Input: [1,0,1,0,1]
# Output: [0,3]
# Input: [1,1,0,1,1]
# Output: [-1,-1]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def threeEqualParts(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        total = sum(A)
        if total % 3 != 0:
            return [-1, -1]
        if total == 0:
            return [0, len(A)-1]

        count = total//3
        nums = [0]*3
        c = 0
        for i in xrange(len(A)):
            if A[i] == 1:
                if c % count == 0:
                    nums[c//count] = i
                c += 1

        while nums[2] != len(A):
            if not A[nums[0]] == A[nums[1]] == A[nums[2]]:
                return [-1, -1]
            nums[0] += 1
            nums[1] += 1
            nums[2] += 1
        return [nums[0]-1, nums[1]]
```

find-and-replace-pattern.py

```
# DESC
# You have a list of words and a pattern, and you want to know which words in word
# s matches the pattern.
# Note:
# Return a list of the words in words that match the given pattern.
# (Recall that a permutation of letters is a bijection from letters to letters: ev
# ery letter maps to another letter, and no two letters map to the same letter.)
# A word matches the pattern if there exists a permutation of letters p so that af
# ter replacing every letter x in the pattern with p(x), we get the desired word.
# You may return the answer in any order.
# Example 1:

# NOTE
# 1 <= pattern.length = words[i].length <= 20
# 1 <= words.length <= 50

# EXAMPLE
# Input: words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"
# Output: ["
# mee","aqq"]
# Explanation: "mee" matches the pattern because there is a permutatio
# n {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.

# 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)
```

island-perimeter.py

```
# DESC
# The island doesn't have "lakes" (water inside that isn't connected to the water
# around the island). One cell is a square with side length 1. The grid is rectang
# ular, width and height don't exceed 100. Determine the perimeter of the island.
# You are given a map in form of a two-dimensional integer grid where 1 represents
#  land and 0 represents water.
# Grid cells are connected horizontally/vertically (not diagonally). The grid is c
# ompletely surrounded by water, and there is exactly one island (i.e., one or mor
# e connected land cells).
# Example:

# NOTE
#

# EXAMPLE
# Input:
# [[0,1,0,0],
#  [1,1,1,0],
#  [0,1,0,0],
#  [1,1,0,0]]
#
# Output: 16
#
# Explanation:
#  The perimeter is the 16 yellow stripes in the image below:

# Time:  O(m * n)
# Space: O(1)

import operator


class Solution(object):
    def islandPerimeter(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        count, repeat = 0, 0

        for i in xrange(len(grid)):
            for j in xrange(len(grid[i])):
                if grid[i][j] == 1:
                    count += 1
                    if i != 0 and grid[i - 1][j] == 1:
                        repeat += 1
                    if j != 0 and grid[i][j - 1] == 1:
                        repeat += 1

        return 4*count - 2*repeat

# Since there are no lakes, every pair of neighbour cells with different values is part of the perimeter
# (more precisely, the edge between them is). So just count the differing pairs, both horizontally and vertica
# (for the latter I simply transpose the grid).
    def islandPerimeter2(self, grid):
        return sum(sum(map(operator.ne, [0] + row, row + [0])) for row in grid + map(list, zip(*grid)))
```

candy.py

```
# cand is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    # @param ratings, a list of integer
    # @return an integer
    def candy(self, ratings):
        candies = [1 for _ in xrange(len(ratings))]
        for i in xrange(1, len(ratings)):
            if ratings[i] > ratings[i - 1]:
                candies[i] = candies[i - 1] + 1

        for i in reversed(xrange(1, len(ratings))):
            if ratings[i - 1] > ratings[i] and candies[i - 1] <= candies[i]:
                candies[i - 1] = candies[i] + 1

        return sum(candies)
```

time-based-key-value-store.py

```
# DESC
# Note:
# 2. get(string key, int timestamp)
# Example 2:
# Example 1:
# Create a timebased key-value store class TimeMap, that supports two operations.
# 1. set(string key, string value, int timestamp)
# set(string key, string value, int timestamp)

# NOTE
# All key/value strings have length in the range [1, 100]
# 1 <= timestamp <= 10^7
# Stores the key and value, along with the given timestamp.
# All key/value strings are lowercase.
# TimeMap.set and TimeMap.get functions will be called a total of 120000 times (co
# mbined) per test case.
# Returns a value such that set(key, value, timestamp_prev) was called previously,
#  with timestamp_prev <= timestamp.
# The timestamps for all TimeMap.set operations are strictly increasing.
# 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
# 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" si
# nce 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"
# Input: inputs = ["TimeMap","set","set","get","get","get","get","get"], inputs =
# [[],["love","high",10],["love","low",20],["love",5],["love",10],["love",15],["lo
# ve",20],["love",25]]
# Output: [null,null,null,"","high","high","low","low"]

# 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)
```

recover-a-tree-from-preorder-traversal.py

```
# DESC
# If a node has only one child, that child is guaranteed to be the left child.
# Example 2:
# Example 3:
# We run a preorder depth first search on the root of a binary tree.
# At each node in this traversal, we output D dashes (where D is the depth of this
#  node), then we output the value of this node.  (If the depth of a node is D, th
# e depth of its immediate child is D+1.  The depth of the root node is 0.)
# Note:
# Given the output S of this traversal, recover the tree and return its root.
# Example 1:

# NOTE
# The number of nodes in the original tree is between 1 and 1000.
# Each node will have a value between 1 and 10^9.

# EXAMPLE
# Input: "1-2--3---4-5--6---7"
# Output: [1,2,5,3,null,6,null,4,null,7]
# Input: "1-401--349---90--88"
# Output: [1,401,null,349,88,90]
# Input: "1-2--3--4-5--6--7"
# Output: [1,2,5,3,4,6,7]

# 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 recoverFromPreorder(self, S):
        """
        :type S: str
        :rtype: TreeNode
        """
        i = 0
        stack = []
        while i < len(S):
            level = 0
            while i < len(S) and S[i] == '-':
                level += 1
                i += 1
            while len(stack) > level:
                stack.pop()
            val = []
            while i < len(S) and S[i] != '-':
                val.append(S[i])
                i += 1
            node = TreeNode(int("".join(val)))
            if stack:
                if stack[-1].left is None:
```

```
                    stack[-1].left = node
                else:
                    stack[-1].right = node
            stack.append(node)
        return stack[0]


# Time:  O(n)
# Space: O(h)
# recursive solution
class Solution2(object):
    def recoverFromPreorder(self, S):
        """
        :type S: str
        :rtype: TreeNode
        """
        def recoverFromPreorderHelper(S, level, i):
            j = i[0]
            while j < len(S) and S[j] == '-':
                j += 1
            if level != j - i[0]:
                return None
            i[0] = j
            while j < len(S) and S[j] != '-':
                j += 1
            node = TreeNode(int(S[i[0]:j]))
            i[0] = j
            node.left = recoverFromPreorderHelper(S, level+1, i)
            node.right = recoverFromPreorderHelper(S, level+1, i)
            return node

        return recoverFromPreorderHelper(S, 0, [0])
```

number-of-music-playlists.py

```
# DESC
# Example 3:
# Note:
# 10^9 + 7
# Example 1:
# Return the number of possible playlists.  As the answer can be very large, retur
# n it modulo 10^9 + 7.
# Example 2:
# Your music player contains N different songs and she wants to listen to L (not n
# ecessarily different) songs during your trip.  You create a playlist so that:

# NOTE
# A song can only be played again only if K other songs have been played
# 0 <= K < N <= L <= 100
# Every song is played at least once

# EXAMPLE
# Input: N = 3, L = 3, K = 1
# Output: 6
# Explanation: There are 6 possible playlists
# . [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].
# Input: N = 2, L = 3, K = 1
# Output: 2
# Explanation: There are 2 possible playlists
# . [1, 2, 1], [2, 1, 2]
# Input: N = 2, L = 3, K = 0
# Output: 6
# Explanation: There are 6 possible playlists
# . [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2, 1], [2, 1, 2], [1, 2, 2]

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

class Solution(object):
    def numMusicPlaylists(self, N, L, K):
        """
        :type N: int
        :type L: int
        :type K: int
        :rtype: int
        """
        M = 10**9+7
        dp = [[0 for _ in xrange(1+L)] for _ in xrange(2)]
        dp[0][0] = dp[1][1] = 1
        for n in xrange(1, N+1):
            dp[n % 2][n] = (dp[(n-1) % 2][n-1] * n) % M
            for l in xrange(n+1, L+1):
                dp[n % 2][l] = ((dp[n % 2][l-1] * max(n-K, 0)) % M + \
                                (dp[(n-1) % 2][l-1] * n) % M) % M
        return dp[N % 2][L]
```

number-of-substrings-containing-all-three-characters.py

```python
# number-of-substrings-containing-all-three-characters is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numberOfSubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, left = 0, [-1]*3
        for right, c in enumerate(s):
            left[ord(c)-ord('a')] = right
            result += min(left)+1
        return result


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def numberOfSubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, left, count = 0, 0, [0]*3
        for right, c in enumerate(s):
            count[ord(s[right])-ord('a')] += 1
            while all(count):
                count[ord(s[left])-ord('a')] -= 1
                left += 1
            result += left
        return result


# Time:  O(n)
# Space: O(1)
class Solution3(object):
    def numberOfSubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, right, count = 0, 0, [0]*3
        for left, c in enumerate(s):
            while right < len(s) and not all(count):
                count[ord(s[right])-ord('a')] += 1
                right += 1
            if all(count):
                result += (len(s)-1) - (right-1) + 1
            count[ord(c)-ord('a')] -= 1
        return result
```

convert-binary-search-tree-to-sorted-doubly-linked-list.py

```python
# convert-binary-search-tree-to-sorted-doubly-linked-list is not found.
# Time:  O(n)
# Space: O(h)

class Node(object):
    def __init__(self, val, left, right):
        self.val = val
        self.left = left
        self.right = right


class Solution(object):
    def treeToDoublyList(self, root):
        """
        :type root: Node
        :rtype: Node
        """
        if not root:
            return None
        left_head, left_tail, right_head, right_tail = root, root, root, root
        if root.left:
            left_head = self.treeToDoublyList(root.left)
            left_tail = left_head.left
        if root.right:
            right_head = self.treeToDoublyList(root.right)
            right_tail = right_head.left
        left_tail.right, right_head.left = root, root
        root.left, root.right = left_tail, right_head
        left_head.left, right_tail.right = right_tail, left_head
        return left_head
```

critical-connections-in-a-network.py

```python
# critical-connections-in-a-network is not found.
# Time:  O(|V| + |E|)
# Space: O(|V| + |E|)

# variant of Tarjan's algorithm (https://www.geeksforgeeks.org/bridge-in-a-graph/)
class Solution(object):
    def criticalConnections(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: List[List[int]]
        """
        def dfs(edges, parent, u, idx, lowlinks, lookup, result):
            if lookup[u]:
                return
            lookup[u] = True
            curr_idx = lowlinks[u] = idx[0]
            idx[0] += 1
            for v in edges[u]:
                if v == parent:
                    continue
                dfs(edges, u, v, idx, lowlinks, lookup, result)
                lowlinks[u] = min(lowlinks[u], lowlinks[v])
                if lowlinks[v] > curr_idx:
                    # if any lowlink of neighbors is larger than curr_idx
                    result.append([u, v])

        edges = [[] for _ in xrange(n)]
        idx, lowlinks, lookup = [0], [0]*n, [False]*n
        result = []
        for u, v in connections:
            edges[u].append(v)
            edges[v].append(u)
        dfs(edges, -1, 0, idx, lowlinks, lookup, result)
        return result
```

cinema-seat-allocation.py

```
# cinema-seat-allocation is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def maxNumberOfFamilies(self, n, reservedSeats):
        """
        :type n: int
        :type reservedSeats: List[List[int]]
        :rtype: int
        """
        lookup = collections.defaultdict(lambda: [False]*3)
        for r, c in reservedSeats:
            if 2 <= c <= 5:
                lookup[r][0] = True
            if 4 <= c <= 7:
                lookup[r][1] = True
            if 6 <= c <= 9:
                lookup[r][2] = True
        result = 2*n
        for a, b, c in lookup.itervalues():
            if not a and not c:
                continue
            if not a or not b or not c:
                result -= 1
                continue
            result -= 2
        return result


# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def maxNumberOfFamilies(self, n, reservedSeats):
        """
        :type n: int
        :type reservedSeats: List[List[int]]
        :rtype: int
        """
        reservedSeats.sort()
        result, i = 2*n, 0
        while i < len(reservedSeats):
            reserved = [False]*3
            curr = reservedSeats[i][0]
            while i < len(reservedSeats) and reservedSeats[i][0] == curr:
                _, c  = reservedSeats[i]
                if 2 <= c <= 5:
                    reserved[0] = True
                if 4 <= c <= 7:
                    reserved[1] = True
                if 6 <= c <= 9:
                    reserved[2] = True
                i += 1
            if not reserved[0] and not reserved[2]:
                continue
```

```
    if not all(reserved):
        result -= 1
        continue
    result -= 2
return result
```

ugly-number-iii.py

```
# DESC
# Example 3:
# 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 2:
# Example 4:
# Constraints:
# Example 1:

# NOTE
# 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]

# EXAMPLE
# Input: n = 1000000000, a = 2, b = 217983653, c = 336916467
# Output: 1999999984
# 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.
# 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.
# 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.

# 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):
            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

```
# DESC
# Example 1:
# Note:
#
# The input is an array of integers. Only the least significant 8 bits of e
# ach integer is used to store the data. This means each integer represents only 1
#  byte of data.
# This is how the UTF-8 encoding would work:
# Given an array of integers representing the data, return whether it is a valid u
# tf-8 encoding.
# A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:
# Example 2:

# NOTE
# For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followe
# d by n-1 bytes with most significant 2 bits being 10.
# For 1-byte character, the first bit is a 0, followed by its unicode code.

# EXAMPLE
# data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 000
# 00001.
#
# Return true.
# It is a valid utf-8 encoding for a 2-bytes character follow
# ed by a 1-byte character.
# Char. number range  |        UTF-8 octet sequence
#       (hexadecimal)    |
#         (binary)
#    ---------------------+---------------------------------------
# ------
#    0000 0000-0000 007F | 0xxxxxxx
#    0000 0080-0000 07FF | 110xxxxx 10xxx
# xxx
#    0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
#    0001 0000-0010 FFFF |
#  11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
# data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 00
# 000100.
#
# 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 wi
# th 10 and that's correct.
# But the second continuation byte does not start with 1
# 0, so it is invalid.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def validUtf8(self, data):
        """
        :type data: List[int]
        :rtype: bool
        """
        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
```

largest-sum-of-averages.py

```
# DESC
# We partition a row of numbers A into at most K adjacent (non-empty) groups, then
#  our score is the sum of the average of each group. What is the largest score we
#  can achieve?
# Note:
# Note that our partition must use every number in A, and that scores are not nece
# ssarily integers.

# NOTE
# Answers within 10^-6 of the correct answer will be accepted as correct.
# 1 <= K <= A.length.
# 1 <= A.length <= 100.
# 1 <= A[i] <= 10000.

# EXAMPLE
# Example:
# Input:
# A = [9,1,2,3,9]
# K = 3
# Output: 20
# Explanation:
# The best choice
# is to partition A into [9], [1, 2, 3], [9]. The answer is 9 + (1 + 2 + 3) / 3 +
# 9 = 20.
# We could have also partitioned A into [9, 1], [2], [3, 9], for example.
#
# That partition would lead to a score of 5 + 2 + 6 = 13, which is worse.

# Time:  O(k * n^2)
# Space: O(n)


class Solution(object):
    def largestSumOfAverages(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: float
        """
        accum_sum = [A[0]]
        for i in xrange(1, len(A)):
            accum_sum.append(A[i]+accum_sum[-1])

        dp = [[0]*len(A) for _ in xrange(2)]
        for k in xrange(1, K+1):
            for i in xrange(k-1, len(A)):
                if k == 1:
                    dp[k % 2][i] = float(accum_sum[i])/(i+1)
                else:
                    for j in xrange(k-2, i):
                        dp[k % 2][i] = \
                            max(dp[k % 2][i],
                                dp[(k-1) % 2][j] +
                                float(accum_sum[i]-accum_sum[j])/(i-j))
        return dp[K % 2][-1]
```

max-difference-you-can-get-from-changing-an-integer.py

```python
# max-difference-you-can-get-from-changing-an-integer is not found.
# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def maxDiff(self, num):
        """
        :type num: int
        :rtype: int
        """
        digits = str(num)
        for b in digits:
            if b < '9':
                break
        if digits[0] != '1':
            a = digits[0]
        else:
            for a in digits:
                if a > '1':
                    break
        return int(digits.replace(b, '9')) - \
               int(digits.replace(a, '1' if digits[0] != '1' else '0'))
```

search-in-a-binary-search-tree.py

```
# DESC
# For example,
# Note that an empty tree is represented by NULL, therefore you would see the expe
# cted output (serialized tree format) as [], not null.
# In the example above, if we want to search the value 5, since there is no node w
# ith value 5, we should return NULL.
# Given the root node of a binary search tree (BST) and a value. You need to find
# the node in the BST that the node's value equals the given value. Return the sub
# tree rooted with that node. If such node doesn't exist, you should return NULL.
# You should return this subtree:

# NOTE
#

# EXAMPLE
# 2
#      / \
#     1   3
# Given the tree:
#         4
#        / \
#       2   7
#      / \
#     1   3
#
# And the val
# ue to search: 2

# Time:  O(h)
# Space: O(1)

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


class Solution(object):
    def searchBST(self, root, val):
        """
        :type root: TreeNode
        :type val: int
        :rtype: TreeNode
        """
        while root and val != root.val:
            if val < root.val:
                root = root.left
            else:
                root = root.right
        return root
```

longest-increasing-subsequence.py

```
# DESC
# Follow up: Could you improve it to O(n log n) time complexity?
# Note:
# Example:
# Given an unsorted array of integers, find the length of longest increasing subse
# quence.

# NOTE
# Your algorithm should run in O(n2) complexity.
# There may be more than one LIS combination, it is only necessary for you to retu
# rn the length.

# EXAMPLE
# Input: [10,9,2,5,3,7,101,18]
# Output: 4
# Explanation: The longest increasing subs
# equence is [2,3,7,101], therefore the length is 4.

# 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]
```

```
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
```

people-whose-list-of-favorite-companies-is-not-a-subset-of-another-list.py

```python
# people-whose-list-of-favorite-companies-is-not-a-subset-of-another-list is not found.
# Time:  O(n * m * l + n^2 * m), n is favoriteCompanies.length
#                              , m is the max of favoriteCompanies[i].length
#                              , l is the max of favoriteCompanies[i][j].length
# Space: O(n * m * l)

class Solution(object):
    def peopleIndexes(self, favoriteCompanies):
        """
        :type favoriteCompanies: List[List[str]]
        :rtype: List[int]
        """
        lookup, comps = {}, []
        for cs in favoriteCompanies:
            comps.append(set())
            for c in cs:
                if c not in lookup:
                    lookup[c] = len(lookup)
                comps[-1].add(lookup[c])
        return [i for i, c1 in enumerate(comps)
                if not any(i != j and len(c1) < len(c2) and c1 < c2
                           for j, c2 in enumerate(comps))]



# Time:  O(n * m * l + n^2 * m * log*(n)), n is favoriteCompanies.length
#                                        , m is the max of favoriteCompanies[i].length
#                                        , l is the max of favoriteCompanies[i][j].length
# Space: O(n * m * l)
class UnionFind(object):
    def __init__(self, data):
        self.data = [set(d) for d in data]
        self.set = range(len(data))

    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
        if len(self.data[x_root]) > len(self.data[y_root]) and \
           self.data[x_root] > self.data[y_root]:
            self.set[y_root] = x_root
        elif len(self.data[x_root]) < len(self.data[y_root]) and \
           self.data[x_root] < self.data[y_root]:
            self.set[x_root] = y_root


class Solution2(object):
    def peopleIndexes(self, favoriteCompanies):
        """
        :type favoriteCompanies: List[List[str]]
        :rtype: List[int]
        """
        lookup, comps = {}, []
```

```
for cs in favoriteCompanies:
    comps.append(set())
    for c in cs:
        if c not in lookup:
            lookup[c] = len(lookup)
        comps[-1].add(lookup[c])
union_find = UnionFind(comps)
for i in xrange(len(comps)):
    for j in xrange(len(comps)):
        if j == i:
            continue
        union_find.union_set(i, j)
return [x for i, x in enumerate(union_find.set) if x == i]
```

print-in-order.py

```
# DESC
# Note:
# Example 2:
# We do not know how the threads will be scheduled in the operating system, even t
# hough the numbers in the input seems to imply the ordering. The input format you
#  see is mainly to ensure our tests' comprehensiveness.
# The same instance of Foo will be passed to three different threads. Thread A wil
# l call first(), thread B will call second(), and thread C will call third(). Des
# ign a mechanism and modify the program to ensure that second() is executed after
#  first(), and third() is executed after second().
# Example 1:
# Suppose we have a class:

# NOTE
#

# EXAMPLE
# public class Foo {
#   public void first() { print("first"); }
#   public void secon
# d() { print("second"); }
#   public void third() { print("third"); }
# }
# Input: [1,3,2]
# Output: "firstsecondthird"
# Explanation: The input [1,3,2] means t
# hread A calls first(), thread B calls third(), and thread C calls second(). "fir
# stsecondthird" is the correct output.
# Input: [1,2,3]
# Output: "firstsecondthird"
# Explanation: There are three threads b
# eing fired asynchronously. The input [1,2,3] means thread A calls first(), threa
# d B calls second(), and thread C calls third(). "firstsecondthird" is the correc
# t output.

# Time:  O(n)
# Space: O(1)

import threading


class Foo(object):
    def __init__(self):
        self.__cv = threading.Condition()
        self.__has_first = False
        self.__has_second = False

    def first(self, printFirst):
        """
        :type printFirst: method
        :rtype: void
        """
        with self.__cv:
            # printFirst() outputs "first". Do not change or remove this line.
            printFirst()
            self.__has_first = True
            self.__cv.notifyAll()
```

```
def second(self, printSecond):
    """
    :type printSecond: method
    :rtype: void
    """
    with self.__cv:
        while not self.__has_first:
            self.__cv.wait()
        # printSecond() outputs "second". Do not change or remove this line.
        printSecond()
        self.__has_second = True
        self.__cv.notifyAll()

def third(self, printThird):
    """
    :type printThird: method
    :rtype: void
    """
    with self.__cv:
        while not self.__has_second:
            self.__cv.wait()
        # printThird() outputs "third". Do not change or remove this line.
        printThird()
        self.__cv.notifyAll()
```

last-stone-weight-ii.py

```python
# DESC
# 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:
# We have a collection of rocks, each rock has a positive integer weight.
# Note:
# Example 1:
# 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.)

# NOTE
# 1 <= stones[i] <= 100
# 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.
# 1 <= stones.length <= 30

# EXAMPLE
# 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 a
# rray converts to [2,1,1,1] then,
# we can combine 2 and 1 to get 1 so the array co
# nverts 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.

# 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)
```

partition-to-k-equal-sum-subsets.py

```
# DESC
# Given an array of integers nums and a positive integer k, find whether it's poss
# ible to divide this array into k non-empty subsets whose sums are all equal.
# Example 1:
# Note:

# NOTE
# 1 <= k <= len(nums) <= 16.
# 0 < nums[i] < 10000.

# EXAMPLE
# Input: nums = [4, 3, 2, 3, 5, 2, 1], k = 4
# Output: True
# Explanation: It's possib
# le to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.

# Time:  O(n*2^n)
# Space: O(2^n)

class Solution(object):
    def canPartitionKSubsets(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: bool
        """
        def dfs(nums, target, used, todo, lookup):
            if lookup[used] is None:
                targ = (todo-1)%target + 1
                lookup[used] = any(dfs(nums, target, used | (1<<i), todo-num, lookup) \
                                   for i, num in enumerate(nums) \
                                   if ((used>>i) & 1) == 0 and num <= targ)
            return lookup[used]

        total = sum(nums)
        if total%k or max(nums) > total//k:
            return False
        lookup = [None] * (1 << len(nums))
        lookup[-1] = True
        return dfs(nums, total//k, 0, total, lookup)


# Time:  O(k^(n-k) * k!)
# Space: O(n)
# DFS solution with pruning.
class Solution2(object):
    def canPartitionKSubsets(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: bool
        """
        def dfs(nums, target, i, subset_sums):
            if i == len(nums):
                return True
            for k in xrange(len(subset_sums)):
                if subset_sums[k]+nums[i] > target:
                    continue
```

```
            subset_sums[k] += nums[i]
            if dfs(nums, target, i+1, subset_sums):
                return True
            subset_sums[k] -= nums[i]
            if not subset_sums[k]: break
        return False

    total = sum(nums)
    if total%k != 0 or max(nums) > total//k:
        return False
    nums.sort(reverse=True)
    subset_sums = [0] * k
    return dfs(nums, total//k, 0, subset_sums)
```

exam-room.py

```
# DESC
# When a student enters the room, they must sit in the seat that maximizes the dis
# tance to the closest person.  If there are multiple such seats, they sit in the
# seat with the lowest number.  (Also, if no one is in the room, then the student
# sits at seat number 0.)
# Example 1:
# Note:
# In an exam room, there are N seats in a single row, numbered 0, 1, 2, ..., N-1.
# Return a class ExamRoom(int N) that exposes two functions: ExamRoom.seat() retur
# ning an int representing what seat the student sat in, and ExamRoom.leave(int p)
#  representing that the student in seat number p now leaves the room.  It is guar
# anteed that any calls to ExamRoom.leave(p) have a student sitting in seat p.
#

# NOTE
# 1 <= N <= 10^9
# Calls to ExamRoom.leave(p) are guaranteed to have a student currently sitting in
#  seat number p.
# ExamRoom.seat() and ExamRoom.leave() will be called at most 10^4 times across al
# l test cases.

# EXAMPLE
# Input: ["ExamRoom","seat","seat","seat","seat","leave","seat"], [[10],[],[],[],[
# ],[4],[]]
# Output: [null,0,9,4,2,null,5]
# Explanation:
# ExamRoom(10) -> null
# seat()
#  -> 0, no one is in the room, then the student sits at seat number 0.
# seat() ->
# 9, the student sits at the last seat number 9.
# seat() -> 4, the student sits at
# the last seat number 4.
# seat() -> 2, the student sits at the last seat number 2.
#
# leave(4) -> null
# seat() -> 5, the student sits at the last seat number 5.

# Time:  seat:  O(logn), amortized
#        leave: O(logn)
# Space: O(n)

import heapq


class ExamRoom(object):

    def __init__(self, N):
        """
        :type N: int
        """
        self.__num = N
        self.__seats = {-1: [-1, self.__num], self.__num: [-1, self.__num]}
        self.__max_heap = [(-self.__distance((-1, self.__num)), -1, self.__num)]

    def seat(self):
        """
        :rtype: int
```

```
        """
        while self.__max_heap[0][1] not in self.__seats or \
              self.__max_heap[0][2] not in self.__seats or \
              self.__seats[self.__max_heap[0][1]][1] != self.__max_heap[0][2] or \
              self.__seats[self.__max_heap[0][2]][0] !=  self.__max_heap[0][1]:
            heapq.heappop(self.__max_heap)  # lazy deletion

        _, left, right = heapq.heappop(self.__max_heap)
        mid = 0 if left == -1 \
              else self.__num-1 if right == self.__num \
              else (left+right) // 2
        self.__seats[mid] = [left, right]
        heapq.heappush(self.__max_heap, (-self.__distance((left, mid)), left, mid))
        heapq.heappush(self.__max_heap, (-self.__distance((mid, right)), mid, right))
        self.__seats[left][1] = mid
        self.__seats[right][0] = mid
        return mid

    def leave(self, p):
        """
        :type p: int
        :rtype: void
        """
        left, right = self.__seats[p]
        self.__seats.pop(p)
        self.__seats[left][1] = right
        self.__seats[right][0] = left
        heapq.heappush(self.__max_heap, (-self.__distance((left, right)), left, right))

    def __distance(self, segment):
        return segment[1]-segment[0]-1 if segment[0] == -1 or segment[1] == self.__num \
               else (segment[1]-segment[0]) // 2
```

largest-number.py

```
# DESC
# Example 2:
# Note: The result may be very large, so you need to return a string instead of an
#  integer.
# Given a list of non negative integers, arrange them such that they form the larg
# est number.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: [3,30,34,5,9]
# Output: "9534330"
# Input: [10,2]
# Output: "210"

# 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'
```

bold-words-in-string.py

```
# bold-words-in-string is not found.
# Time:  O(n * l), n is the length of S, l is the average length of words
# Space: O(t)    , t is the size of trie

import collections
import functools


class Solution(object):
    def boldWords(self, words, S):
        """
        :type words: List[str]
        :type S: str
        :rtype: str
        """
        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()
        for i, word in enumerate(words):
            functools.reduce(dict.__getitem__, word, trie).setdefault("_end")

        lookup = [False] * len(S)
        for i in xrange(len(S)):
            curr = trie
            k = -1
            for j in xrange(i, len(S)):
                if S[j] not in curr:
                    break
                curr = curr[S[j]]
                if "_end" in curr:
                    k = j
            for j in xrange(i, k+1):
                lookup[j] = True

        result = []
        for i in xrange(len(S)):
            if lookup[i] and (i == 0 or not lookup[i-1]):
                result.append("<b>")
            result.append(S[i])
            if lookup[i] and (i == len(S)-1 or not lookup[i+1]):
                result.append("</b>")
        return "".join(result)


# Time:  O(n * d * l), l is the average length of words
# Space: O(n)
class Solution2(object):
    def boldWords(self, words, S):
        """
        :type words: List[str]
        :type S: str
        :rtype: str
        """
        lookup = [0] * len(S)
        for d in words:
            pos = S.find(d)
            while pos != -1:
                lookup[pos:pos+len(d)] = [1] * len(d)
                pos = S.find(d, pos+1)
```

```
result = []
for i in xrange(len(S)):
    if lookup[i] and (i == 0 or not lookup[i-1]):
        result.append("<b>")
    result.append(S[i])
    if lookup[i] and (i == len(S)-1 or not lookup[i+1]):
        result.append("</b>")
return "".join(result)
```

design-in-memory-file-system.py

```
# design-in-memory-file-system is not found.
# Time:  ls: O(l + klogk), l is the path length, k is the number of entries in the last level directory
#        mkdir: O(l)
#        addContentToFile: O(l + c), c is the content size
#        readContentFromFile: O(l + c)
# Space: O(n + s), n is the number of dir/file nodes, s is the total content size.

class TrieNode(object):

    def __init__(self):
        self.is_file = False
        self.children = {}
        self.content = ""

class FileSystem(object):

    def __init__(self):
        self.__root = TrieNode()


    def ls(self, path):
        """
        :type path: str
        :rtype: List[str]
        """
        curr = self.__getNode(path)

        if curr.is_file:
            return [self.__split(path, '/')[-1]]

        return sorted(curr.children.keys())


    def mkdir(self, path):
        """
        :type path: str
        :rtype: void
        """
        curr = self.__putNode(path)
        curr.is_file = False


    def addContentToFile(self, filePath, content):
        """
        :type filePath: str
        :type content: str
        :rtype: void
        """
        curr = self.__putNode(filePath)
        curr.is_file = True
        curr.content += content


    def readContentFromFile(self, filePath):
        """
        :type filePath: str
        :rtype: str
        """
```

```
    return self.__getNode(filePath).content


def __getNode(self, path):
    curr = self.__root
    for s in self.__split(path, '/'):
        curr = curr.children[s]
    return curr


def __putNode(self, path):
    curr = self.__root
    for s in self.__split(path, '/'):
        if s not in curr.children:
            curr.children[s] = TrieNode()
        curr = curr.children[s]
    return curr


def __split(self, path, delim):
    if path == '/':
        return []
    return path.split('/')[1:]
```

distinct-subsequences-ii.py

```
# DESC
# Since the result may be large, return the answer modulo 10^9 + 7.
# Example 3:
# Example 1:
# Example 2:
# 10^9 + 7
# Note:
# Given a string S, count the number of distinct, non-empty subsequences of S .

# NOTE
# 1 <= S.length <= 2000
# S contains only lowercase letters.

# EXAMPLE
# Input: "aba"
# Output: 6
# Explanation: The 6 distinct subsequences are "a", "b", "a
# b", "ba", "aa" and "aba".
# Input: "aaa"
# Output: 3
# Explanation: The 3 distinct subsequences are "a", "aa" an
# d "aaa".
# Input: "abc"
# Output: 7
# Explanation: The 7 distinct subsequences are "a", "b", "c
# ", "ab", "ac", "bc", and "abc".

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def distinctSubseqII(self, S):
        """
        :type S: str
        :rtype: int
        """
        M = 10**9 + 7
        result, dp = 0, [0]*26
        for c in S:
            result, dp[ord(c)-ord('a')] = 2*result-dp[ord(c)-ord('a')]+1, result+1
        return result % M
```

longest-zigzag-path-in-a-binary-tree.py

```python
# longest-zigzag-path-in-a-binary-tree is not found.
# 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 longestZigZag(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, result):
            if not node:
                return [-1, -1]
            left, right = dfs(node.left, result), dfs(node.right, result)
            result[0] = max(result[0], left[1]+1, right[0]+1)
            return [left[1]+1, right[0]+1]

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

max-stack.py

```
# max-stack is not found.
# Time:  push:    O(1)
#        pop:     O(n), there is no built-in SortedDict in python. If applied, it could be reduced to O(logn)
#        popMax:  O(n)
#        top:     O(1)
#        peekMax: O(1)
# Space: O(n), n is the number of values in the current stack

import collections


class MaxStack(object):

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.__idx_to_val = collections.defaultdict(int)
        self.__val_to_idxs = collections.defaultdict(list)
        self.__top = None
        self.__max = None


    def push(self, x):
        """
        :type x: int
        :rtype: void
        """
        idx = self.__val_to_idxs[self.__top][-1]+1 if self.__val_to_idxs else 0
        self.__idx_to_val[idx] = x
        self.__val_to_idxs[x].append(idx)
        self.__top = x
        self.__max = max(self.__max, x)


    def pop(self):
        """
        :rtype: int
        """
        val = self.__top
        self.__remove(val)
        return val


    def top(self):
        """
        :rtype: int
        """
        return self.__top


    def peekMax(self):
        """
        :rtype: int
        """
        return self.__max
```

```
def popMax(self):
    """
    :rtype: int
    """
    val = self.__max
    self.__remove(val)
    return val


def __remove(self, val):
    idx = self.__val_to_idxs[val][-1]
    self.__val_to_idxs[val].pop()
    if not self.__val_to_idxs[val]:
        del self.__val_to_idxs[val]
    del self.__idx_to_val[idx]
    if val == self.__top:
        self.__top = self.__idx_to_val[max(self.__idx_to_val.keys())] if self.__idx_to_val else None
    if val == self.__max:
        self.__max = max(self.__val_to_idxs.keys()) if self.__val_to_idxs else None
```

partition-array-into-disjoint-intervals.py

```python
# DESC
# Return the length of left after such a partitioning.  It is guaranteed that such
#  a partitioning exists.
# Given an array A, partition it into two (contiguous) subarrays left and right so
#  that:
# Example 2:
# Note:
# Example 1:

# NOTE
# 2 <= A.length <= 30000
# It is guaranteed there is at least one way to partition A as described.
# Every element in left is less than or equal to every element in right.
# left and right are non-empty.
# left has the smallest possible size.
# 0 <= A[i] <= 10^6

# EXAMPLE
# Input: [5,0,3,8,6]
# Output: 3
# Explanation: left = [5,0,3], right = [8,6]
# Input: [1,1,1,0,6,12]
# Output: 4
# Explanation: left = [1,1,1,0], right = [6,12]

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

class Solution(object):
    def partitionDisjoint(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        B = A[:]
        for i in reversed(xrange(len(A)-1)):
            B[i] = min(B[i], B[i+1])
        p_max = 0
        for i in xrange(1, len(A)):
            p_max = max(p_max, A[i-1])
            if p_max <= B[i]:
                return i
```

two-sum-less-than-k.py

```
# two-sum-less-than-k is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def twoSumLessThanK(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        A.sort()
        result = -1
        left, right = 0, len(A)-1
        while left < right:
            if A[left]+A[right] >= K:
                right -= 1
            else:
                result = max(result, A[left]+A[right])
                left += 1
        return result
```

range-addition-ii.py

```
# DESC
# You need to count and return the number of maximum integers in the matrix after
# performing all the operations.
# Example 1:
# Note:
# Operations are represented by a 2D array, and each operation is represented by a
# n array with two positive integers a and b, which means M[i][j] should be added
# by one for all 0 <= i < a and 0 <= j < b.
# Given an m * n matrix M initialized with all 0's and several update operations.

# NOTE
# The range of a is [1,m], and the range of b is [1,n].
# The range of operations size won't exceed 10,000.
# The range of m and n is [1,40000].

# EXAMPLE
# Input:
# m = 3, n = 3
# operations = [[2,2],[3,3]]
# Output: 4
# Explanation:
# Initiall
# y, M =
# [[0, 0, 0],
#  [0, 0, 0],
#  [0, 0, 0]]
#
# After performing [2,2], M =
# [[1, 1
# , 0],
#  [1, 1, 0],
#  [0, 0, 0]]
#
# After performing [3,3], M =
# [[2, 2, 1],
#  [2, 2,
# 1],
#  [1, 1, 1]]
#
# So the maximum integer in M is 2, and there are four of it in M
# . So return 4.

# Time:  O(p), p is the number of ops
# Space: O(1)

class Solution(object):
    def maxCount(self, m, n, ops):
        """
        :type m: int
        :type n: int
        :type ops: List[List[int]]
        :rtype: int
        """
        for op in ops:
            m = min(m, op[0])
            n = min(n, op[1])
        return m*n
```

longest-harmonious-subsequence.py

```
# DESC
# Example 1:
# We define a harmounious array as an array where the difference between its maxim
# um value and its minimum value is exactly 1.
# Note: The length of the input array will not exceed 20,000.
# Now, given an integer array, you need to find the length of its longest harmonio
# us subsequence among all its possible subsequences.

# NOTE
#

# EXAMPLE
# Input: [1,3,2,2,5,2,3,7]
# Output: 5
# Explanation: The longest harmonious subsequen
# ce is [3,2,2,2,3].

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

import collections


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

coloring-a-border.py

```
# DESC
# Example 3:
# Two squares belong to the same connected component if and only if they have the
# same color and are next to each other in any of the 4 directions.
# Given a 2-dimensional grid of integers, each value in the grid represents the co
# lor of the grid square at that location.
# Example 1:
# Given a square at location (r0, c0) in the grid and a color, color the border of
#  the connected component of that square with the given color, and return the fin
# al grid.
# Example 2:
# The border of a connected component is all the squares in the connected componen
# t that are either 4-directionally adjacent to a square not in the component, or
# on the boundary of the grid (the first or last row or column).
# Note:

# NOTE
# 1 <= color <= 1000
# 0 <= r0 < grid.length
# 0 <= c0 < grid[0].length
# 1 <= grid.length <= 50
# 1 <= grid[i][j] <= 1000
# 1 <= grid[0].length <= 50

# EXAMPLE
# Input: grid = [[1,1],[1,2]], r0 = 0, c0 = 0, color = 3
# Output: [[3, 3], [3, 2]]
# 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]]
# Input: grid = [[1,2,2],[2,3,2]], r0 = 0, c0 = 1, color = 3
# Output: [[1, 3, 3], [
# 2, 3, 3]]

# 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
```

possible-bipartition.py

```
# DESC
# Example 1:
# Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone
#  into two groups of any size.
# Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people
# numbered a and b into the same group.
# Example 3:
# Constraints:
# Return true if and only if it is possible to split everyone into two groups in t
# his way.
# Each person may dislike some other people, and they should not go into the same
# group.
# Example 2:

# NOTE
# dislikes[i].length == 2
# 0 <= dislikes.length <= 10000
# There does not exist i != j for which dislikes[i] == dislikes[j].
# 1 <= dislikes[i][j] <= N
# dislikes[i][0] < dislikes[i][1]
# 1 <= N <= 2000

# EXAMPLE
# Input: N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]
# Output: false
# Input: N = 4, dislikes = [[1,2],[1,3],[2,4]]
# Output: true
# Explanation: group1 [1
# ,4], group2 [2,3]
# Input: N = 3, dislikes = [[1,2],[1,3],[2,3]]
# Output: false

# Time:  O(|V| + |E|)
# Space: O(|V| + |E|)

import collections


class Solution(object):
    def possibleBipartition(self, N, dislikes):
        """
        :type N: int
        :type dislikes: List[List[int]]
        :rtype: bool
        """
        adj = [[] for _ in xrange(N)]
        for u, v in dislikes:
            adj[u-1].append(v-1)
            adj[v-1].append(u-1)

        color = [0]*N
        color[0] = 1
        q = collections.deque([0])
        while q:
            cur = q.popleft()
            for nei in adj[cur]:
                if color[nei] == color[cur]:
                    return False
```

```
            elif color[nei] == -color[cur]:
                continue
            color[nei] = -color[cur]
            q.append(nei)
    return True
```

validate-stack-sequences.py

```
# DESC
# Example 1:
# Example 2:
# Constraints:
# Given two sequences pushed and popped with distinct values, return true if and o
# nly if this could have been the result of a sequence of push and pop operations
# on an initially empty stack.

# NOTE
# 0 <= pushed[i], popped[i] < 1000
# pushed is a permutation of popped.
# pushed and popped have distinct values.
# 0 <= pushed.length == popped.length <= 1000

# EXAMPLE
# Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
# Output: true
# Explanation: We m
# ight do the following sequence:
# push(1), push(2), push(3), push(4), pop() -> 4,
#
# push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
# Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
# Output: false
# Explanation: 1 c
# annot 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)
```

find-winner-on-a-tic-tac-toe-game.py

```python
# find-winner-on-a-tic-tac-toe-game is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def tictactoe(self, moves):
        """
        :type moves: List[List[int]]
        :rtype: str
        """
        row, col = [[0]*3 for _ in xrange(2)], [[0]*3 for _ in xrange(2)]
        diag, anti_diag = [0]*2, [0]*2
        p = 0
        for r, c in moves:
            row[p][r] += 1
            col[p][c] += 1
            diag[p] += r == c
            anti_diag[p] += r+c == 2
            if 3 in (row[p][r], col[p][c], diag[p], anti_diag[p]):
                return "AB"[p]
            p ^= 1
        return "Draw" if len(moves) == 9 else "Pending"
```

4sum.py

```
# 4sum is not found.
# 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)
# 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)
```

maximum-distance-in-arrays.py

```python
# maximum-distance-in-arrays is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxDistance(self, arrays):
        """
        :type arrays: List[List[int]]
        :rtype: int
        """
        result, min_val, max_val = 0,  arrays[0][0], arrays[0][-1]
        for i in xrange(1, len(arrays)):
            result = max(result, \
                         max(max_val - arrays[i][0], \
                             arrays[i][-1] - min_val))
            min_val = min(min_val, arrays[i][0])
            max_val = max(max_val, arrays[i][-1])
        return result
```

132-pattern.py

```
# 132-pattern is not found.
# Time:  O(n)
# Space: O(n)


class Solution(object):
    def find132pattern(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        ak = float("-inf")
        st = []
        for i in reversed(xrange(len(nums))):
            if nums[i] < ak:
                return True
            else:
                while st and nums[i] > st[-1]:
                    ak = st.pop()
            st.append(nums[i])
        return False
```

course-schedule-iv.py

```
# course-schedule-iv is not found.
# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def checkIfPrerequisite(self, n, prerequisites, queries):
        """
        :type n: int
        :type prerequisites: List[List[int]]
        :type queries: List[List[int]]
        :rtype: List[bool]
        """
        def floydWarshall(n, graph):
            reachable = set(map(lambda x: x[0]*n+x[1], graph))
            for k in xrange(n):
                for i in xrange(n):
                    for j in xrange(n):
                        if i*n+j not in reachable and (i*n+k in reachable and k*n+j in reachable):
                            reachable.add(i*n+j)
            return reachable

        reachable = floydWarshall(n, prerequisites)
        return [i*n+j in reachable for i, j in queries]


# Time:  O(n * q)
# Space: O(p + n)
import collections


class Solution2(object):
    def checkIfPrerequisite(self, n, prerequisites, queries):
        """
        :type n: int
        :type prerequisites: List[List[int]]
        :type queries: List[List[int]]
        :rtyp
        """
        graph = collections.defaultdict(list)
        for u, v in prerequisites:
            graph[u].append(v)
        result = []
        for i, j in queries:
            stk, lookup = [i], set([i])
            while stk:
                node = stk.pop()
                for nei in graph[node]:
                    if nei in lookup:
                        continue
                    stk.append(nei)
                    lookup.add(nei)
            result.append(j in lookup)
        return result
```

bulb-switcher.py

```
# DESC
# Example:
# 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 tog
# gle every i bulb. For the n-th round, you only toggle the last bulb. Find how ma
# ny bulbs are on after n rounds.

# NOTE
#

# 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 th
# ree bulbs are [on, off, on].
# After third round, the three bulbs are [on, off, of
# f].
#
# 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))
```

range-sum-query-2d-mutable.py

```
# range-sum-query-2d-mutable is not found.
# Time:  ctor:   O(m * n)
#        update: O(logm * logn)
#        query:  O(logm * logn)
# Space: O(m * n)

class NumMatrix(object):
    def __init__(self, matrix):
        """
        initialize your data structure here.
        :type matrix: List[List[int]]
        """
        if not matrix:
            return
        self.__matrix = matrix
        self.__bit = [[0] * (len(self.__matrix[0]) + 1) \
                      for _ in xrange(len(self.__matrix) + 1)]
        for i in xrange(1, len(self.__bit)):
            for j in xrange(1, len(self.__bit[0])):
                self.__bit[i][j] = matrix[i-1][j-1] + self.__bit[i-1][j] + \
                                   self.__bit[i][j-1] - self.__bit[i-1][j-1]
        for i in reversed(xrange(1, len(self.__bit))):
            for j in reversed(xrange(1, len(self.__bit[0]))):
                last_i, last_j = i - (i & -i), j - (j & -j)
                self.__bit[i][j] = self.__bit[i][j] - self.__bit[i][last_j] - \
                                   self.__bit[last_i][j] + self.__bit[last_i][last_j]

    def update(self, row, col, val):
        """
        update the element at matrix[row,col] to val.
        :type row: int
        :type col: int
        :type val: int
        :rtype: void
        """
        if val - self.__matrix[row][col]:
            self.__add(row, col, val - self.__matrix[row][col])
            self.__matrix[row][col] = val

    def sumRegion(self, row1, col1, row2, col2):
        """
        sum of elements matrix[(row1,col1)..(row2,col2)], inclusive.
        :type row1: int
        :type col1: int
        :type row2: int
        :type col2: int
        :rtype: int
        """
        return self.__sum(row2, col2) - self.__sum(row2, col1 - 1) - \
               self.__sum(row1 - 1, col2) + self.__sum(row1 - 1, col1 - 1)

    def __sum(self, row, col):
        row += 1
        col += 1
        ret = 0
        i = row
        while i > 0:
            j = col
```

```
        while j > 0:
            ret += self.__bit[i][j]
            j -= (j & -j)
        i -= (i & -i)
    return ret

def __add(self, row, col, val):
    row += 1
    col += 1
    i = row
    while i <= len(self.__matrix):
        j = col
        while j <= len(self.__matrix[0]):
            self.__bit[i][j] += val
            j += (j & -j)
        i += (i & -i)
```

path-in-zigzag-labelled-binary-tree.py

```
# DESC
# In the odd numbered rows (ie., the first, third, fifth,...), the labelling is le
# ft to right, while in the even numbered rows (second, fourth, sixth,...), the la
# belling is right to left.
# In an infinite binary tree where every node has two children, the nodes are labe
# lled in row order.
# Given the label of a node in this tree, return the labels in the path from the r
# oot of the tree to the node with that label.
# Constraints:
# Example 1:
# Example 2:

# NOTE
# 1 <= label <= 10^6

# EXAMPLE
# Input: label = 26
# Output: [1,2,6,10,26]
# Input: label = 14
# Output: [1,3,4,14]

# 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
# DESC
# Given a palindromic string palindrome, replace exactly one character by any lowe
# rcase English letter so that the string becomes the lexicographically smallest p
# ossible string that isn't a palindrome.
# After doing so, return the final string.  If there is no way to do so, return th
# e empty string.
# Example 1:
# Constraints:
# Example 2:

# NOTE
# 1 <= palindrome.length <= 1000
# palindrome consists of only lowercase English letters.

# EXAMPLE
# Input: palindrome = "abccba"
# Output: "aaccba"
# Input: palindrome = "a"
# Output: ""

# 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 ""
```

available-captures-for-rook.py

```
# DESC
# Return the number of pawns the rook can capture in one move.
# Note:
# Example 3:
# On an 8 x 8 chessboard, there is one white rook.  There also may be empty square
# s, white bishops, and black pawns.  These are given as characters 'R', '.', 'B',
#  and 'p' respectively. Uppercase characters represent white pieces, and lowercas
# e characters represent black pieces.
# Example 2:
# The rook moves as in the rules of Chess: it chooses one of four cardinal directi
# ons (north, east, west, and south), then moves in that direction until it choose
# s to stop, reaches the edge of the board, or captures an opposite colored pawn b
# y moving to the same square it occupies.  Also, rooks cannot move into the same
# square as other friendly bishops.
# Example 1:

# NOTE
# There is exactly one cell with board[i][j] == 'R'
# board[i][j] is either 'R', '.', 'B', or 'p'
# board.length == board[i].length == 8

# EXAMPLE
# Input: [[".",".",".",".",".",".",".",".",."],[".","p","p","p","p","p",".","."],["."
# ,"p","p","B","p","p",".","."],[".","p","B","R","B","p",".","."],[".","p","p","B"
# ,"p","p",".",".","."],[".","p","p","p","p","p",".","."],[".",".",".",".",".",".",".","."
# ,"."],[".",".",".",".",".",".",".",".","."]]
# Output: 0
# Explanation:
# Bishops are blo
# cking the rook to capture any pawn.
# Input: [[".",".",".",".",".",".",".",".","."],[".",".",".","p",".",".",".","."],["."
# ,".",".","R",".",".",".","p"],[".",".",".",".",".",".",".","."],[".",".",".",".","."
# ,".",".",".",".","."],[".",".",".","p",".",".",".","."],[".",".",".",".",".",".",".","."
# ,"."],[".",".",".",".",".",".",".",".","."]]
# Output: 3
# Explanation:
# In this example
#  the rook is able to capture all the pawns.
# Input: [[".",".",".",".",".",".",".",".","."],[".",".",".","p",".",".",".","."],["."
# ,".",".","p",".",".",".",".","."],["p","p",".","R",".","p","B",".","."],[".",".",".",".","."
# ,".",".",".",".","."],[".",".",".","B",".",".",".",".","."],[".",".",".","p",".",".",".","."
# ,"."],[".",".",".",".",".",".",".",".","."]]
# Output: 3
# Explanation:
# The rook can ca
# pture the pawns at positions b5, d6 and f5.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def numRookCaptures(self, board):
        """
        :type board: List[List[str]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

```
r, c = None, None
for i in xrange(8):
    if r is not None:
        break
    for j in xrange(8):
        if board[i][j] == 'R':
            r, c = i, j
            break

result = 0
for d in directions:
    nr, nc = r+d[0], c+d[1]
    while 0 <= nr < 8 and 0 <= nc < 8:
        if board[nr][nc] == 'p':
            result += 1
        if board[nr][nc] != '.':
            break
        nr, nc= nr+d[0], nc+d[1]
return result
```

maximal-rectangle.py

```
# DESC
# Example:
# Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle con
# taining only 1's and return its area.

# NOTE
#

# EXAMPLE
# Input:
# [
#   ["1","0","1","0","0"],
#   ["1","0","1","1","1"],
#   ["1","1","1","1","1
# "],
#   ["1","0","0","1","0"]
# ]
# Output: 6

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

class Solution(object):
    def maximalRectangle(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        def largestRectangleArea(heights):
            increasing, area, i = [], 0, 0
            while i <= len(heights):
                if not increasing or (i < len(heights) and heights[i] > heights[increasing[-1]]):
                    increasing.append(i)
                    i += 1
                else:
                    last = increasing.pop()
                    if not increasing:
                        area = max(area, heights[last] * i)
                    else:
                        area = max(area, heights[last] * (i - increasing[-1] - 1 ))
            return area

        if not matrix:
            return 0

        result = 0
        heights = [0] * len(matrix[0])
        for i in xrange(len(matrix)):
            for j in xrange(len(matrix[0])):
                heights[j] = heights[j] + 1 if matrix[i][j] == '1' else 0
            result = max(result, largestRectangleArea(heights))

        return result


# Time:  O(n^2)
# Space: O(n)
# DP solution.
```

```
class Solution2(object):
    def maximalRectangle(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        if not matrix:
            return 0

        result = 0
        m = len(matrix)
        n = len(matrix[0])
        L = [0 for _ in xrange(n)]
        H = [0 for _ in xrange(n)]
        R = [n for _ in xrange(n)]

        for i in xrange(m):
            left = 0
            for j in xrange(n):
                if matrix[i][j] == '1':
                    L[j] = max(L[j], left)
                    H[j] += 1
                else:
                    L[j] = 0
                    H[j] = 0
                    R[j] = n
                    left = j + 1

            right = n
            for j in reversed(xrange(n)):
                if matrix[i][j] == '1':
                    R[j] = min(R[j], right)
                    result = max(result, H[j] * (R[j] - L[j]))
                else:
                    right = j

        return result
```

balance-a-binary-search-tree.py

```python
# balance-a-binary-search-tree is not found.
# 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)
            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)
```

```python
# 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))
```

make-array-strictly-increasing.py

```python
# make-array-strictly-increasing is not found.
# Time:  O(n^2 * logn)
# Space: O(n)

import collections
import bisect


class Solution(object):
    def makeArrayIncreasing(self, arr1, arr2):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :rtype: int
        """
        arr2 = sorted(set(arr2))
        dp = {0: -1}  # dp[min_cost] = end_with_val
        for val1 in arr1:
            next_dp = collections.defaultdict(lambda: float("inf"))
            for cost, val in dp.iteritems():
                if val < val1:
                    next_dp[cost] = min(next_dp[cost], val1)
                k = bisect.bisect_right(arr2, val)
                if k == len(arr2):
                    continue
                next_dp[cost+1] = min(next_dp[cost+1], arr2[k])
            dp = next_dp
            if not dp:
                return -1
        return min(dp.iterkeys())
```

flatten-a-multilevel-doubly-linked-list.py

```
# DESC
# We use the multilevel linked list from Example 1 above:
# Flatten the list so that all the nodes appear in a single-level, doubly linked l
# ist. You are given the head of the first level of the list.
# Constraints:
# Example 2:
# How multilevel linked list is represented in test case:
# The serialization of each level is as follows:
# You are given a doubly linked list which in addition to the next and previous po
# inters, it could have a child pointer, which may or may not point to a separate
# doubly linked list. These child lists may have one or more children of their own
# , and so on, to produce a multilevel data structure, as shown in the example bel
# ow.
# Example 1:
# Merging the serialization of each level and removing trailing nulls we obtain:
# To serialize all levels together we will add nulls in each level to signify no n
# ode connects to the upper node of the previous level. The serialization becomes:
# Example 3:

# NOTE
# Number of Nodes will not exceed 1000.
# 1 <= Node.val <= 10^5

# EXAMPLE
# Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
# Output: [1,2
# ,3,7,8,11,12,9,10,4,5,6]
# Explanation:
#
# The multilevel linked list in the input i
# s as follows:
#
#
#
# After flattening the multilevel linked list it becomes:
# Input: head = [1,2,null,3]
# Output: [1,3,2]
# Explanation:
#
# The input multilevel li
# nked list is as follows:
#
#   1---2---NULL
#   |
#   3---NULL
# [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
# [1,2,3,4,5,6,null]
# [7,8,9,10,null]
# [11,12,null]
# Input: head = []
# Output: []
# [1,2,3,4,5,6,null]
# [null,null,7,8,9,10,null]
# [null,11,12,null]
# 1---2---3---4---5---6--NULL
#          |
#          7---8---9---10--NULL
#
```

```
#   |
#              11--12--NULL

# Time:  O(n)
# Space: O(1)

class Node(object):
    def __init__(self, val, prev, next, child):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child


class Solution(object):
    def flatten(self, head):
        """
        :type head: Node
        :rtype: Node
        """
        curr = head
        while curr:
            if curr.child:
                curr_next = curr.next
                curr.child.prev = curr
                curr.next = curr.child
                last_child = curr
                while last_child.next:
                    last_child = last_child.next
                if curr_next:
                    last_child.next = curr_next
                    curr_next.prev = last_child
                curr.child = None
            curr = curr.next
        return head
```

verify-preorder-serialization-of-a-binary-tree.py

```
# DESC
# Example 2:
# Example 3:
# Given a string of comma separated values, verify whether it is a correct preorde
# r traversal serialization of a binary tree. Find an algorithm without reconstruc
# ting the tree.
# Example 1:
# Each comma separated value in the string must be either an integer or a characte
# r '#' representing null pointer.
# One way to serialize a binary tree is to use pre-order traversal. When we encoun
# ter a non-null node, we record the node's value. If it is a null node, we record
#  using a sentinel value such as #.
# "9,3,4,#,#,1,#,#,2,#,6,#,#"
# For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,
# #,#,2,#,6,#,#", where # represents a null node.
# You may assume that the input format is always valid, for example it could never
#  contain two consecutive commas such as "1,,3".

# NOTE
#

# EXAMPLE
# Input: "9,#,#,1"
# Output: false
# Input: "1,#"
# Output: false
# Input: "9,3,4,#,#,1,#,#,2,#,6,#,#"
# Output: true
# _9_
#     /   \
#    3     2
#   / \   / \
#  4   1  #  6
# / \ / \   / \
# # # # #   # #

# 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:
            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

```
# DESC
# Note:
# Example 1:

# NOTE
# The integers in the given array are in the range of [0, 1000].
# The length of the given array won't exceed 1000.

# EXAMPLE
# Input: [2,2,3,4]
# Output: 3
# Explanation:
# Valid combinations are:
# 2,3,4 (using th
# e first 2)
# 2,3,4 (using the second 2)
# 2,2,3

# 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
```

binary-tree-upside-down.py

```
# binary-tree-upside-down is not found.
# Time:  O(n)
# Space: O(1)

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 root of the upside down tree
    def upsideDownBinaryTree(self, root):
        p, parent, parent_right = root, None, None

        while p:
            left = p.left
            p.left = parent_right
            parent_right = p.right
            p.right = parent
            parent = p
            p = left

        return parent


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param root, a tree node
    # @return root of the upside down tree
    def upsideDownBinaryTree(self, root):
        return self.upsideDownBinaryTreeRecu(root, None)

    def upsideDownBinaryTreeRecu(self, p, parent):
        if p is None:
            return parent

        root = self.upsideDownBinaryTreeRecu(p.left, p)
        if parent:
            p.left = parent.right
        else:
            p.left = None
        p.right = parent

        return root
```

create-target-array-in-the-given-order.py

```python
# create-target-array-in-the-given-order is not found.
# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def createTargetArray(self, nums, index):
        """
        :type nums: List[int]
        :type index: List[int]
        :rtype: List[int]
        """
        for i in xrange(len(nums)):
            for j in xrange(i):
                if index[j] >= index[i]:
                    index[j] += 1
        result = [0]*(len(nums))
        for i in xrange(len(nums)):
            result[index[i]] = nums[i]
        return result


# Time:  O(n^2)
# Space: O(1)
import itertools


class Solution2(object):
    def createTargetArray(self, nums, index):
        """
        :type nums: List[int]
        :type index: List[int]
        :rtype: List[int]
        """
        result = []
        for i, x in itertools.izip(index, nums):
            result.insert(i, x)
        return result
```

second-minimum-node-in-a-binary-tree.py

```python
# DESC
# If no such second minimum value exists, output -1 instead.
# Example 1:
# Example 2:
# Given such a binary tree, you need to output the second minimum value in the set
#  made of all the nodes' value in the whole tree.
# Given a non-empty special binary tree consisting of nodes with the non-negative
# value, where each node in this tree has exactly two or zero sub-node. If the nod
# e has two sub-nodes, then this node's value is the smaller value among its two s
# ub-nodes. More formally, the property root.val = min(root.left.val, root.right.v
# al) always holds.

# NOTE
#

# EXAMPLE
# Input:
#     2
#    / \
#   2   5
#      / \
#     5   7
#
# Output: 5
# Explanation: The smal
# lest value is 2, the second smallest value is 5.
# Input:
#     2
#    / \
#   2   2
#
# Output: -1
# Explanation: The smallest value is 2, b
# ut there isn't any second smallest value.

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

import heapq


class Solution(object):
    def findSecondMinimumValue(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def findSecondMinimumValueHelper(root, max_heap, lookup):
            if not root:
                return
            if root.val not in lookup:
                heapq.heappush(max_heap, -root.val)
                lookup.add(root.val)
                if len(max_heap) > 2:
                    lookup.remove(-heapq.heappop(max_heap))
            findSecondMinimumValueHelper(root.left, max_heap, lookup)
            findSecondMinimumValueHelper(root.right, max_heap, lookup)
```

```
max_heap, lookup = [], set()
findSecondMinimumValueHelper(root, max_heap, lookup)
if len(max_heap) < 2:
    return -1
return -max_heap[0]
```

binary-gap.py

```
# binary-ga is not found.
# Time:  O(logn) = O(1) due to n is a 32-bit number
# Space: O(1)

class Solution(object):
    def binaryGap(self, N):
        """
        :type N: int
        :rtype: int
        """
        result = 0
        last = None
        for i in xrange(32):
            if (N >> i) & 1:
                if last is not None:
                    result = max(result, i-last)
                last = i
        return result
```

coin-change-2.py

```
# DESC
# You can assume that
# Example 3:
# Example 2:
# You are given coins of different denominations and a total amount of money. Writ
# e 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:
# Note:

# NOTE
# 1 <= coin <= 5000
# the answer is guaranteed to fit into signed 32-bit integer
# the number of coins is less than 500
# 0 <= amount <= 5000

# EXAMPLE
# 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
# Input: amount = 3, coins = [2]
# Output: 0
# Explanation: the amount of 3 cannot be
# made up just with coins of 2.
# Input: amount = 10, coins = [10]
# Output: 1

# 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]
```

rle-iterator.py

```
# DESC
# For example, we start with A = [3,8,0,9,2,5], which is a run-length encoding of
# the sequence [8,8,8,5,5].  This is because the sequence can be read as "three ei
# ghts, zero nines, two fives".
# Note:
# The iterator is initialized by RLEIterator(int[] A), where A is a run-length enc
# oding of some sequence.  More specifically, for all even i, A[i] tells us the nu
# mber of times that the non-negative integer value A[i+1] is repeated in the sequ
# ence.
# Write an iterator that iterates through a run-length encoded sequence.
# Example 1:
# RLEIterator(int[] A)
# The iterator supports one function: next(int n), which exhausts the next n eleme
# nts (n >= 1) and returns the last element exhausted in this way.  If there is no
#  element left to exhaust, next returns -1 instead.

# NOTE
# Each call to RLEIterator.next(int n) will have 1 <= n <= 10^9.
# 0 <= A[i] <= 10^9
# A.length is an even integer.
# 0 <= A.length <= 1000
# There are at most 1000 calls to RLEIterator.next(int n) per test case.

# EXAMPLE
# Input: ["RLEIterator","next","next","next","next"], [[[3,8,0,9,2,5]],[2],[1],[1]
# ,[2]]
# Output: [null,8,8,5,-1]
# Explanation:
# RLEIterator is initialized with RLEI
# terator([3,8,0,9,2,5]).
# This maps to the sequence [8,8,8,5,5].
# RLEIterator.next
# is then called 4 times:
#
# .next(2) exhausts 2 terms of the sequence, returning 8.
#   The remaining sequence is now [8, 5, 5].
#
# .next(1) exhausts 1 term of the sequ
# ence, returning 8.  The remaining sequence is now [5, 5].
#
# .next(1) exhausts 1 t
# erm of the sequence, returning 5.  The remaining sequence is now [5].
#
# .next(2)
# exhausts 2 terms, returning -1.  This is because the first term exhausted was 5,
#
# but the second term did not exist.  Since the last term exhausted does not exis
# t, we return -1.

# Time:  O(n)
# Space: O(1)

class RLEIterator(object):

    def __init__(self, A):
        """
        :type A: List[int]
        """
```

```
        self.__A = A
        self.__i = 0
        self.__cnt = 0

    def next(self, n):
        """
        :type n: int
        :rtype: int
        """
        while self.__i < len(self.__A):
            if  n > self.__A[self.__i] - self.__cnt:
                n -= self.__A[self.__i] - self.__cnt
                self.__cnt = 0
                self.__i += 2
            else:
                self.__cnt += n
                return self.__A[self.__i+1]
        return -1
```

redundant-connection-ii.py

```
# DESC
# Return an edge that can be removed so that the resulting graph is a rooted tree
# of N nodes.  If there are multiple answers, return the answer that occurs last i
# n the given 2D-array.
# Example 1:
# Example 2:
# The resulting graph is given as a 2D-array of edges.  Each element of edges is a
#  pair [u, v] that represents a directed edge connecting nodes u and v, where u i
# s a parent of child v.
# Note:
# The given input is a directed graph that started as a rooted tree with N nodes (
# with distinct values 1, 2, ..., N), with one additional directed edge added.  Th
# e added edge has two different vertices chosen from 1 to N, and was not an edge
# that already existed.
# In this problem, a rooted tree is a directed graph such that, there is exactly o
# ne node (the root) for which all other nodes are descendants of this node, plus
# every node has exactly one parent, except for the root node which has no parents
# .

# NOTE
# The size of the input 2D-array will be between 3 and 1000.
# Every integer represented in the 2D-array will be between 1 and N, where N is th
# e size of the input array.

# EXAMPLE
# Input: [[1,2], [1,3], [2,3]]
# Output: [2,3]
# Explanation: The given directed graph
#  will be like this:
#   1
#  / \
# v   v
# 2-->3
# Input: [[1,2], [2,3], [3,4], [4,1], [1,5]]
# Output: [4,1]
# Explanation: The given
# directed graph will be like this:
# 5 <- 1 -> 2
#      ^    |
#      |    v
#      4 <-
# 3

# Time:  O(nlog*n) ~= O(n), n is the length of the positions
# 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 findRedundantDirectedConnection(self, edges):
        """
        :type edges: List[List[int]]
        :rtype: List[int]
        """
        cand1, cand2 = [], []
        parent = {}
        for edge in edges:
            if edge[1] not in parent:
                parent[edge[1]] = edge[0]
            else:
                cand1 = [parent[edge[1]], edge[1]]
                cand2 = edge

        union_find = UnionFind(len(edges)+1)
        for edge in edges:
            if edge == cand2:
                continue
            if not union_find.union_set(*edge):
                return cand1 if cand2 else edge
        return cand2
```

subsets.py

```
# DESC
# Note: The solution set must not contain duplicate subsets.
# Given a set of distinct integers, nums, return all possible subsets (the power set).
# Example:

# NOTE
#

# 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


# 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:])
```

shortest-path-in-a-grid-with-obstacles-elimination.py

```python
# shortest-path-in-a-grid-with-obstacles-elimination is not found.
# Time:  O(m * n * k)
# Space: O(m * n)

# A* Search Algorithm without heap
class Solution(object):
    def shortestPath(self, grid, k):
        """
        :type grid: List[List[int]]
        :type k: int
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        def dot(a, b):
            return a[0]*b[0]+a[1]*b[1]

        def g(a, b):
            return abs(a[0]-b[0])+abs(a[1]-b[1])

        def a_star(grid, b, t, k):
            f, dh = g(b, t), 2
            closer, detour = [(b, k)], []
            lookup = {}
            while closer or detour:
                if not closer:
                    f += dh
                    closer, detour = detour, closer
                b, k = closer.pop()
                if b == t:
                    return f
                if b in lookup and lookup[b] >= k:
                    continue
                lookup[b] = k
                for dx, dy in directions:
                    nb = (b[0]+dx, b[1]+dy)
                    if not (0 <= nb[0] < len(grid) and 0 <= nb[1] < len(grid[0]) and
                            (grid[nb[0]][nb[1]] == 0 or k > 0) and
                            (nb not in lookup or lookup[nb] < k)):
                        continue
                    (closer if dot((dx, dy), (t[0]-b[0], t[1]-b[1])) > 0 else detour).append((nb, k-int(grid[n
            return -1

        return a_star(grid, (0, 0), (len(grid)-1, len(grid[0])-1), k)
```

guess-the-word.py

```
# DESC
# For each test case, you have 10 guesses to guess the word. At the end of any num
# ber of calls, if you have made 10 or less calls to master.guess and at least one
#  of these guesses was the secret, you pass the testcase.
# This problem is an interactive problem new to the LeetCode platform.
# You may call master.guess(word) to guess a word.  The guessed word should have t
# ype string and must be from the original list with 6 lowercase letters.
# This function returns an integer type, representing the number of exact matches
# (value and position) of your guess to the secret word.  Also, if your guess is n
# ot in the given wordlist, it will return -1 instead.
# Besides the example test case below, there will be 5 additional test cases, each
#  with 100 words in the word list.  The letters of each word in those testcases w
# ere chosen independently at random from 'a' to 'z', such that every word in the
# given word lists is unique.
# Note:  Any solutions that attempt to circumvent the judge will result in disqual
# ification.
# We are given a word list of unique words, each word is 6 letters long, and one w
# ord in this list is chosen as secret.

# NOTE
#

# EXAMPLE
# Example 1:
# Input: secret = "acckzz", wordlist = ["acckzz","ccbazz","eiowzz","abc
# czz"]
#
# Explanation:
#
# master.guess("aaaaaa") returns -1, because "aaaaaa" is not
# in wordlist.
# master.guess("acckzz") returns 6, because "acckzz" is secret and ha
# s all 6 matches.
# master.guess("ccbazz") returns 3, because "ccbazz" has 3 matche
# s.
# master.guess("eiowzz") returns 2, because "eiowzz" has 2 matches.
# master.gues
# s("abcczz") returns 4, because "abcczz" has 4 matches.
#
# We made 5 calls to maste
# r.guess and one of them was the secret, so we pass the test case.

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

import collections
import itertools


class Solution(object):
    def findSecretWord(self, wordlist, master):
        """
        :type wordlist: List[Str]
        :type master: Master
        :rtype: None
        """
        def solve(H, possible):
            min_max_group, best_guess = possible, None
```

```
            for guess in possible:
                groups = [[] for _ in xrange(7)]
                for j in possible:
                    if j != guess:
                        groups[H[guess][j]].append(j)
                max_group = max(groups, key=len)
                if len(max_group) < len(min_max_group):
                    min_max_group, best_guess = max_group, guess
            return best_guess

        H = [[sum(a == b for a, b in itertools.izip(wordlist[i], wordlist[j]))
              for j in xrange(len(wordlist))]
              for i in xrange(len(wordlist))]
        possible = range(len(wordlist))
        n = 0
        while possible and n < 6:
            guess = solve(H, possible)
            n = master.guess(wordlist[guess])
            possible = [j for j in possible if H[guess][j] == n]


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def findSecretWord(self, wordlist, master):
        """
        :type wordlist: List[Str]
        :type master: Master
        :rtype: None
        """
        def solve(H, possible):
            min_max_group, best_guess = possible, None
            for guess in possible:
                groups = [[] for _ in xrange(7)]
                for j in possible:
                    if j != guess:
                        groups[H[guess][j]].append(j)
                max_group = groups[0]
                if len(max_group) < len(min_max_group):
                    min_max_group, best_guess = max_group, guess
            return best_guess

        H = [[sum(a == b for a, b in itertools.izip(wordlist[i], wordlist[j]))
              for j in xrange(len(wordlist))]
              for i in xrange(len(wordlist))]
        possible = range(len(wordlist))
        n = 0
        while possible and n < 6:
            guess = solve(H, possible)
            n = master.guess(wordlist[guess])
            possible = [j for j in possible if H[guess][j] == n]
```

array-transformation.py

```python
# array-transformation is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def transformArray(self, arr):
        """
        :type arr: List[int]
        :rtype: List[int]
        """
        def is_changable(arr):
            return any(arr[i-1] > arr[i] < arr[i+1] or
                       arr[i-1] < arr[i] > arr[i+1]
                       for i in xrange(1, len(arr)-1))

        while is_changable(arr):
            new_arr = arr[:]
            for i in xrange(1, len(arr)-1):
                new_arr[i] += arr[i-1] > arr[i] < arr[i+1]
                new_arr[i] -= arr[i-1] < arr[i] > arr[i+1]
            arr = new_arr
        return arr
```

kth-smallest-element-in-a-sorted-matrix.py

```
# DESC
# Note:
#
# You may assume k is always valid, 1  k  n2.
# Example:
# Note that it is the kth smallest element in the sorted order, not the kth distin
# ct element.
# 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
#

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

# 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
```

squares-of-a-sorted-array.py

```
# squares-of-a-sorted-arra is not found.
# Time:  O(n)
# Space: O(1)

import bisect


class Solution(object):
    def sortedSquares(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        right = bisect.bisect_left(A, 0)
        left = right-1
        result = []
        while 0 <= left or right < len(A):
            if right == len(A) or \
               (0 <= left and A[left]**2 < A[right]**2):
                result.append(A[left]**2)
                left -= 1
            else:
                result.append(A[right]**2)
                right += 1
        return result
```

word-ladder.py

```
# DESC
# Example 2:
# 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:
# Example 1:
# Note:

# NOTE
# Each transformed word must exist in the word list.
# You may assume no duplicates in the word list.
# All words have the same length.
# Return 0 if there is no such transformation sequence.
# Only one letter can be changed at a time.
# All words contain only lowercase alphabetic characters.
# You may assume beginWord and endWord are non-empty and are not the same.

# EXAMPLE
# Input:
# beginWord = "hit"
# endWord = "cog"
# wordList = ["hot","dot","dog","lot","lo
# g"]
#
# Output: 0
#
# Explanation: The endWord "cog" is not in wordList, therefore no
# possible transformation.
# 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.

# 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
```

check-if-array-pairs-are-divisible-by-k.py

```
# check-if-array-pairs-are-divisible-by-k is not found.
# Time:  O(n)
# Space: O(k)

import collections


class Solution(object):
    def canArrange(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: bool
        """
        count = collections.Counter(i%k for i in arr)
        return (0 not in count or not count[0]%2) and \
                all(k-i in count and count[i] == count[k-i] for i in xrange(1, k) if i in count)
```

maximum-product-of-two-elements-in-an-array.py

```python
# maximum-product-of-two-elements-in-an-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxProduct(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        m1 = m2 = 0
        for num in nums:
            if num > m1:
                m1, m2 = num, m1
            elif num > m2:
                m2 = num
        return (m1-1)*(m2-1)
```

camelcase-matching.py

```
# DESC
# Example 1:
# Example 2:
# A query word matches a given pattern if we can insert lowercase letters to the p
# attern word so that it equals the query. (We may insert each character at any po
# sition, and may insert 0 characters.)
# Example 3:
# queries
# 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.
# Note:

# NOTE
# 1 <= pattern.length <= 100
# 1 <= queries.length <= 100
# All strings consists only of lower and upper case English letters.
# 1 <= queries[i].length <= 100

# EXAMPLE
# 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".
# Input: queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"
# ], pattern = "FB"
# Output: [true,false,true,true,false]
# Explanation:
# "FooBar" ca
# n be generated like this "F" + "oo" + "B" + "ar".
# "FootBall" can be generated li
# ke this "F" + "oot" + "B" + "all".
# "FrameBuffer" can be generated like this "F"
# + "rame" + "B" + "uffer".
# Input: queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"
# ], pattern = "FoBaT"
# Output: [false,true,false,false,false]
# Explanation:
# "FooBa
# rTest" can be generated like this "Fo" + "o" + "Ba" + "r" + "T" + "est".

# Time:  O(n * l), n is number of quries
#                , 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]
        """
        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
```

bulls-and-cows.py

```
# DESC
# Note: You may assume that the secret number and your friend's guess only contain
#  digits, and their lengths are always equal.
# You are playing the following Bulls and Cows game with your friend: You write do
# wn a number and ask your friend to guess what the number is. Each time your frie
# nd makes a guess, you provide a hint that indicates how many digits in said gues
# s match your secret number exactly in both digit and position (called "bulls") a
# nd how many digits match the secret number but locate in the wrong position (cal
# led "cows"). Your friend will use successive guesses and hints to eventually der
# ive the secret number.
# Example 1:
# Example 2:
# Please note that both secret number and friend's guess may contain duplicate digits.
# Write a function to return a hint according to the secret number and friend's gu
# ess, use A to indicate the bulls and B to indicate the cows.

# NOTE
#

# EXAMPLE
# Input: secret = "1807", guess = "7810"
#
# Output: "1A3B"
#
# Explanation: 1 bull and
# 3 cows. The bull is 8, the cows are 0, 1 and 7.
# Input: secret = "1123", guess = "0111"
#
# Output: "1A1B"
#
# Explanation: The 1st 1 i
# n friend's guess is a bull, the 2nd or 3rd 1 is a cow.

# Time:  O(n)
# Space: O(10) = O(1)

import operator


# One pass solution.
from collections import defaultdict, Counter
from itertools import izip, imap


class Solution(object):
    def getHint(self, secret, guess):
        """
        :type secret: str
        :type guess: str
        :rtype: str
        """
        A, B = 0, 0
        s_lookup, g_lookup = defaultdict(int), defaultdict(int)
        for s, g in izip(secret, guess):
            if s == g:
                A += 1
            else:
                if s_lookup[g]:
```

```
                    s_lookup[g] -= 1
                    B += 1
                else:
                    g_lookup[g] += 1
                if g_lookup[s]:
                    g_lookup[s] -= 1
                    B += 1
                else:
                    s_lookup[s] += 1

        return "%dA%dB" % (A, B)


# Two pass solution.
class Solution2(object):
    def getHint(self, secret, guess):
        """
        :type secret: str
        :type guess: str
        :rtype: str
        """
        A = sum(imap(operator.eq, secret, guess))
        B = sum((Counter(secret) & Counter(guess)).values()) - A
        return "%dA%dB" % (A, B)
```

encode-and-decode-tinyurl.py

```
# DESC
# TinyURL is a URL shortening service where you enter a URL such as https://leetco
# de.com/problems/design-tinyurl and it returns a short URL such as http://tinyurl
# .com/4e9iAk.
# encode
# Design the encode and decode methods for the TinyURL service. There is no restri
# ction on how your encode/decode algorithm should work. You just need to ensure t
# hat a URL can be encoded to a tiny URL and the tiny URL can be decoded to the or
# iginal URL.

# NOTE
#

# EXAMPLE
#

# 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/'

    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

```
# DESC
# Note:
# Bonus point if you are able to do this using only O(n) extra space, where n is t
# he total number of rows in the triangle.
# The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).
# For example, given the following triangle
# Given a triangle, find the minimum path sum from top to bottom. Each step you ma
# y move to adjacent numbers on the row below.

# NOTE
#

# EXAMPLE
# [
#      [2],
#     [3,4],
#    [6,5,7],
#   [4,1,8,3]
# ]

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)
```

pascals-triangle.py

```python
# pascals-triangle is not found.
# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    # @return a list of lists of integers
    def generate(self, numRows):
        result = []
        for i in xrange(numRows):
            result.append([])
            for j in xrange(i + 1):
                if j in (0, i):
                    result[i].append(1)
                else:
                    result[i].append(result[i - 1][j - 1] + result[i - 1][j])
        return result

    def generate2(self, numRows):
        if not numRows: return []
        res = [[1]]
        for i in range(1, numRows):
            res += [map(lambda x, y: x + y, res[-1] + [0], [0] + res[-1])]
        return res[:numRows]

    def generate3(self, numRows):
        """
        :type numRows: int
        :rtype: List[List[int]]
        """
        if numRows == 0: return []
        if numRows == 1: return [[1]]
        res = [[1], [1, 1]]

        def add(nums):
            res = nums[:1]
            for i, j in enumerate(nums):
                if i < len(nums) - 1:
                    res += [nums[i] + nums[i + 1]]
            res += nums[:1]
            return res

        while len(res) < numRows:
            res.extend([add(res[-1])])
        return res
```

find-k-length-substrings-with-no-repeated-characters.py

```python
# find-k-length-substrings-with-no-repeated-characters is not found.
# Time:  O(n)
# Space: O(k)

class Solution(object):
    def numKLenSubstrNoRepeats(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: int
        """
        result, i = 0, 0
        lookup = set()
        for j in xrange(len(S)):
            while S[j] in lookup:
                lookup.remove(S[i])
                i += 1
            lookup.add(S[j])
            result += j-i+1 >= K
        return result
```

get-equal-substrings-within-budget.py

```python
# get-equal-substrings-within-budget is not found.
# 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
```

check-if-a-string-can-break-another-string.py

```
# check-if-a-string-can-break-another-string is not found.
# Time:  O(n)
# Space: O(1)

import collections
import string


class Solution(object):
    def checkIfCanBreak(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: bool
        """
        def is_break(count1, count2):
            curr1, curr2 = 0, 0
            for c in string.ascii_lowercase:
                curr1 += count1[c]
                curr2 += count2[c]
                if curr1 < curr2:
                    return False
            return True

        count1, count2 = collections.Counter(s1), collections.Counter(s2)
        return is_break(count1, count2) or is_break(count2, count1)


# Time:  O(nlogn)
# Space: O(1)
import itertools


class Solution2(object):
    def checkIfCanBreak(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: bool
        """
        return not {1, -1}.issubset(set(cmp(a, b) for a, b in itertools.izip(sorted(s1), sorted(s2))))


# Time:  O(nlogn)
# Space: O(1)
import itertools


class Solution3(object):
    def checkIfCanBreak(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: bool
        """
        s1, s2 = sorted(s1), sorted(s2)
        return all(a >= b for a, b in itertools.izip(s1, s2)) or \
               all(a <= b for a, b in itertools.izip(s1, s2))
```

score-after-flipping-matrix.py

```
# DESC
# Note:
# Return the highest possible score.
# After making any number of moves, every row of this matrix is interpreted as a b
# inary number, and the score of the matrix is the sum of these numbers.
# Example 1:
# A move consists of choosing any row or column, and toggling each value in that r
# ow or column: changing all 0s to 1s, and all 1s to 0s.
# We have a two dimensional matrix A where each value is 0 or 1.

# NOTE
# 1 <= A[0].length <= 20
# A[i][j] is 0 or 1.
# 1 <= A.length <= 20

# EXAMPLE
# 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

# 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
```

path-sum.py

```
# DESC
# return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.
# Note: A leaf is a node with no children.
# Given a binary tree and a sum, determine if the tree has a root-to-leaf path suc
# h that adding up all the values along the path equals the given sum.
# Example:
# Given the below binary tree and sum = 22,

# NOTE
#

# EXAMPLE
# 5
#      / \
#     4   8
#    /   / \
#   11  13  4
#  /  \      \
# 7    2      1

# 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 boolean
    def hasPathSum(self, root, sum):
        if root is None:
            return False

        if root.left is None and root.right is None and root.val == sum:
            return True

        return self.hasPathSum(root.left, sum - root.val) or self.hasPathSum(root.right, sum - root.val)
```

sentence-screen-fitting.py

```
# sentence-screen-fitting is not found.
# Time:  O(r + n * c)
# Space: O(n)

class Solution(object):
    def wordsTyping(self, sentence, rows, cols):
        """
        :type sentence: List[str]
        :type rows: int
        :type cols: int
        :rtype: int
        """
        def words_fit(sentence, start, cols):
            if len(sentence[start]) > cols:
                return 0

            s, count = len(sentence[start]), 1
            i = (start + 1) % len(sentence)
            while s + 1 + len(sentence[i]) <= cols:
                s += 1 + len(sentence[i])
                count += 1
                i = (i + 1) % len(sentence)
            return count

        wc = [0] * len(sentence)
        for i in xrange(len(sentence)):
            wc[i] = words_fit(sentence, i, cols)

        words, start = 0, 0
        for i in xrange(rows):
            words += wc[start]
            start = (start + wc[start]) % len(sentence)
        return words / len(sentence)
```

magic-squares-in-grid.py

```
# DESC
# Example 1:
# Note:
# Given an grid of integers, how many 3 x 3 "magic square" subgrids are there?  (E
# ach subgrid is contiguous).
# A 3 x 3 magic square is a 3 x 3 grid filled with distinct numbers from 1 to 9 su
# ch that each row, column, and both diagonals all have the same sum.

# NOTE
# 1 <= grid[0].length <= 10
# 0 <= grid[i][j] <= 15
# 1 <= grid.length <= 10

# EXAMPLE
# Input: [[4,3,8,4],
#         [9,5,1,9],
#         [2,7,6,2]]
# Output: 1
# Explanation:
#
# The following subgrid is a 3 x 3 magic square:
# 438
# 951
# 276
#
# while this one is n
# ot:
# 384
# 519
# 762
#
# In total, there is only one magic square inside the given grid.

# Time:  O(m * n)
# Space: O(1)


class Solution(object):
    def numMagicSquaresInside(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        def magic(grid, r, c):
            expect = k * (k**2+1) // 2
            nums = set()
            min_num = float("inf")
            sum_diag, sum_anti = 0, 0
            for i in xrange(k):
                sum_diag += grid[r+i][c+i]
                sum_anti += grid[r+i][c+k-1-i]
                sum_r, sum_c = 0, 0
                for j in xrange(k):
                    min_num = min(min_num, grid[r+i][c+j])
                    nums.add(grid[r+i][c+j])
                    sum_r += grid[r+i][c+j]
                    sum_c += grid[r+j][c+i]
                if not (sum_r == sum_c == expect):
```

```
            return False
    return sum_diag == sum_anti == expect and \
        len(nums) == k**2 and \
        min_num == 1

k = 3
result = 0
for r in xrange(len(grid)-k+1):
    for c in xrange(len(grid[r])-k+1):
        if magic(grid, r, c):
            result += 1
return result
```

subarray-sums-divisible-by-k.py

```
# DESC
# Note:
# Example 1:
# Given an array A of integers, return the number of (contiguous, non-empty) subar
# rays that have a sum divisible by K.

# NOTE
# 1 <= A.length <= 30000
# -10000 <= A[i] <= 10000
# 2 <= K <= 10000

# EXAMPLE
# Input: A = [4,5,0,-2,-3,1], K = 5
# Output: 7
# Explanation: There are 7 subarrays w
# ith 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]

# 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
```

assign-cookies.py

```
# DESC
# Example 2:
# Example 1:
# Assume you are an awesome parent and want to give your children some cookies. Bu
# t, you should give each child at most one cookie. Each child i has a greed facto
# r gi, which is the minimum size of a cookie that the child will be content with;
#  and each cookie j has a size sj. If sj >= gi, we can assign the cookie j to the
#  child i, and the child i will be content. Your goal is to maximize the number o
# f your content children and output the maximum number.
# Note:
#
# You may assume the greed factor is always positive.
#
# You cannot assign m
# ore than one cookie to one child.

# NOTE
#

# EXAMPLE
# Input: [1,2], [1,2,3]
#
# Output: 2
#
# Explanation: You have 2 children and 3 cookies
# . The greed factors of 2 children are 1, 2.
# You have 3 cookies and their sizes
# are big enough to gratify all of the children,
# You need to output 2.
# Input: [1,2,3], [1,1]
#
# Output: 1
#
# Explanation: You have 3 children and 2 cookies
# . The greed factors of 3 children are 1, 2, 3.
# And even though you have 2 cooki
# es, since their size is both 1, you could only make the child whose greed factor
#  is 1 content.
# You need to output 1.

# Time:  O(nlogn)
# Space: O(1)


class Solution(object):
    def findContentChildren(self, g, s):
        """
        :type g: List[int]
        :type s: List[int]
        :rtype: int
        """
        g.sort()
        s.sort()

        result, i = 0, 0
        for j in xrange(len(s)):
            if i == len(g):
                break
```

```
    if s[j] >= g[i]:
        result += 1
        i += 1
return result
```

wildcard-matching.py

```
# DESC
# Example 4:
# Note:
# Example 3:
# Example 1:
# Given an input string (s) and a pattern (p), implement wildcard pattern matching
#  with support for '?' and '*'.
# Example 5:
# Example 2:
# The matching should cover the entire input string (not partial).

# NOTE
# s could be empty and contains only lowercase letters a-z.
# p could be empty and contains only lowercase letters a-z, and characters like ? or *.

# EXAMPLE
# Input:
# s = "aa"
# p = "*"
# Output: true
# Explanation: '*' matches any sequence.
# Input:
# s = "adceb"
# p = "*a*b"
# Output: true
# Explanation: The first '*' matches th
# e empty sequence, while the second '*' matches the substring "dce".
# Input:
# s = "acdcb"
# p = "a*c?b"
# Output: false
# Input:
# s = "aa"
# p = "a"
# Output: false
# Explanation: "a" does not match the entire
#  string "aa".
# '?' Matches any single character.
# '*' Matches any sequence of characters (includ
# ing the empty sequence).
# Input:
# s = "cb"
# p = "?a"
# Output: false
# Explanation: '?' matches 'c', but the sec
# ond letter is 'a', which does not match 'b'.

# Time:  O(m + n) ~ O(m * n)
# Space: O(1)

# iterative solution with greedy
class Solution(object):
    def isMatch(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: bool
        """
```

```
        count = 0  # used for complexity check
        p_ptr, s_ptr, last_s_ptr, last_p_ptr = 0, 0, -1, -1
        while s_ptr < len(s):
            if p_ptr < len(p) and (s[s_ptr] == p[p_ptr] or p[p_ptr] == '?'):
                s_ptr += 1
                p_ptr += 1
            elif p_ptr < len(p) and p[p_ptr] == '*':
                p_ptr += 1
                last_s_ptr = s_ptr
                last_p_ptr = p_ptr
            elif last_p_ptr != -1:
                last_s_ptr += 1
                s_ptr = last_s_ptr
                p_ptr = last_p_ptr
            else:
                assert(count <= (len(p)+1) * (len(s)+1))
                return False
            count += 1  # used for complexity check

        while p_ptr < len(p) and p[p_ptr] == '*':
            p_ptr += 1
            count += 1  # used for complexity check

        assert(count <= (len(p)+1) * (len(s)+1))
        return p_ptr == len(p)


# dp with rolling window
# Time:  O(m * n)
# Space: O(n)
class Solution2(object):
    # @return a boolean
    def isMatch(self, s, p):
        k = 2
        result = [[False for j in xrange(len(p) + 1)] for i in xrange(k)]

        result[0][0] = True
        for i in xrange(1, len(p) + 1):
            if p[i-1] == '*':
                result[0][i] = result[0][i-1]
        for i in xrange(1,len(s) + 1):
            result[i % k][0] = False
            for j in xrange(1, len(p) + 1):
                if p[j-1] != '*':
                    result[i % k][j] = result[(i-1) % k][j-1] and (s[i-1] == p[j-1] or p[j-1] == '?')
                else:
                    result[i % k][j] = result[i % k][j-1] or result[(i-1) % k][j]

        return result[len(s) % k][len(p)]


# dp
# Time:  O(m * n)
# Space: O(m * n)
class Solution3(object):
    # @return a boolean
    def isMatch(self, s, p):
        result = [[False for j in xrange(len(p) + 1)] for i in xrange(len(s) + 1)]

        result[0][0] = True
```

```
        for i in xrange(1, len(p) + 1):
            if p[i-1] == '*':
                result[0][i] = result[0][i-1]
        for i in xrange(1,len(s) + 1):
            result[i][0] = False
            for j in xrange(1, len(p) + 1):
                if p[j-1] != '*':
                    result[i][j] = result[i-1][j-1] and (s[i-1] == p[j-1] or p[j-1] == '?')
                else:
                    result[i][j] = result[i][j-1] or result[i-1][j]

        return result[len(s)][len(p)]


# recursive, slowest, TLE
class Solution4(object):
    # @return a boolean
    def isMatch(self, s, p):
        if not p or not s:
            return not s and not p

        if p[0] != '*':
            if p[0] == s[0] or p[0] == '?':
                return self.isMatch(s[1:], p[1:])
            else:
                return False
        else:
            while len(s) > 0:
                if self.isMatch(s, p[1:]):
                    return True
                s = s[1:]
            return self.isMatch(s, p[1:])
```

lowest-common-ancestor-of-deepest-leaves.py

```
# DESC
# Example 3:
# Example 1:
# Example 2:
# Constraints:
# Recall that:
# Given a rooted binary tree, return the lowest common ancestor of its deepest leaves.

# NOTE
# The depth of the root of the tree is 0, and if the depth of a node is d, the dep
# th 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 de
# pth such that every node in S is in the subtree with root A.
# The given tree will have between 1 and 1000 nodes.
# Each node of the tree will have a distinct value between 1 and 1000.
# The node of a binary tree is a leaf if and only if it has no children

# EXAMPLE
# Input: root = [1,2,3,4]
# Output: [4]
# Input: root = [1,2,3,4,5]
# Output: [2,4,5]
# Input: root = [1,2,3]
# Output: [1,2,3]
# Explanation:
# The deepest leaves are the n
# odes 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 seri
# alization "[1,2,3]".

# 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
        """
        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

```
# 3sum-with-multiplicit is not found.
# 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)
```

minimize-malware-spread-ii.py

```
# DESC
# Example 2:
# (This problem is the same as Minimize Malware Spread, with the differences bolded.)
# Note:
# We will remove one node from the initial list, completely removing it and any co
# nnections from this node to any other node.  Return the node that if removed, wo
# uld minimize M(initial).  If multiple nodes could be removed to minimize M(initi
# al), return such a node with the smallest index.
# Suppose M(initial) is the final number of nodes infected with malware in the ent
# ire network, after the spread of malware stops.
# Example 3:
# Example 1:
# Some nodes initial are initially infected by malware.  Whenever two nodes are di
# rectly connected and at least one of those two nodes is infected by malware, bot
# h nodes will be infected by malware.  This spread of malware will continue until
#  no more nodes can be infected in this manner.
# In a network of nodes, each node i is directly connected to another node j if an
# d only if graph[i][j] = 1.

# NOTE
# 1 <= initial.length < graph.length
# graph[i][i] = 1
# 0 <= initial[i] < graph.length
# 0 <= graph[i][j] == graph[j][i] <= 1
# 1 < graph.length = graph[0].length <= 300

# EXAMPLE
# Input: graph = [[1,1,0,0],[1,1,1,0],[0,1,1,1],[0,0,1,1]], initial = [0,1]
# Output: 1
# Input: graph = [[1,1,0],[1,1,1],[0,1,1]], initial = [0,1]
# Output: 1
# Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]
# Output: 0

# Time:  O(n^2)
# 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:
            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 minMalwareSpread(self, graph, initial):
    """
    :type graph: List[List[int]]
    :type initial: List[int]
    :rtype: int
    """
    initial_set = set(initial)
    clean = [i for i in xrange(len(graph)) if i not in initial_set]
    union_find = UnionFind(len(graph))
    for i in xrange(len(clean)):
        for j in xrange(i+1, len(clean)):
            if graph[clean[i]][clean[j]] == 1:
                union_find.union_set(clean[i], clean[j])
    union_size = collections.Counter(union_find.find_set(i) for i in xrange(len(graph)))

    shared_union = collections.defaultdict(set)
    for i in initial:
        for j in clean:
            if graph[i][j] == 1:
                x = union_find.find_set(j)
                shared_union[x].add(i)

    result, total = float("inf"), float("-inf")
    for i in initial:
        lookup = set()
        curr = 0
        for j in clean:
            if graph[i][j] == 1:
                x = union_find.find_set(j)
                if len(shared_union[x]) == 1 and \
                   x not in lookup:
                    curr += union_size[x]
                    lookup.add(x)
        if curr > total or \
           (curr == total and i < result):
            total = curr
            result = i
    return result
```

max-consecutive-ones-iii.py

```
# DESC
# Return the length of the longest (contiguous) subarray that contains only 1s.
# Note:
# Example 2:
# Given an array A of 0s and 1s, we may change up to K values from 0 to 1.
# Example 1:

# NOTE
# A[i] is 0 or 1
# 1 <= A.length <= 20000
# 0 <= K <= A.length

# EXAMPLE
# 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 unde
# rlined.
# 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.

# 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
# DESC
# Given two numbers arr1 and arr2 in base -2, return the result of adding them tog
# ether.
# Example 1:
# Return the result of adding arr1 and arr2 in the same format: as an array of 0s
# and 1s with no leading zeros.
# arr = [1,1,0,1]
# Note:
# Each number is given in array format:  as an array of 0s and 1s, from most signi
# ficant bit to least significant bit.  For example, arr = [1,1,0,1] represents th
# e number (-2)^3 + (-2)^2 + (-2)^0 = -3.  A number arr in array format is also gu
# aranteed to have no leading zeros: either arr == [0] or arr[0] == 1.

# NOTE
# 1 <= arr1.length <= 1000
# 1 <= arr2.length <= 1000
# arr1[i] is 0 or 1
# arr2[i] is 0 or 1
# arr1 and arr2 have no leading zeros

# EXAMPLE
# 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.

# 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
```

number-of-1-bits.py

```
# DESC
# Example 2:
# If this function is called many times, how would you optimize it?
# Write a function that takes an unsigned integer and return the number of '1' bit
# s it has (also known as the Hamming weight).
# Follow up:
# Example 3:
# Example 1:
# Note:

# NOTE
# Note that in some languages such as Java, there is no unsigned integer type. In
# this case, the input will be given as signed integer type and should not affect
# your implementation, as the internal binary representation of the integer is the
#  same whether it is signed or unsigned.
# In Java, the compiler represents the signed integers using 2's complement notati
# on. Therefore, in Example 3 above the input represents the signed integer -3.

# EXAMPLE
# Input: 00000000000000000000000000001011
# Output: 3
# Explanation: The input binary
# string 00000000000000000000000000001011 has a total of three '1' bits.
# Input: 11111111111111111111111111111101
# Output: 31
# Explanation: The input binary
#  string 11111111111111111111111111111101 has a total of thirty one '1' bits.
# Input: 00000000000000000000000010000000
# Output: 1
# Explanation: The input binary
# string 00000000000000000000000010000000 has a total of one '1' bit.

# Time:  O(32), bit shift in python is not O(1), it's O(k), k is the number of bits shifted
#             , see https://github.com/python/cpython/blob/2.7/Objects/longobject.c#L3652
# Space: O(1)

class Solution(object):
    # @param n, an integer
    # @return an integer
    def hammingWeight(self, n):
        n = (n & 0x55555555) + ((n >> 1) & 0x55555555)
        n = (n & 0x33333333) + ((n >> 2) & 0x33333333)
        n = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F)
        n = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF)
        n = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF)
        return n


# Time:  O(logn/4) = O(32/4 + 8*4) = O(32)
# Space: O(1)
# https://github.com/gcc-mirror/gcc/blob/master/libgcc/libgcc2.c#L856
class Solution2(object):
    def __init__(self):
        self.__popcount_tab = \
        [ \
            0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5, \
            1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6, \
            1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6, \
```

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

    # @param n, an integer
    # @return an integer
    def hammingWeight(self, n):
        result = 0
        while n:
            result += self.__popcount_tab[n & 0xff]
            n >>= 8
        return result


# Time:  O(logn) = O(32)
# Space: O(1)
class Solution3(object):
    # @param n, an integer
    # @return an integer
    def hammingWeight(self, n):
        result = 0
        while n:
            n &= n - 1
            result += 1
        return result
```

minimum-knight-moves.py

```
# minimum-knight-moves is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def minKnightMoves(self, x, y):
        """
        :type x: int
        :type y: int
        :rtype: int
        """
        # we can observe from:
        # [0]
        # [3, 2]
        # [2,(1),4]
        # [3, 2, 3, 2]
        # [2, 3,(2) 3, 4]
        # [3, 4, 3, 4, 3, 4]
        # [4, 3, 4,(3),4, 5, 4]
        # [5, 4, 5, 4, 5, 4, 5, 6]
        # [4, 5, 4, 5,(4),5, 6, 5, 6]
        # [5, 6, 5, 6, 5, 6, 5, 6, 7, 6]
        # [6, 5, 6, 5, 6,(5),6, 7, 6, 7, 8]
        # [7, 6, 7, 6, 7, 6, 7, 6, 7, 8, 7, 8]
        # [6, 7, 6, 7, 6, 7,(6),7, 8, 7, 8, 9, 8]
        # [7, 8, 7, 8, 7, 8, 7, 8, 7, 8, 9, 8, 9, 10]
        # [8, 7, 8, 7, 8, 7, 8,(7),8, 9, 8, 9, 10, 9, 10]
        # [9, 8, 9, 8, 9, 8, 9, 8, 9, 8, 9, 10, 9, 10, 11, 10]

        x, y = abs(x), abs(y)
        if x < y:
            x, y = y, x
        lookup = {(0, 0):0, (1, 0):3, (2, 2):4}  # special cases
        if (x, y) in lookup:
            return lookup[(x, y)]
        k = x-y
        if y > k:
            # if 2y > x, every period 3 of y (or k) with fixed another is increased by 2 (or 1)
            # and start from (2k, k) with (k) when y = k (diagonal line)
            # ex. (0, 0) ~ (12, 12) ~ ... : 0 => 2,4(special case),2 => 4,4,4 => 6,6,6 => 8,8,8 => ...
            # ex. (2, 1) ~ (14, 13) ~ ... : 1 => 3,3,3 => 5,5,5 => 7,7,7 => 9,9,9 => ...
            return k - 2*((k-y)//3)
        # if 2y <= x, every period 4 of k (or y) with fixed another is increased by 2
        # and start from (2k, k) with (k) when y = k (vertical line)
        # ex. (0, 0) ~ (11, 0) ~ ... : 0,3(special case),2,3 => 2,3,4,5 => 4,5,6,7 => ...
        # ex. (2, 1) ~ (13, 1) ~ ... : 1,2,3,4 => 3,4,5,6 => 5,6,7,8 => ...
        return k - 2*((k-y)//4)


# Time:  O(n^2)
# Space: O(n^2)
class Solution2(object):
    def __init__(self):
        self.__lookup = {(0, 0):0, (1, 1):2, (1, 0):3}  # special cases

    def minKnightMoves(self, x, y):
        """
        :type x: int
```

```
        :type y: int
        :rtype: int
        """
        def dp(x, y):
            x, y = abs(x), abs(y)
            if x < y:
                x, y = y, x
            if (x, y) not in self.__lookup:  # greedy, smaller x, y is always better if not special cases
                self.__lookup[(x, y)] = min(dp(x-1, y-2), dp(x-2, y-1)) + 1
            return self.__lookup[(x, y)]
        return dp(x, y)
```

di-string-match.py

```python
# DESC
# Example 2:
# Return any permutation A of [0, 1, ..., N] such that for all i = 0, ..., N-1:
# Example 3:
# Example 1:
# Note:
# Given a string S that only contains "I" (increase) or "D" (decrease), let N = S.
# length.

# NOTE
# If S[i] == "I", then A[i] < A[i+1]
# If S[i] == "D", then A[i] > A[i+1]
# S only contains characters "I" or "D".
# 1 <= S.length <= 10000

# EXAMPLE
# Input: "IDID"
# Output: [0,4,1,3,2]
# Input: "III"
# Output: [0,1,2,3]
# Input: "DDI"
# Output: [3,2,0,1]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def diStringMatch(self, S):
        """
        :type S: str
        :rtype: List[int]
        """
        result = []
        left, right = 0, len(S)
        for c in S:
            if c == 'I':
                result.append(left)
                left += 1
            else:
                result.append(right)
                right -= 1
        result.append(left)
        return result
```

minimum-number-of-increments-on-subarrays-to-form-a-target-array.py

```python
# minimum-number-of-increments-on-subarrays-to-form-a-target-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minNumberOperations(self, target):
        """
        :type target: List[int]
        :rtype: int
        """
        return target[0]+sum(max(target[i]-target[i-1], 0) for i in xrange(1, len(target)))


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


class Solution2(object):
    def minNumberOperations(self, target):
        """
        :type target: List[int]
        :rtype: int
        """
        return sum(max(b-a, 0) for b, a in itertools.izip(target, [0]+target))
```

map-sum-pairs.py

```
# DESC
# For the method sum, you'll be given a string representing the prefix, and you ne
# ed to return the sum of all the pairs' value whose key starts with the prefix.
# insert
# For the method insert, you'll be given a pair of (string, integer). The string r
# epresents the key and the integer represents the value. If the key already exist
# ed, then the original key-value pair will be overridden to the new one.
# Implement a MapSum class with insert, and sum methods.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: insert("apple", 3), Output: Null
# Input: sum("ap"), Output: 3
# Input: inser
# t("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
```

```
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
# DESC
# 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 co
# lors an uncolored neighbor of the chosen node (either the left child, right chil
# d, or parent of the chosen node.)
# Initially, the first player names a value x with 1 <= x <= n, and the second pla
# yer names a value y with 1 <= y <= n and y != x.  The first player colors the no
# de with value x red, and the second player colors the node with value y blue.
# 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 t
# he player that colored more nodes.
# You are the second player.  If it is possible to choose such a y to ensure you w
# in the game, return true.  If it is not possible, return false.
# Constraints:
# Example 1:
# Two players play a turn based game on a binary tree.  We are given the root of t
# his 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.

# NOTE
# 1 <= x <= n <= 100
# root is the root of a binary tree with n nodes and distinct node values from 1 to n.
# n is odd.

# EXAMPLE
# 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.

# 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
```

convert-binary-number-in-a-linked-list-to-integer.py

```python
# convert-binary-number-in-a-linked-list-to-integer is not found.
# Time:  O(n)
# Space: O(1)

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


class Solution(object):
    def getDecimalValue(self, head):
        """
        :type head: ListNode
        :rtype: int
        """
        result = 0
        while head:
            result = result*2 + head.val
            head = head.next
        return result
```

lemonade-change.py

```
# DESC
# Example 1:
# Note:
# Example 4:
# Example 2:
# Return true if and only if you can provide every customer with correct change.
# At a lemonade stand, each lemonade costs $5.
# Each customer will only buy one lemonade and pay with either a $5, $10, or $20 b
# ill.  You must provide the correct change to each customer, so that the net tran
# saction is that the customer pays $5.
# Note that you don't have any change in hand at first.
# bills
# Example 3:
# Customers are standing in a queue to buy from you, and order one at a time (in t
# he order specified by bills).

# NOTE
# bills[i] will be either 5, 10, or 20.
# 0 <= bills.length <= 10000

# EXAMPLE
# Input: [5,5,10]
# Output: true
# Input: [10,10]
# Output: false
# Input: [5,5,5,10,20]
# Output: true
# Explanation:
# From the first 3 customers, we c
# ollect three $5 bills in order.
# From the fourth customer, we collect a $10 bill
# and give back a $5.
# From the fifth customer, we give a $10 bill and a $5 bill.
# S
# ince all customers got correct change, we output true.
# Input: [5,5,10,10,20]
# Output: false
# Explanation:
# From the first two customers i
# n order, we collect two $5 bills.
# For the next two customers in order, we collec
# t a $10 bill and give back a $5 bill.
# For the last customer, we can't give chang
# e of $15 back because we only have two $10 bills.
# Since not every customer recei
# ved correct change, the answer is false.

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def lemonadeChange(self, bills):
        """
        :type bills: List[int]
        :rtype: bool
```

```
        """
        coins = [20, 10, 5]
        counts = collections.defaultdict(int)
        for bill in bills:
            counts[bill] += 1
            change = bill - coins[-1]
            for coin in coins:
                if change == 0:
                    break
                if change >= coin:
                    count = min(counts[coin], change//coin)
                    counts[coin] -= count
                    change -= coin * count
            if change != 0:
                return False
        return True


class Solution2(object):
    def lemonadeChange(self, bills):
        """
        :type bills: List[int]
        :rtype: bool
        """
        five, ten = 0, 0
        for bill in bills:
            if bill == 5:
                five += 1
            elif bill == 10:
                if not five:
                    return False
                five -= 1
                ten += 1
            else:
                if ten and five:
                    ten -= 1
                    five -= 1
                elif five >= 3:
                    five -= 3
                else:
                    return False
        return True
```

longest-consecutive-sequence.py

```python
# DESC
# Your algorithm should run in O(n) complexity.
# Given an unsorted array of integers, find the length of the longest consecutive
# elements sequence.
# Example:

# NOTE
#

# EXAMPLE
# Input: [100, 4, 200, 1, 3, 2]
# Output: 4
# Explanation: The longest consecutive ele
# ments sequence is [1, 2, 3, 4]. Therefore its length is 4.

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

class Solution(object):
    # @param num, a list of integer
    # @return an integer
    def longestConsecutive(self, num):
        result, lengths = 1, {key: 0 for key in num}
        for i in num:
            if lengths[i] == 0:
                lengths[i] = 1
                left, right = lengths.get(i - 1, 0), lengths.get(i + 1, 0)
                length = 1 + left + right
                result, lengths[i - left], lengths[i + right] = max(result, length), length, length
        return result
```

find-n-unique-integers-sum-up-to-zero.py

```python
# find-n-unique-integers-sum-up-to-zero is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def sumZero(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        return [i for i in xrange(-(n//2), n//2+1) if not (i == 0 and n%2 == 0)]
```

non-overlapping-intervals.py

```
# DESC
# Note:
# Given a collection of intervals, find the minimum number of intervals you need t
# o remove to make the rest of the intervals non-overlapping.
# Example 1:
# Example 3:
# Example 2:

# NOTE
# Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap ea
# ch other.
# You may assume the interval's end point is always bigger than its start point.

# EXAMPLE
# Input: [[1,2],[1,2],[1,2]]
# Output: 2
# Explanation: You need to remove two [1,2] t
# o make the rest of intervals non-overlapping.
# Input: [[1,2],[2,3]]
# Output: 0
# Explanation: You don't need to remove any of the
# intervals since they're already non-overlapping.
# 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.

# 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
```

two-sum-ii-input-array-is-sorted.py

```python
# two-sum-ii-input-array-is-sorted is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def twoSum(self, nums, target):
        start, end = 0, len(nums) - 1

        while start != end:
            sum = nums[start] + nums[end]
            if sum > target:
                end -= 1
            elif sum < target:
                start += 1
            else:
                return [start + 1, end + 1]
```

find-all-good-strings.py

```python
# find-all-good-strings is not found.
# Time:  O(m * n)
# Space: O(m)

class Solution(object):
    def findGoodStrings(self, n, s1, s2, evil):
        """
        :type n: int
        :type s1: str
        :type s2: str
        :type evil: str
        :rtype: int
        """
        MOD = 10**9+7
        def getPrefix(pattern):
            prefix = [-1]*len(pattern)
            j = -1
            for i in xrange(1, len(pattern)):
                while j != -1 and pattern[j+1] != pattern[i]:
                    j = prefix[j]
                if pattern[j+1] == pattern[i]:
                    j += 1
                prefix[i] = j
            return prefix

        prefix = getPrefix(evil)
        dp = [[[[0]*len(evil) for _ in xrange(2)] for _ in xrange(2)] for _ in xrange(2)]
        dp[0][0][0][0] = 1
        for i in xrange(n):
            dp[(i+1)%2] = [[[0]*len(evil) for _ in xrange(2)] for _ in xrange(2)]
            for j in xrange(2):
                for k in xrange(2):
                    min_c = 'a' if j else s1[i]
                    max_c = 'z' if k else s2[i]
                    for l in xrange(len(evil)):
                        if not dp[i%2][j][k][l]:
                            continue
                        for c in xrange(ord(min_c)-ord('a'), ord(max_c)-ord('a')+1):
                            c = chr(c+ord('a'))
                            m = l-1
                            while m != -1 and evil[m+1] != c:
                                m = prefix[m]
                            if evil[m+1] == c:
                                m += 1
                            if m+1 == len(evil):
                                continue
                            dp[(i+1)%2][j or (s1[i] != c)][k or (s2[i] != c)][m+1] = \
                                (dp[(i+1)%2][j or (s1[i] != c)][k or (s2[i] != c)][m+1] + dp[i%2][j][k][l]) % 
        result = 0
        for j in xrange(2):
            for k in xrange(2):
                for l in xrange(len(evil)):
                    result = (result + dp[n%2][j][k][l]) % MOD
        return result
```

single-number.py

```
# DESC
# Example 2:
# Given a non-empty array of integers, every element appears twice except for one.
#  Find that single one.
# Your algorithm should have a linear runtime complexity. Could you implement it w
# ithout using extra memory?
# Example 1:
# Note:

# NOTE
#

# EXAMPLE
# Input: [2,2,1]
# Output: 1
# Input: [4,1,2,1,2]
# Output: 4

# Time:  O(n)
# Space: O(1)

import operator
from functools import reduce


class Solution(object):
    """
    :type nums: List[int]
    :rtype: int
    """
    def singleNumber(self, A):
        return reduce(operator.xor, A)
```

max-dot-product-of-two-subsequences.py

```python
# max-dot-product-of-two-subsequences is not found.
# Time:  O(m * n)
# Space: O(min(m, n))

class Solution(object):
    def maxDotProduct(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: int
        """
        if len(nums1) < len(nums2):
            return self.maxDotProduct(nums2, nums1)
        dp = [[0]*len(nums2) for i in xrange(2)]
        for i in xrange(len(nums1)):
            for j in xrange(len(nums2)):
                dp[i%2][j] = nums1[i]*nums2[j]
                if i and j:
                    dp[i%2][j] += max(dp[(i-1)%2][j-1], 0)
                if i:
                    dp[i%2][j] = max(dp[i%2][j], dp[(i-1)%2][j])
                if j:
                    dp[i%2][j] = max(dp[i%2][j], dp[i%2][j-1])
        return dp[(len(nums1)-1)%2][-1]
```

reverse-substrings-between-each-pair-of-parentheses.py

```python
# reverse-substrings-between-each-pair-of-parentheses is not found.
# 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()
                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

```
# DESC
# Example 2:
# Example 4:
# 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.
# Example 3:
# Example 5:
# Example 1:
# 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", "192.1
# 68.1.312" and "192.168@1.1" are invalid IP addresses.
# Constraints:

# NOTE
# s consists of digits only.
# 0 <= s.length <= 3000

# EXAMPLE
# Input: s = "101023"
# Output: ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","10
# 1.0.2.3"]
# Input: s = "25525511135"
# Output: ["255.255.11.135","255.255.111.35"]
# Input: s = "1111"
# Output: ["1.1.1.1"]
# Input: s = "0000"
# Output: ["0.0.0.0"]
# Input: s = "010010"
# Output: ["0.10.0.10","0.100.1.0"]

# 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
```

contain-virus.py

```
# DESC
# The world is modeled as a 2-D array of cells, where 0 represents uninfected cell
# s, and 1 represents cells contaminated with the virus.  A wall (and only one wal
# l) can be installed between any two 4-directionally adjacent cells, on the share
# d boundary.
# Example 3:
# Note:
# A virus is spreading rapidly, and your task is to quarantine the infected area b
# y installing walls.
# Every night, the virus spreads to all neighboring cells in all four directions u
# nless blocked by a wall.
# Resources are limited. Each day, you can install walls
# around only one region -- the affected area (continuous block of infected cells)
#  that threatens the most uninfected cells the following night. There will never
# be a tie.
# Can you save the day? If so, what is the number of walls required? If not, and t
# he world becomes fully infected, return the number of walls used.
# Example 2:
# Example 1:

# NOTE
# Each grid[i][j] will be either 0 or 1.
# The number of rows and columns of grid will each be in the range [1, 50].
# Throughout the described process, there is always a contiguous viral region that
#  will infect strictly more uncontaminated squares in the next round.

# EXAMPLE
# Input: grid =
# [[1,1,1,0,0,0,0,0,0],
#  [1,0,1,0,1,1,1,1,1],
#  [1,1,1,0,0,0,0,0,0]]
#
# Output: 13
# Explanation: The region on the left only builds two new walls.
# Input: grid =
# [[1,1,1],
#  [1,0,1],
#  [1,1,1]]
# Output: 4
# Explanation: Even though
# there is only one cell saved, there are 4 walls built.
# Notice that walls are onl
# y built on the shared boundary of two different cells.
# Input: grid =
# [[0,1,0,0,0,0,0,1],
#  [0,1,0,0,0,0,0,1],
#  [0,0,0,0,0,0,0,1],
#  [0,0
# ,0,0,0,0,0,0]]
# Output: 10
# Explanation:
# There are 2 contaminated regions.
# On the
# first day, add 5 walls to quarantine the viral region on the left. The board aft
# er the virus spreads is:
#
# [[0,1,0,0,0,0,1,1],
#  [0,1,0,0,0,0,1,1],
```

```
#  [0,0,0,0,0,0,
# 1,1],
#  [0,0,0,0,0,0,0,1]]
#
# On the second day, add 5 walls to quarantine the vira
# l region on the right. The virus is fully contained.

# Time:  O((m * n)^(4/3)), days = O((m * n)^(1/3))
# Space: O(m * n)

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

        def dfs(grid, r, c, lookup, regions, frontiers, perimeters):
            if (r, c) in lookup:
                return
            lookup.add((r, c))
            regions[-1].add((r, c))
            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:
                    dfs(grid, nr, nc, lookup, regions, frontiers, perimeters)
                elif grid[nr][nc] == 0:
                    frontiers[-1].add((nr, nc))
                    perimeters[-1] += 1

        result = 0
        while True:
            lookup, regions, frontiers, perimeters = set(), [], [], []
            for r, row in enumerate(grid):
                for c, val in enumerate(row):
                    if val == 1 and (r, c) not in lookup:
                        regions.append(set())
                        frontiers.append(set())
                        perimeters.append(0)
                        dfs(grid, r, c, lookup, regions, frontiers, perimeters)

            if not regions: break

            triage_idx = frontiers.index(max(frontiers, key = len))
            for i, region in enumerate(regions):
                if i == triage_idx:
                    result += perimeters[i]
                    for r, c in region:
                        grid[r][c] = -1
                    continue
                for r, c in region:
                    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] == 0:
                    grid[nr][nc] = 1

    return result
```

find-positive-integer-solution-for-a-given-equation.py

```
# DESC
# Given a function  f(x, y) and a value z, return all positive integer pairs x and
#  y where f(x,y) == z.
# For custom testing purposes you're given an integer function_id and a target z a
# s input, where function_id represent one function from an secret internal list,
# on the examples you'll know only two functions from the list.
# The function interface is defined like this:
# You may return the solutions in any order.
# Constraints:
# Example 2:
# The function is constantly increasing, i.e.:
# Example 1:

# NOTE
# 1 <= function_id <= 9
# It's also guaranteed that f(x, y) will fit in 32 bit signed integer if 1 <= x, y
#  <= 1000
# 1 <= z <= 100
# f(x, y) < f(x, y + 1)
# f(x, y) < f(x + 1, y)
# It's guaranteed that the solutions of f(x, y) == z will be on the range 1 <= x,
# y <= 1000

# EXAMPLE
# Input: function_id = 1, z = 5
# Output: [[1,4],[2,3],[3,2],[4,1]]
# Explanation: fun
# ction_id = 1 means that f(x, y) = x + y
# Input: function_id = 2, z = 5
# Output: [[1,5],[5,1]]
# Explanation: function_id = 2
#  means that f(x, y) = x * y
# interface CustomFunction {
# public:
#   // Returns positive integer f(x, y) for any
#  given positive integer x and y.
#   int f(int x, int y);
# };

# Time:  O(n)
# Space: O(1)

"""
   This is the custom function interface.
   You should not implement it, or speculate about its implementation
   class CustomFunction:
       # Returns f(x, y) for any given positive integers x and y.
       # Note that f(x, y) is increasing with respect to both x and y.
       # i.e. f(x, y) < f(x + 1, y), f(x, y) < f(x, y + 1)
       def f(self, x, y):

"""
class Solution(object):
    def findSolution(self, customfunction, z):
        """
        :type num: int
        :type z: int
        :rtype: List[List[int]]
```

```
    """
    result = []
    x, y = 1, 1
    while customfunction.f(x, y) < z:
        y += 1
    while y > 0:
        while y > 0 and customfunction.f(x, y) > z:
            y -= 1
        if y > 0 and customfunction.f(x, y) == z:
            result.append([x, y])
        x += 1
    return result
```

random-point-in-non-overlapping-rectangles.py

```
# DESC
# Solution
# Example 1:
# The input is two lists: the subroutines called and their arguments. Solution's c
# onstructor has one argument, the array of rectangles rects. pick has no argument
# s. Arguments are always wrapped with a list, even if there aren't any.
# Example 2:
# Note:
# Explanation of Input Syntax:
# Given a list of non-overlapping axis-aligned rectangles rects, write a function
# pick which randomly and uniformily picks an integer point in the space covered b
# y the rectangles.

# NOTE
# 1 <= rects.length <= 100
# ith rectangle = rects[i] = [x1,y1,x2,y2], where [x1, y1] are the integer coordin
# ates of the bottom-left corner, and [x2, y2] are the integer coordinates of the
# top-right corner.
# pick return a point as an array of integer coordinates [p_x, p_y]
# length and width of each rectangle does not exceed 2000.
# An integer point is a point that has integer coordinates.
# pick is called at most 10000 times.
# A point on the perimeter of a rectangle is included in the space covered by the
# rectangles.

# EXAMPLE
# 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]]
# Input:
# ["Solution","pick","pick","pick"]
# [[[[1,1,5,5]]],[],[],[]]
# Output:
# [nul
# l,[4,1],[4,1],[3,3]]

# 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]
```

intersection-of-three-sorted-arrays.py

```
# intersection-of-three-sorted-arrays is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def arraysIntersection(self, arr1, arr2, arr3):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :type arr3: List[int]
        :rtype: List[int]
        """
        result = []
        i, j, k = 0, 0, 0
        while i != len(arr1) and j != len(arr2) and k != len(arr3):
            if arr1[i] == arr2[j] == arr3[k]:
                result.append(arr1[i])
                i += 1
                j += 1
                k += 1
            else:
                curr = max(arr1[i], arr2[j], arr3[k])
                while i != len(arr1) and arr1[i] < curr:
                    i += 1
                while j != len(arr2) and arr2[j] < curr:
                    j += 1
                while k != len(arr3) and arr3[k] < curr:
                    k += 1
        return result


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def arraysIntersection(self, arr1, arr2, arr3):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :type arr3: List[int]
        :rtype: List[int]
        """
        intersect = reduce(set.intersection, map(set, [arr2, arr3]))
        return [x for x in arr1 if x in intersect]
```

convert-a-number-to-hexadecimal.py

```
# DESC
# Note:
# Example 2:
# Example 1:
# Given an integer, write an algorithm to convert it to hexadecimal. For negative
# integer, two's complement method is used.

# NOTE
# The hexadecimal string must not contain extra leading 0s. If the number is zero,
#  it is represented by a single zero character '0'; otherwise, the first characte
# r in the hexadecimal string will not be the zero character.
# The given number is guaranteed to fit within the range of a 32-bit signed integer.
# All letters in hexadecimal (a-f) must be in lowercase.
# You must not use any method provided by the library which converts/formats the n
# umber to hex directly.

# EXAMPLE
# Input:
# -1
#
# Output:
# "ffffffff"
# Input:
# 26
#
# Output:
# "1a"

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def toHex(self, num):
        """
        :type num: int
        :rtype: str
        """
        if not num:
            return "0"

        result = []
        while num and len(result) != 8:
            h = num & 15
            if h < 10:
                result.append(str(chr(ord('0') + h)))
            else:
                result.append(str(chr(ord('a') + h-10)))
            num >>= 4
        result.reverse()

        return "".join(result)
```

partition-labels.py

```python
# DESC
# Note:
# Example 1:
# A string S of lowercase English letters is given. We want to partition this stri
# ng into as many parts as possible so that each letter appears in at most one par
# t, and return a list of integers representing the size of these parts.

# NOTE
# S will have length in range [1, 500].
# S will consist of lowercase English letters ('a' to 'z') only.

# EXAMPLE
# Input: S = "ababcbacadefegdehijhklij"
# Output: [9,7,8]
# Explanation:
# The partition
#  is "ababcbaca", "defegde", "hijhklij".
# This is a partition so that each letter
# appears in at most one part.
# A partition like "ababcbacadefegde", "hijhklij" is
# incorrect, because it splits S into less parts.

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

class Solution(object):
    def partitionLabels(self, S):
        """
        :type S: str
        :rtype: List[int]
        """
        lookup = {c: i for i, c in enumerate(S)}
        first, last = 0, 0
        result = []
        for i, c in enumerate(S):
            last = max(last, lookup[c])
            if i == last:
                result.append(i-first+1)
                first = i+1
        return result
```

super-ugly-number.py

```
# DESC
# primes
# Example:
# Super ugly numbers are positive numbers whose all prime factors are in the given
#  prime list primes of size k.
# Write a program to find the nth super ugly number.
# Note:

# NOTE
# 0 < k  100, 0 < n  106, 0 < primes[i] < 1000.
# The given numbers in primes are in ascending order.
# The nth super ugly number is guaranteed to fit in a 32-bit signed integer.
# 1 is a super ugly number for any given primes.

# 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 g
# iven primes = [2,7,13,19] of size 4.

# 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
        """
        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]:
                    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
```

stamping-the-sequence.py

```
# DESC
# You want to form a target string of lowercase letters.
# Note:
# target.length
# For example, if the initial sequence is "?????", and your stamp is "abc",  then
# you may make "abc??", "?abc?", "??abc" in the first turn.  (Note that the stamp
# must be fully contained in the boundaries of the sequence in order to stamp.)
# Example 1:
# Example 2:
# Also, if the sequence is possible to stamp, it is guaranteed it is possible to s
# tamp within 10 * target.length moves.  Any answers specifying more than this num
# ber of moves will not be accepted.
# If the sequence is possible to stamp, then return an array of the index of the l
# eft-most letter being stamped at each turn.  If the sequence is not possible to
# stamp, return an empty array.
# For example, if the sequence is "ababc", and the stamp is "abc", then we could r
# eturn the answer [0, 2], corresponding to the moves "?????" -> "abc??" -> "ababc
# ".
# At the beginning, your sequence is target.length '?' marks.  You also have a sta
# mp of lowercase letters.
# On each turn, you may place the stamp over the sequence, and replace every lette
# r in the sequence with the corresponding letter from the stamp.  You can make up
#  to 10 * target.length turns.

# NOTE
# 1 <= stamp.length <= target.length <= 1000
# stamp and target only contain lowercase letters.

# EXAMPLE
# Input: stamp = "abc", target = "ababc"
# Output: [0,2]
# ([1,0,2] would also be acce
# pted as an answer, as well as some other answers.)
# Input: stamp = "abca", target = "aabcaca"
# Output: [3,0,1]

# Time:  O((n - m) * m)
# Space: O((n - m) * m)

import collections


class Solution(object):
    def movesToStamp(self, stamp, target):
        M, N = len(stamp), len(target)

        q = collections.deque()
        lookup = [False]*N
        result = []
        A = []
        for i in xrange(N-M+1):
            made, todo = set(), set()
            for j, c in enumerate(stamp):
                if c == target[i+j]:
                    made.add(i+j)
                else:
                    todo.add(i+j)
            A.append((made, todo))
```

```
        if todo:
            continue
        result.append(i)
        for m in made:
            if lookup[m]:
                continue
            q.append(m)
            lookup[m] = True

    while q:
        i = q.popleft()
        for j in xrange(max(0, i-M+1), min(N-M, i)+1):
            made, todo = A[j]
            if i not in todo:
                continue
            todo.discard(i)
            if todo:
                continue
            result.append(j)
            for m in made:
                if lookup[m]:
                    continue
                q.append(m)
                lookup[m] = True
    return result[::-1] if all(lookup) else []
```

minimum-increment-to-make-array-unique.py

```
# DESC
# Example 1:
# Given an array of integers A, a move consists of choosing any A[i], and incremen
# ting it by 1.
# Return the least number of moves to make every value in A unique.
# Note:
# Example 2:

# NOTE
# 0 <= A.length <= 40000
# 0 <= A[i] < 40000

# EXAMPLE
# Input: [1,2,2]
# Output: 1
# Explanation:  After 1 move, the array could be [1, 2, 3].
# 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 fo
# r the array to have all unique values.

# 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
```

first-unique-character-in-a-string.py

```python
# DESC
# Note: You may assume the string contains only lowercase English letters.
# Examples:
# Given a string, find the first non-repeating character in it and return its inde
# x. If it doesn't exist, return -1.

# NOTE
#

# EXAMPLE
# s = "leetcode"
# return 0.
#
# s = "loveleetcode"
# return 2.

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

from collections import defaultdict

class Solution(object):
    def firstUniqChar(self, s):
        """
        :type s: str
        :rtype: int
        """
        lookup = defaultdict(int)
        candidtates = set()
        for i, c in enumerate(s):
            if lookup[c]:
                candidtates.discard(lookup[c])
            else:
                lookup[c] = i+1
                candidtates.add(i+1)

        return min(candidtates)-1 if candidtates else -1
```

find-right-interval.py

```
# DESC
# NOTE: input types have been changed on April 15, 2019. Please reset to default c
# ode definition to get new method signature.
# For any interval i, you need to store the minimum interval j's index, which mean
# s that the interval j has the minimum start point to build the "right" relations
# hip 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.
# Given a set of intervals, for each of the interval i, check if there exists an i
# nterval j whose start point is bigger than or equal to the end point of the inte
# rval i, which can be called that j is on the "right" of i.
# Example 2:
# Note:
# Example 3:
# Example 1:

# 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
# Input: [ [1,2] ]
#
# Output: [-1]
#
# Explanation: There is only one interval in the c
# ollection, so it outputs -1.
# Input: [ [3,4], [2,3], [1,2] ]
#
# Output: [-1, 0, 1]
#
# Explanation: There is no sat
# isfied "right" interval for [3,4].
# For [2,3], the interval [3,4] has minimum-"ri
# ght" start point;
# For [1,2], the interval [2,3] has minimum-"right" start point.
# Input: [ [1,4], [2,3], [3,4] ]
#
# Output: [-1, 2, -1]
#
# Explanation: There is no sa
# tisfied "right" interval for [1,4] and [3,4].
# For [2,3], the interval [3,4] has
# minimum-"right" start point.

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

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
```

maximum-average-subarray-i.py

```python
# DESC
# Given an array consisting of n integers, find the contiguous subarray of given l
# ength k that has the maximum average value. And you need to output the maximum a
# verage value.
# Note:
# Example 1:

# NOTE
# Elements of the given array will be in the range [-10,000, 10,000].
# 1 <= k <= n <= 30,000.

# EXAMPLE
# Input: [1,12,-5,-6,50,3], k = 4
# Output: 12.75
# Explanation: Maximum average is (1
# 2-5-6+50)/4 = 51/4 = 12.75

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findMaxAverage(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: float
        """
        result = total = sum(nums[:k])
        for i in xrange(k, len(nums)):
            total += nums[i] - nums[i-k]
            result = max(result, total)
        return float(result) / k
```

longest-happy-string.py

```python
# longest-happy-string is not found.
# 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'))
        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)
```

range-module.py

```
# DESC
# Example 1:
# Note:
# A Range Module is a module that tracks ranges of numbers. Your task is to design
#  and implement the following interfaces in an efficient manner.

# NOTE
# The total number of calls to removeRange in a single test case is at most 1000.
# A half open interval [left, right) denotes all real numbers left <= x < right.
# addRange(int left, int right) Adds the half-open interval [left, right), trackin
# g every real number in that interval.  Adding an interval that partially overlap
# s with currently tracked numbers should add any numbers in the interval [left, r
# ight) that are not already tracked.
# The total number of calls to queryRange in a single test case is at most 5000.
# queryRange(int left, int right) Returns true if and only if every real number in
#  the interval [left, right)
#  is currently being tracked.
# 0 < left < right < 10^9 in all calls to addRange, queryRange, removeRange.
# removeRange(int left, int right) Stops tracking every real number currently bein
# g tracked in the interval [left, right).
# The total number of calls to addRange in a single test case is at most 1000.

# EXAMPLE
# addRange(10, 20): null
# removeRange(14, 16): null
# queryRange(10, 14): true (Every
#  number in [10, 14) is being tracked)
# queryRange(13, 15): false (Numbers like 14
# , 14.03, 14.17 in [13, 15) are not being tracked)
# queryRange(16, 17): true (The
# number 16 in [16, 17) is still being tracked, despite the remove operation)

# Time:  addRange:    O(n)
#        removeRange: O(n)
#        queryRange:  O(logn)
# Space: O(n)

import bisect


class RangeModule(object):

    def __init__(self):
        self.__intervals = []

    def addRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: void
        """
        tmp = []
        i = 0
        for interval in self.__intervals:
            if right < interval[0]:
                tmp.append((left, right))
                break
            elif interval[1] < left:
```

```
                tmp.append(interval)
            else:
                left = min(left, interval[0])
                right = max(right, interval[1])
            i += 1
        if i == len(self.__intervals):
            tmp.append((left, right))
        while i < len(self.__intervals):
            tmp.append(self.__intervals[i])
            i += 1
        self.__intervals = tmp

    def queryRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: bool
        """
        i = bisect.bisect_left(self.__intervals, (left, float("inf")))
        if i: i -= 1
        return bool(self.__intervals) and \
               self.__intervals[i][0] <= left and \
               right <= self.__intervals[i][1]

    def removeRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: void
        """
        tmp = []
        for interval in self.__intervals:
            if interval[1] <= left or interval[0] >= right:
                tmp.append(interval)
            else:
                if interval[0] < left:
                    tmp.append((interval[0], left))
                if right < interval[1]:
                    tmp.append((right, interval[1]))
        self.__intervals = tmp
```

shortest-common-supersequence.py

```
# DESC
# (A string S is a subsequence of string T if deleting some number of characters f
# rom T (possibly 0, and the characters are chosen anywhere from T) results in the
#  string S.)
# Example 1:
# Given two strings str1 and str2, return the shortest string that has both str1 a
# nd str2 as subsequences.  If multiple answers exist, you may return any of them.
# Note:

# NOTE
# 1 <= str1.length, str2.length <= 1000
# str1 and str2 consist of lowercase English letters.

# EXAMPLE
# Input: str1 = "abac", str2 = "cab"
# Output: "cabac"
# Explanation:
# str1 = "abac" i
# s a subsequence of "cabac" because we can delete the first "c".
# str2 = "cab" is
# a subsequence of "cabac" because we can delete the last "ac".
# The answer provide
# d is the shortest such string that satisfies these properties.

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

class Solution(object):
    def shortestCommonSupersequence(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """
        dp = [[0 for _ in xrange(len(str2)+1)] for _ in xrange(2)]
        bt = [[None for _ in xrange(len(str2)+1)] for _ in xrange(len(str1)+1)]
        for i, c in enumerate(str1):
            bt[i+1][0] = (i, 0, c)
        for j, c in enumerate(str2):
            bt[0][j+1] = (0, j, c)
        for i in xrange(len(str1)):
            for j in xrange(len(str2)):
                if dp[i % 2][j+1] > dp[(i+1) % 2][j]:
                    dp[(i+1) % 2][j+1] = dp[i % 2][j+1]
                    bt[i+1][j+1] = (i, j+1, str1[i])
                else:
                    dp[(i+1) % 2][j+1] = dp[(i+1) % 2][j]
                    bt[i+1][j+1] = (i+1, j, str2[j])
                if str1[i] != str2[j]:
                    continue
                if dp[i % 2][j]+1 > dp[(i+1) % 2][j+1]:
                    dp[(i+1) % 2][j+1] = dp[i % 2][j]+1
                    bt[i+1][j+1] = (i, j, str1[i])

        i, j = len(str1), len(str2)
        result = []
        while i != 0 or j != 0:
            i, j, c = bt[i][j]
```

```
    result.append(c)
result.reverse()
return "".join(result)
```

maximum-length-of-repeated-subarray.py

```
# maximum-length-of-repeated-subarra is not found.
# Time:  O(m * n)
# Space: O(min(m, n))

import collections


class Solution(object):
    def findLength(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        if len(A) < len(B): return self.findLength(B, A)
        result = 0
        dp = [[0] * (len(B)+1) for _ in xrange(2)]
        for i in xrange(len(A)):
            for j in xrange(len(B)):
                if A[i] == B[j]:
                    dp[(i+1)%2][j+1] = dp[i%2][j]+1
                else:
                    dp[(i+1)%2][j+1] = 0
            result = max(result, max(dp[(i+1)%2]))
        return result


# Time:  O(m * n * log(min(m, n)))
# Space: O(min(m, n))
# Binary search + rolling hash solution (226 ms)
class Solution2(object):
    def findLength(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        if len(A) > len(B): return self.findLength(B, A)
        M, p = 10**9+7, 113
        p_inv = pow(p, M-2, M)
        def check(guess):
            def rolling_hashes(source, length):
                if length == 0:
                    yield 0, 0
                    return

                val, power = 0, 1
                for i, x in enumerate(source):
                    val = (val + x*power) % M
                    if i < length - 1:
                        power = (power*p) % M
                    else:
                        yield val, i-(length-1)
                        val = (val-source[i-(length-1)])*p_inv % M

            hashes = collections.defaultdict(list)
            for hash_val, i in rolling_hashes(A, guess):
                hashes[hash_val].append(i)
```

```
            for hash_val, j in rolling_hashes(B, guess):
                if any(A[i:i+guess] == B[j:j+guess] for i in hashes[hash_val]):
                    return True
            return False

        left, right = 0, min(len(A), len(B)) + 1
        while left < right:
            mid = left + (right-left)/2
            if not check(mid):  # find the min idx such that check(idx) == false
                right = mid
            else:
                left = mid+1
        return left-1


# Time:  O(m * n * min(m, n) * log(min(m, n)))
# Space: O(min(m^2, n^2))
# Binary search (122 ms)
class Solution3(object):
    def findLength(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        if len(A) > len(B): return self.findLength(B, A)

        def check(length):
            lookup = set(A[i:i+length] \
                         for i in xrange(len(A)-length+1))
            return any(B[j:j+length] in lookup \
                       for j in xrange(len(B)-length+1))

        A = ''.join(map(chr, A))
        B = ''.join(map(chr, B))
        left, right = 0, min(len(A), len(B)) + 1
        while left < right:
            mid = left + (right-left)/2
            if not check(mid):  # find the min idx such that check(idx) == false
                right = mid
            else:
                left = mid+1
        return left-1
```

best-meeting-point.py

```
# best-meeting-point is not found.
# Time:  O(m * n)
# Space: O(m + n)

from random import randint


class Solution(object):
    def minTotalDistance(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        x = [i for i, row in enumerate(grid) for v in row if v == 1]
        y = [j for row in grid for j, v in enumerate(row) if v == 1]
        mid_x = self.findKthLargest(x, len(x) / 2 + 1)
        mid_y = self.findKthLargest(y, len(y) / 2 + 1)

        return sum([abs(mid_x-i) + abs(mid_y-j)
                    for i, row in enumerate(grid)
                    for j, v in enumerate(row) if v == 1])

    def findKthLargest(self, nums, k):
        left, right = 0, len(nums) - 1
        while left <= right:
            pivot_idx = randint(left, right)
            new_pivot_idx = self.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(self, 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
```

generate-parentheses.py

```python
# DESC
# Given n pairs of parentheses, write a function to generate all combinations of w
# ell-formed parentheses.
# For example, given n = 3, a solution set is:

# NOTE
#

# EXAMPLE
# [
#   "((()))",
#   "(()())",
#   "(())()",
#   "()(())",
#   "()()()"
# ]

# 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

```
# DESC
# Find out how many ways to assign symbols to make sum of integers equal to target S.
# 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.
# Example 1:
# Constraints:

# NOTE
# 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.

# EXAMPLE
# 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 t
# o assign symbols to make the sum of nums be target 3.

# 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)
```

roman-to-integer.py

```
# DESC
# Roman numerals are usually written largest to smallest from left to right. Howev
# er, 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 princip
# le applies to the number nine, which is written as IX. There are six instances w
# here subtraction is used:
# Example 3:
# Example 1:
# Example 5:
# Example 2:
# Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.
# Given a roman numeral, convert it to an integer. Input is guaranteed to be withi
# n the range from 1 to 3999.
# For example, two is written as II in Roman numeral, just two one's added togethe
# r. Twelve is written as, XII, which is simply X + II. The number twenty seven is
#  written as XXVII, which is XX + V + II.
# Example 4:

# NOTE
# 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.
# I can be placed before V (5) and X (10) to make 4 and 9.

# EXAMPLE
# Symbol       Value
# I             1
# V             5
# X             10
# L
#   50
# C             100
# D             500
# M             1000
# Input: "MCMXCIV"
# Output: 1994
# Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
# Input: "IV"
# Output: 4
# Input: "IX"
# Output: 9
# Input: "LVIII"
# Output: 58
# Explanation: L = 50, V= 5, III = 3.
# Input: "III"
# Output: 3

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @return an integer
    def romanToInt(self, s):
        numeral_map = {"I": 1, "V": 5, "X": 10, "L": 50, "C":100, "D": 500, "M": 1000}
        decimal = 0
        for i in xrange(len(s)):
            if i > 0 and numeral_map[s[i]] > numeral_map[s[i - 1]]:
                decimal += numeral_map[s[i]] - 2 * numeral_map[s[i - 1]]
            else:
```

```
        decimal += numeral_map[s[i]]
    return decimal
```

k-diff-pairs-in-an-array.py

```python
# k-diff-pairs-in-an-arra is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def findPairs(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        if k < 0: return 0
        result, lookup = set(), set()
        for num in nums:
            if num-k in lookup:
                result.add(num-k)
            if num+k in lookup:
                result.add(num)
            lookup.add(num)
        return len(result)
```

tallest-billboard.py

```
# DESC
# Return the largest possible height of your billboard installation.  If you canno
# t support the billboard, return 0.
# You are installing a billboard and want it to have the largest height.  The bill
# board will have two steel supports, one on each side.  Each steel support must b
# e an equal height.
# Example 2:
# Example 3:
# You have a collection of rods which can be welded together.  For example, if you
#  have rods of lengths 1, 2, and 3, you can weld them together to make a support
# of length 6.
# Note:
# Example 1:

# NOTE
# 0 <= rods.length <= 20
# The sum of rods is at most 5000.
# 1 <= rods[i] <= 1000

# EXAMPLE
# Input: [1,2,3,4,5,6]
# Output: 10
# Explanation: We have two disjoint subsets {2,3,5
# } and {4,6}, which have the same sum = 10.
# Input: [1,2]
# Output: 0
# Explanation: The billboard cannot be supported, so we ret
# urn 0.
# Input: [1,2,3,6]
# Output: 6
# Explanation: We have two disjoint subsets {1,2,3} and
#  {6}, which have the same sum = 6.

# Time:  O(n * 3^(n/2))
# Space: O(3^(n/2))

import collections


class Solution(object):
    def tallestBillboard(self, rods):
        """
        :type rods: List[int]
        :rtype: int
        """
        def dp(A):
            lookup = collections.defaultdict(int)
            lookup[0] = 0
            for x in A:
                for d, y in lookup.items():
                    lookup[d+x] = max(lookup[d+x], y)
                    lookup[abs(d-x)] = max(lookup[abs(d-x)], y + min(d, x))
            return lookup

        left, right = dp(rods[:len(rods)//2]), dp(rods[len(rods)//2:])
        return max(left[d]+right[d]+d for d in left if d in right)
```

valid-word-square.py

```
# valid-word-square is not found.
# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def validWordSquare(self, words):
        """
        :type words: List[str]
        :rtype: bool
        """
        for i in xrange(len(words)):
            for j in xrange(len(words[i])):
                if j >= len(words) or i >= len(words[j]) or \
                   words[j][i] != words[i][j]:
                   return False
        return True
```

number-of-submatrices-that-sum-to-target.py

```python
# DESC
# x1, y1, x2, y2
# A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y] with x1 <= x <=
# x2 and y1 <= y <= y2.
# Example 2:
# Example 1:
# Given a matrix, and a target, return the number of non-empty submatrices that su
# m to target.
# Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2') are different if they
# have some coordinate that is different: for example, if x1 != x1'.
# Note:

# NOTE
# 1 <= matrix.length <= 300
# 1 <= matrix[0].length <= 300
# -10^8 <= target <= 10^8
# -1000 <= matrix[i] <= 1000

# EXAMPLE
# Input: matrix = [[1,-1],[-1,1]], target = 0
# Output: 5
# Explanation: The two 1x2 s
# ubmatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.
# Input: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
# Output: 4
# Explanation: The
#  four 1x1 submatrices that only contain 0.

# Time:  O(m^2*n), m is min(r, c), n is max(r, c)
# Space: O(n), which doesn't include transposed space

import collections


class Solution(object):
    def numSubmatrixSumTarget(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: int
        """
        if len(matrix) > len(matrix[0]):
            return self.numSubmatrixSumTarget(map(list, zip(*matrix)), target)

        for i in xrange(len(matrix)):
            for j in xrange(len(matrix[i])-1):
                matrix[i][j+1] += matrix[i][j]

        result = 0
        for i in xrange(len(matrix)):
            prefix_sum = [0]*len(matrix[i])
            for j in xrange(i, len(matrix)):
                lookup = collections.defaultdict(int)
                lookup[0] = 1
                for k in xrange(len(matrix[j])):
                    prefix_sum[k] += matrix[j][k]
                    if prefix_sum[k]-target in lookup:
                        result += lookup[prefix_sum[k]-target]
```

```
            lookup[prefix_sum[k]] += 1
    return result
```

to-lower-case.py

```
# DESC
# Implement function ToLowerCase() that has a string parameter str, and returns th
# e same string in lowercase.
# Example 1:
# Example 3:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: "LOVELY"
# Output: "lovely"
# Input: "here"
# Output: "here"
# Input: "Hello"
# Output: "hello"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def toLowerCase(self, str):
        """
        :type str: str
        :rtype: str
        """
        return "".join([chr(ord('a')+ord(c)-ord('A'))
                        if 'A' <= c <= 'Z' else c for c in str])
```

permutation-in-string.py

```
# DESC
# Example 2:
# Constraints:
# Example 1:
# 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.

# NOTE
# The input strings only contain lower case letters.
# The length of both given strings is in range [1, 10,000].

# EXAMPLE
# Input:s1= "ab" s2 = "eidboaoo"
# Output: False
# Input: s1 = "ab" s2 = "eidbaooo"
# Output: True
# Explanation: s2 contains one permu
# tation of s1 ("ba").

# 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
```

regular-expression-matching.py

```
# DESC
# Example 2:
# Example 3:
# Example 1:
# Given an input string (s) and a pattern (p), implement regular expression matchi
# ng with support for '.' and '*'.
# Example 5:
# Note:
# The matching should cover the entire input string (not partial).
# Example 4:

# NOTE
# s could be empty and contains only lowercase letters a-z.
# p could be empty and contains only lowercase letters a-z, and characters like . or *.

# EXAMPLE
# Input:
# s = "mississippi"
# p = "mis*is*p*."
# Output: false
# Input:
# s = "aab"
# p = "c*a*b"
# Output: true
# Explanation: c can be repeated 0 times
# , a can be repeated 1 time. Therefore, it matches "aab".
# '.' Matches any single character.
# '*' Matches zero or more of the preceding element.
# Input:
# s = "ab"
# p = ".*"
# Output: true
# Explanation: ".*" means "zero or more (*)
# of any character (.)".
# Input:
# s = "aa"
# p = "a*"
# Output: true
# Explanation: '*' means zero or more of the
#  preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".
# Input:
# s = "aa"
# p = "a"
# Output: false
# Explanation: "a" does not match the entire
#  string "aa".

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

class Solution(object):
    # @return a boolean
    def isMatch(self, s, p):
        k = 3
        result = [[False for j in xrange(len(p) + 1)] for i in xrange(k)]

        result[0][0] = True
        for i in xrange(2, len(p) + 1):
```

```
            if p[i-1] == '*':
                result[0][i] = result[0][i-2]

        for i in xrange(1,len(s) + 1):
            if i > 1:
                result[0][0] = False
            for j in xrange(1, len(p) + 1):
                if p[j-1] != '*':
                    result[i % k][j] = result[(i-1) % k][j-1] and (s[i-1] == p[j-1] or p[j-1] == '.')
                else:
                    result[i % k][j] = result[i % k][j-2] or (result[(i-1) % k][j] and (s[i-1] == p[j-2] or p[

        return result[len(s) % k][len(p)]

# dp
# Time:  O(m * n)
# Space: O(m * n)
class Solution2(object):
    # @return a boolean
    def isMatch(self, s, p):
        result = [[False for j in xrange(len(p) + 1)] for i in xrange(len(s) + 1)]

        result[0][0] = True
        for i in xrange(2, len(p) + 1):
            if p[i-1] == '*':
                result[0][i] = result[0][i-2]

        for i in xrange(1,len(s) + 1):
            for j in xrange(1, len(p) + 1):
                if p[j-1] != '*':
                    result[i][j] = result[i-1][j-1] and (s[i-1] == p[j-1] or p[j-1] == '.')
                else:
                    result[i][j] = result[i][j-2] or (result[i-1][j] and (s[i-1] == p[j-2] or p[j-2] == '.'))

        return result[len(s)][len(p)]

# iteration
class Solution3(object):
    # @return a boolean
    def isMatch(self, s, p):
        p_ptr, s_ptr, last_s_ptr, last_p_ptr = 0, 0, -1, -1
        last_ptr = []
        while s_ptr < len(s):
            if p_ptr < len(p) and (p_ptr == len(p) - 1 or p[p_ptr + 1] != '*') and \
            (s_ptr < len(s) and (p[p_ptr] == s[s_ptr] or p[p_ptr] == '.')):
                    s_ptr += 1
                    p_ptr += 1
            elif p_ptr < len(p) - 1 and (p_ptr != len(p) - 1 and p[p_ptr + 1] == '*'):
                p_ptr += 2
                last_ptr.append([s_ptr, p_ptr])
            elif  last_ptr:
                [last_s_ptr, last_p_ptr] = last_ptr.pop()
                while last_ptr and p[last_p_ptr - 2] != s[last_s_ptr] and p[last_p_ptr - 2] != '.':
                    [last_s_ptr, last_p_ptr] = last_ptr.pop()

                if p[last_p_ptr - 2] == s[last_s_ptr] or p[last_p_ptr - 2] == '.':
                    last_s_ptr += 1
                    s_ptr = last_s_ptr
                    p_ptr = last_p_ptr
                    last_ptr.append([s_ptr, p_ptr])
```

```
                else:
                    return False
            else:
                return False

        while p_ptr < len(p) - 1 and p[p_ptr] == '.' and p[p_ptr + 1] == '*':
            p_ptr += 2

        return p_ptr == len(p)

# recursive
class Solution4(object):
    # @return a boolean
    def isMatch(self, s, p):
        if not p:
            return not s

        if len(p) == 1 or p[1] != '*':
            if len(s) > 0 and (p[0] == s[0] or p[0] == '.'):
                return self.isMatch(s[1:], p[1:])
            else:
                return False
        else:
            while len(s) > 0 and (p[0] == s[0] or p[0] == '.'):
                if self.isMatch(s, p[2:]):
                    return True
                s = s[1:]
            return self.isMatch(s, p[2:])
```

maximum-swap.py

```
# maximum-swa is not found.
# 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))
```

binary-tree-longest-consecutive-sequence-ii.py

```python
# binary-tree-longest-consecutive-sequence-ii is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def longestConsecutive(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def longestConsecutiveHelper(root):
            if not root:
                return 0, 0
            left_len = longestConsecutiveHelper(root.left)
            right_len = longestConsecutiveHelper(root.right)
            cur_inc_len, cur_dec_len = 1, 1
            if root.left:
                if root.left.val == root.val + 1:
                    cur_inc_len = max(cur_inc_len, left_len[0] + 1)
                elif root.left.val == root.val - 1:
                    cur_dec_len = max(cur_dec_len, left_len[1] + 1)
            if root.right:
                if root.right.val == root.val + 1:
                    cur_inc_len = max(cur_inc_len, right_len[0] + 1)
                elif root.right.val == root.val - 1:
                    cur_dec_len = max(cur_dec_len, right_len[1] + 1)
            self.max_len = max(self.max_len, cur_dec_len + cur_inc_len - 1)
            return cur_inc_len, cur_dec_len

        self.max_len = 0
        longestConsecutiveHelper(root)
        return self.max_len
```

unique-paths-iii.py

```
# DESC
# Return the number of 4-directional walks from the starting square to the ending
# square, that walk over every non-obstacle square exactly once.
# Example 3:
# Note:
# Example 1:
# Example 2:
# On a 2-dimensional grid, there are 4 types of squares:

# NOTE
# 0 represents empty squares we can walk over.
# 2 represents the ending square.  There is exactly one ending square.
# 1 <= grid.length * grid[0].length <= 20
# -1 represents obstacles that we cannot walk over.
# 1 represents the starting square.  There is exactly one starting square.

# EXAMPLE
# Input: [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]
# Output: 2
# Explanation: We have the follo
# wing two paths:
# 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),
# (2,2)
# 2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)
# Input: [[1,0,0,0],[0,0,0,0],[0,0,0,2]]
# Output: 4
# Explanation: We have the follow
# ing four paths:
# 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),
# (2,2),(2,3)
# 2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3)
# ,(2,3)
# 3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3
# )
# 4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)
# Input: [[0,1],[2,0]]
# Output: 0
# Explanation:
# There is no path that walks over ev
# ery empty square exactly once.
# Note that the starting and ending square can be a
# nywhere in the grid.

# Time:  O(m * n * 2^(m * n))
# Space: O(m * n * 2^(m * n))

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

        def index(grid, r, c):
            return 1 << (r*len(grid[0])+c)

        def dp(grid, src, dst, todo, lookup):
```

```
    if src == dst:
        return int(todo == 0)
    key = (src, todo)
    if key in lookup:
        return lookup[key]

    result = 0
    for d in directions:
        r, c = src[0]+d[0], src[1]+d[1]
        if 0 <= r < len(grid) and 0 <= c < len(grid[0]) and \
           grid[r][c] % 2 == 0 and \
           todo & index(grid, r, c):
            result += dp(grid, (r, c), dst, todo ^ index(grid, r, c), lookup)

    lookup[key] = result
    return lookup[key]

todo = 0
src, dst = None, None
for r, row in enumerate(grid):
    for c, val in enumerate(row):
        if val % 2 == 0:
            todo |= index(grid, r, c)
        if val == 1:
            src = (r, c)
        elif val == 2:
            dst = (r, c)
return dp(grid, src, dst, todo, {})
```

all-paths-from-source-lead-to-destination.py

```python
# all-paths-from-source-lead-to-destination is not found.
# Time:  O(n + e)
# Space: O(n + e)

import collections


class Solution(object):
    def leadsToDestination(self, n, edges, source, destination):
        """
        :type n: int
        :type edges: List[List[int]]
        :type source: int
        :type destination: int
        :rtype: bool
        """
        UNVISITED, VISITING, DONE = range(3)
        def dfs(children, node, destination, status):
            if status[node] == DONE:
                return True
            if status[node] == VISITING:
                return False
            status[node] = VISITING
            if node not in children and node != destination:
                return False
            if node in children:
                for child in children[node]:
                    if not dfs(children, child, destination, status):
                        return False
            status[node] = DONE
            return True

        children = collections.defaultdict(list)
        for parent, child in edges:
            children[parent].append(child)
        return dfs(children, source, destination, [0]*n)
```

minimum-flips-to-make-a-or-b-equal-to-c.py

```python
# minimum-flips-to-make-a-or-b-equal-to-c is not found.
# Time:  O(31)
# Space: O(1)

class Solution(object):
    def minFlips(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: int
        """
        def number_of_1_bits(n):
            result = 0
            while n:
                n &= n-1
                result += 1
            return result

        return number_of_1_bits((a|b)^c) + number_of_1_bits(a&b&~c)


# Time:  O(31)
# Space: O(1)
class Solution2(object):
    def minFlips(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: int
        """
        result = 0
        for i in xrange(31):
            a_i, b_i, c_i = map(lambda x: x&1, [a, b, c])
            if (a_i | b_i) != c_i:
                result += 2 if a_i == b_i == 1 else 1
            a, b, c = a >> 1, b >> 1, c >> 1
        return result
```

verify-preorder-sequence-in-binary-search-tree.py

```python
# verify-preorder-sequence-in-binary-search-tree is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {integer[]} preorder
    # @return {boolean}
    def verifyPreorder(self, preorder):
        low, i = float("-inf"), -1
        for p in preorder:
            if p < low:
                return False
            while i >= 0 and p > preorder[i]:
                low = preorder[i]
                i -= 1
            i += 1
            preorder[i] = p
        return True

# Time:  O(n)
# Space: O(h)
class Solution2(object):
    # @param {integer[]} preorder
    # @return {boolean}
    def verifyPreorder(self, preorder):
        low = float("-inf")
        path = []
        for p in preorder:
            if p < low:
                return False
            while path and p > path[-1]:
                low = path[-1]
                path.pop()
            path.append(p)
        return True
```

best-time-to-buy-and-sell-stock-with-cooldown.py

```
# DESC
# Say you have an array for which the ith element is the price of a given stock on
#  day i.
# Example:
# Design an algorithm to find the maximum profit. You may complete as many transac
# tions as you like (ie, buy one and sell one share of the stock multiple times) w
# ith the following restrictions:

# NOTE
# 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])
```

design-compressed-string-iterator.py

```python
# design-compressed-string-iterator is not found.
# Time:  O(1)
# Space: O(1)

import re


class StringIterator(object):

    def __init__(self, compressedString):
        """
        :type compressedString: str
        """
        self.__result = re.findall(r"([a-zA-Z])(\d+)", compressedString)
        self.__index, self.__num, self.__ch = 0, 0, ' '

    def next(self):
        """
        :rtype: str
        """
        if not self.hasNext():
            return ' '
        if self.__num == 0:
            self.__ch = self.__result[self.__index][0]
            self.__num = int(self.__result[self.__index][1])
            self.__index += 1
        self.__num -= 1
        return self.__ch


    def hasNext(self):
        """
        :rtype: bool
        """
        return self.__index != len(self.__result) or self.__num != 0
```

largest-plus-sign.py

```
# DESC
# Example 3:
# Note:
# grid[x][y] = 1
# Examples of Axis-Aligned Plus Signs of Order k:
# Example 2:
# Example 1:
# In a 2D grid from (0, 0) to (N-1, N-1), every cell contains a 1, except those ce
# lls in the given list mines which are 0.  What is the largest axis-aligned plus
# sign of 1s contained in the grid?  Return the order of the plus sign.  If there
# is none, return 0.
# An "axis-aligned plus sign of 1s of order k" has some center grid[x][y] = 1 alon
# g with 4 arms of length k-1 going up, down, left, and right, and made of 1s.  Th
# is is demonstrated in the diagrams below.  Note that there could be 0s or 1s bey
# ond the arms of the plus sign, only the relevant area of the plus sign is checke
# d for 1s.

# NOTE
# mines will have length at most 5000.
# mines[i] will be length 2 and consist of integers in the range [0, N-1].
# (Additionally, programs submitted in C, C++, or C# will be judged with a slightl
# y smaller time limit.)
# N will be an integer in the range [1, 500].

# EXAMPLE
# Input: N = 1, mines = [[0, 0]]
# Output: 0
# Explanation:
# There is no plus sign, so
# return 0.
# Input: N = 2, mines = []
# Output: 1
# Explanation:
# There is no plus sign of order 2
# , but there is of order 1.
# Input: N = 5, mines = [[4, 2]]
# Output: 2
# Explanation:
# 11111
# 11111
# 11111
# 11111
# 11
# 011
# In the above grid, the largest plus sign can only be order 2.  One of them i
# s marked in bold.
# Order 1:
# 000
# 010
# 000
#
# Order 2:
# 00000
# 00100
# 01110
# 00100
# 00000
#
```

```
# Order 3:
# 0000000
# 0
# 001000
# 0001000
# 0111110
# 0001000
# 0001000
# 0000000

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

class Solution(object):
    def orderOfLargestPlusSign(self, N, mines):
        """
        :type N: int
        :type mines: List[List[int]]
        :rtype: int
        """
        lookup = {tuple(mine) for mine in mines}
        dp = [[0] * N for _ in xrange(N)]
        result = 0
        for i in xrange(N):
            l = 0
            for j in xrange(N):
                l = 0 if (i, j) in lookup else l+1
                dp[i][j] = l
            l = 0
            for j in reversed(xrange(N)):
                l = 0 if (i, j) in lookup else l+1
                dp[i][j] = min(dp[i][j], l)

        for j in xrange(N):
            l = 0
            for i in xrange(N):
                l = 0 if (i, j) in lookup else l+1
                dp[i][j] = min(dp[i][j], l)
            l = 0
            for i in reversed(xrange(N)):
                l = 0 if (i, j) in lookup else l+1
                dp[i][j] = min(dp[i][j], l)
                result = max(result, dp[i][j])
        return result
```

find-critical-and-pseudo-critical-edges-in-minimum-spanning-tree.py

```python
# find-critical-and-pseudo-critical-edges-in-minimum-spanning-tree is not found.
# Time:  O(nlogn)
# Space: O(n)

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:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        self.count -= 1
        return True


class Solution(object):
    def findCriticalAndPseudoCriticalEdges(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: List[List[int]]
        """
        def MST(n, edges, unused=None, used=None):
            union_find = UnionFind(n)
            weight = 0
            if used is not None:
                u, v, w, _ = edges[used]
                if union_find.union_set(u, v):
                    weight += w
            for i, (u, v, w, _) in enumerate(edges):
                if i == unused:
                    continue
                if union_find.union_set(u, v):
                    weight += w
            return weight if union_find.count == 1 else float("inf")

        for i, edge in enumerate(edges):
            edge.append(i)
        edges.sort(key=lambda x: x[2])
        mst = MST(n, edges)
        result = [[], []]
        for i, edge in enumerate(edges):
            if mst < MST(n, edges, unused=i):
                result[0].append(edge[3])
            elif mst == MST(n, edges, used=i):
                result[1].append(edge[3])
        return result
```

reaching-points.py

```
# DESC
# Note:
# Given a starting point (sx, sy) and a target point (tx, ty), return True if and
# only if a sequence of moves exists to transform the point (sx, sy) to (tx, ty).
# Otherwise, return False.
# A move consists of taking a point (x, y) and transforming it to either (x, x+y)
# or (x+y, y).
# (sx, sy)

# NOTE
# sx, sy, tx, ty will all be integers in the range [1, 10^9].

# EXAMPLE
# Examples:
# Input: sx = 1, sy = 1, tx = 3, ty = 5
# Output: True
# Explanation:
# One se
# ries of moves that transforms the starting point to the target is:
# (1, 1) -> (1,
#  2)
# (1, 2) -> (3, 2)
# (3, 2) -> (3, 5)
#
# Input: sx = 1, sy = 1, tx = 2, ty = 2
# Out
# put: False
#
# Input: sx = 1, sy = 1, tx = 1, ty = 1
# Output: True

# Time:  O(log(max(m, n)))
# Space: O(1)

class Solution(object):
    def reachingPoints(self, sx, sy, tx, ty):
        """
        :type sx: int
        :type sy: int
        :type tx: int
        :type ty: int
        :rtype: bool
        """
        while tx >= sx and ty >= sy:
            if tx < ty:
                sx, sy = sy, sx
                tx, ty = ty, tx
            if ty > sy:
                tx %= ty
            else:
                return (tx - sx) % ty == 0

        return False
```

maximum-side-length-of-a-square-with-sum-less-than-or-equal-to-threshold.py

```python
# maximum-side-length-of-a-square-with-sum-less-than-or-equal-to-threshold is not found.
# Time:  O(m * n * log(min(m, n)))
# Space: O(m * n)

class Solution(object):
    def maxSideLength(self, mat, threshold):
        """
        :type mat: List[List[int]]
        :type threshold: int
        :rtype: int
        """
        def check(dp, mid, threshold):
            for i in xrange(mid, len(dp)):
                for j in xrange(mid, len(dp[0])):
                    if dp[i][j] - dp[i-mid][j] - dp[i][j-mid] + dp[i-mid][j-mid] <= threshold:
                        return True
            return False

        dp = [[0 for _ in xrange(len(mat[0])+1)] for _ in xrange(len(mat)+1)]
        for i in xrange(1, len(mat)+1):
            for j in xrange(1, len(mat[0])+1):
                dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + mat[i-1][j-1]

        left, right = 0, min(len(mat), len(mat[0])+1)
        while left <= right:
            mid = left + (right-left)//2
            if not check(dp, mid, threshold):
                right = mid-1
            else:
                left = mid+1
        return right
```

move-sub-tree-of-n-ary-tree.py

```
# move-sub-tree-of-n-ary-tree is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a Node.
class Node(object):
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []


# one pass solution without recursion
class Solution(object):
    def moveSubTree(self, root, p, q):
        """
        :type root: Node
        :type p: Node
        :type q: Node
        :rtype: Node
        """
        def iter_find_parents(node, parent, p, q, is_ancestor, lookup):
            stk = [(1, [node, None, False])]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node, parent, is_ancestor = params
                    if node in (p, q):
                        lookup[node] = parent
                        if len(lookup) == 2:
                            return is_ancestor
                    stk.append((2, [node, is_ancestor, reversed(node.children)]))
                else:
                    node, is_ancestor, it = params
                    child = next(it, None)
                    if not child:
                        continue
                    stk.append((2, [node, is_ancestor, it]))
                    stk.append((1, [child, node, is_ancestor or node == p]))
            assert(False)
            return False

        lookup = {}
        is_ancestor = iter_find_parents(root, None, p, q, False, lookup)
        if p in lookup and lookup[p] == q:
            return root
        q.children.append(p)
        if not is_ancestor:
            lookup[p].children.remove(p)
        else:
            lookup[q].children.remove(q)
            if p == root:
                root = q
            else:
                lookup[p].children[lookup[p].children.index(p)] = q
        return root


# Time:  O(n)
```

```python
# Space: O(h)
# one pass solution with recursion (bad in deep tree)
class Solution_Recu(object):
    def moveSubTree(self, root, p, q):
        """
        :type root: Node
        :type p: Node
        :type q: Node
        :rtype: Node
        """
        def find_parents(node, parent, p, q, is_ancestor, lookup):
            if node in (p, q):
                lookup[node] = parent
                if len(lookup) == 2:
                    return True, is_ancestor
            for child in node.children:
                found, new_is_ancestor = find_parents(child, node, p, q, is_ancestor or node == p, lookup)
                if found:
                    return True, new_is_ancestor
            return False, False

        lookup = {}
        is_ancestor = find_parents(root, None, p, q, False, lookup)[1]
        if p in lookup and lookup[p] == q:
            return root
        q.children.append(p)
        if not is_ancestor:
            lookup[p].children.remove(p)
        else:
            lookup[q].children.remove(q)
            if p == root:
                root = q
            else:
                lookup[p].children[lookup[p].children.index(p)] = q
        return root


# Time:  O(n)
# Space: O(h)
# two pass solution without recursion
class Solution2(object):
    def moveSubTree(self, root, p, q):
        """
        :type root: Node
        :type p: Node
        :type q: Node
        :rtype: Node
        """
        def iter_find_parents(node, parent, p, q, lookup):
            stk = [(1, [node, None])]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node, parent = params
                    if node in (p, q):
                        lookup[node] = parent
                        if len(lookup) == 2:
                            return
                    stk.append((2, [node, reversed(node.children)]))
                else:
```

```
                node, it = params
                child = next(it, None)
                if not child:
                    continue
                stk.append((2, [node, it]))
                stk.append((1, [child, node]))

        def iter_is_ancestor(node, q):
            stk = [(1, [node])]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node = params[0]
                    stk.append((2, [reversed(node.children)]))
                else:
                    it = params[0]
                    child = next(it, None)
                    if not child:
                        continue
                    if child == q:
                        return True
                    stk.append((2, [it]))
                    stk.append((1, [child]))
            return False

        lookup = {}
        iter_find_parents(root, None, p, q, lookup)
        if p in lookup and lookup[p] == q:
            return root
        q.children.append(p)
        if not iter_is_ancestor(p, q):
            lookup[p].children.remove(p)
        else:
            lookup[q].children.remove(q)
            if p == root:
                root = q
            else:
                lookup[p].children[lookup[p].children.index(p)] = q
        return root


# Time:  O(n)
# Space: O(h)
# two pass solution with recursion (bad in deep tree)
class Solution2_Recu(object):
    def moveSubTree(self, root, p, q):
        """
        :type root: Node
        :type p: Node
        :type q: Node
        :rtype: Node
        """
        def find_parents(node, parent, p, q, lookup):
            if node in (p, q):
                lookup[node] = parent
                if len(lookup) == 2:
                    return True
            for child in node.children:
                if find_parents(child, node, p, q, lookup):
                    return True
```

```
    return False

def is_ancestor(node, q):
    for child in node.children:
        if node == q or is_ancestor(child, q):
            return True
    return False

lookup = {}
find_parents(root, None, p, q, lookup)
if p in lookup and lookup[p] == q:
    return root
q.children.append(p)
if not is_ancestor(p, q):
    lookup[p].children.remove(p)
else:
    lookup[q].children.remove(q)
    if p == root:
        root = q
    else:
        lookup[p].children[lookup[p].children.index(p)] = q
return root
```

basic-calculator-iv.py

```
# DESC
# The format of the output is as follows:
# Note:
# expression = "1 + 2 * 3"
# Expressions are evaluated in the usual order: brackets first, then multiplicatio
# n, then addition and subtraction. For example, expression = "1 + 2 * 3" has an a
# nswer of ["7"].
# Given an expression such as expression = "e + 8 - a + 5" and an evaluation map s
# uch as {"e": 1} (given in terms of evalvars = ["e"] and evalints = [1]), return
# a list of tokens representing the simplified expression, such as ["-1*a","14"]
# Examples:

# NOTE
# A chunk is either an expression in parentheses, a variable, or a non-negative in
# teger.
# expression will have length in range [1, 250].
# For each term of free variables with non-zero coefficient, we write the free var
# iables within a term in sorted order lexicographically. For example, we would ne
# ver write a term like "b*a*c", only "a*b*c".
# A variable is a string of lowercase letters (not including digits.) Note that va
# riables can be multiple letters, and note that variables never have a leading co
# efficient or unary operator like "2x" or "-x".
# The leading coefficient of the term is placed directly to the left with an aster
# isk separating it from the variables (if they exist.)  A leading coefficient of
# 1 is still printed.
# An example of a well formatted answer is ["-2*a*a*a", "3*a*a*b", "3*b*b", "4*a",
#  "5*c", "-6"]
# An expression alternates chunks and symbols, with a space separating each chunk
# and symbol.
# Terms (including constant terms) with coefficient 0 are not included.  For examp
# le, an expression of "0" has an output of [].
# evalvars, evalints will have equal lengths in range [0, 100].
# Terms have degree equal to the number of free variables being multiplied, counti
# ng multiplicity. (For example, "a*a*b*c" has degree 4.) We write the largest deg
# ree terms of our answer first, breaking ties by lexicographic order ignoring the
#  leading coefficient of the term.

# EXAMPLE
# Input: expression = "e + 8 - a + 5", evalvars = ["e"], evalints = [1]
# Output: ["
# -1*a","14"]
#
# Input: expression = "e - 8 + temperature - pressure",
# evalvars = ["
# e", "temperature"], evalints = [1, 12]
# Output: ["-1*pressure","5"]
#
# Input: expre
# ssion = "(e + 8) * (e - 8)", evalvars = [], evalints = []
# Output: ["1*e*e","-64"
# ]
#
# Input: expression = "7 - 7", evalvars = [], evalints = []
# Output: []
#
# Input:
# expression = "a * b * c + b * a * c * 4", evalvars = [], evalints = []
# Output: [
```

```
# "5*a*b*c"]
#
# Input: expression = "((a - b) * (b - c) + (c - a)) * ((a - b) + (b -
#  c) * (c - a))",
# evalvars = [], evalints = []
# Output: ["-1*a*a*b*b","2*a*a*b*c",
# "-1*a*a*c*c","1*a*b*b*b","-1*a*b*b*c","-1*a*b*c*c","1*a*c*c*c","-1*b*b*b*c","2*b
# *b*c*c","-1*b*c*c*c","2*a*a*b","-2*a*a*c","-2*a*b*b","2*a*c*c","1*b*b*b","-1*b*b
# *c","1*b*c*c","-1*c*c*c","-1*a*a","1*a*b","1*a*c","-1*b*c"]

# Time:  +:        O(d * t), t is the number of terms,
#                           d is the average degree of terms
#        -:        O(d * t)
#        *:        O(d * t^2)
#        eval:     O(d * t)
#        to_list:  O(d * tlogt)
# Space: O(e + d * t), e is the number of evalvars

import collections
import itertools


class Poly(collections.Counter):
    def __init__(self, expr=None):
        if expr is None:
            return
        if expr.isdigit():
            self.update({(): int(expr)})
        else:
            self[(expr,)] += 1

    def __add__(self, other):
        self.update(other)
        return self

    def __sub__(self, other):
        self.update({k: -v for k, v in other.items()})
        return self

    def __mul__(self, other):
        def merge(k1, k2):
            result = []
            i, j = 0, 0
            while i != len(k1) or j != len(k2):
                if j == len(k2):
                    result.append(k1[i])
                    i += 1
                elif i == len(k1):
                    result.append(k2[j])
                    j += 1
                elif k1[i] < k2[j]:
                    result.append(k1[i])
                    i += 1
                else:
                    result.append(k2[j])
                    j += 1
            return result

        result = Poly()
        for k1, v1 in self.items():
```

```
            for k2, v2 in other.items():
                result.update({tuple(merge(k1, k2)): v1*v2})
        return result

    def eval(self, lookup):
        result = Poly()
        for polies, c in self.items():
            key = []
            for var in polies:
                if var in lookup:
                    c *= lookup[var]
                else:
                    key.append(var)
            result[tuple(key)] += c
        return result

    def to_list(self):
        return ["*".join((str(v),) + k)
                for k, v in sorted(self.items(),
                                   key=lambda x: (-len(x[0]), x[0]))
                if v]


class Solution(object):
    def basicCalculatorIV(self, expression, evalvars, evalints):
        """
        :type expression: str
        :type evalvars: List[str]
        :type evalints: List[int]
        :rtype: List[str]
        """
        def compute(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)

        def parse(s):
            if not s:
                return Poly()
            operands, operators = [], []
            operand = ""
            for i in reversed(xrange(len(s))):
                if s[i].isalnum():
                    operand += s[i]
                    if i == 0 or not s[i-1].isalnum():
                        operands.append(Poly(operand[::-1]))
                        operand = ""
                elif s[i] == ')' or s[i] == '*':
                    operators.append(s[i])
                elif s[i] == '+' or s[i] == '-':
                    while operators and operators[-1] == '*':
                        compute(operands, operators)
                    operators.append(s[i])
                elif s[i] == '(':
                    while operators[-1] != ')':
```

```
                compute(operands, operators)
            operators.pop()
    while operators:
        compute(operands, operators)
    return operands[-1]

lookup = dict(itertools.izip(evalvars, evalints))
return parse(expression).eval(lookup).to_list()
```

simplify-path.py

```
# DESC
# Example 6:
# In a UNIX-style file system, a period . refers to the current directory. Further
# more, a double period .. moves the directory up a level.
# Example 2:
# Given an absolute path for a file (Unix-style), simplify it. Or in other words,
# convert it to the canonical path.
# Example 3:
# Example 1:
# Example 5:
# Note that the returned canonical path must always begin with a slash /, and ther
# e 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 mus
# t be the shortest string representing the absolute path.
# Example 4:

# NOTE
#

# EXAMPLE
# Input: "/a//b////c/d//././/.."
# Output: "/a/b/c"
# Input: "/a/./b/../../c/"
# Output: "/c"
# Input: "/home/"
# Output: "/home"
# Explanation: Note that there is no trailing slas
# h after the last directory name.
# Input: "/../"
# Output: "/"
# Explanation: Going one level up from the root director
# y is a no-op, as the root level is the highest level you can go.
# Input: "/home//foo/"
# Output: "/home/foo"
# Explanation: In the canonical path, mul
# tiple consecutive slashes are replaced by a single one.
# Input: "/a/../../b/../c//.//"
# Output: "/c"

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

class Solution(object):
    # @param path, a string
    # @return a string
    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

```
# DESC
# Example 1:
# Construct the maximum tree by the given array and output the root node of this tree.
# Given an integer array with no duplicates. A maximum tree building on this array
#  is defined as follow:
# Note:

# NOTE
# The size of the given array will be in the range [1,1000].
# The left subtree is the maximum tree constructed from left part subarray divided
#  by the maximum number.
# The root is the maximum number in the array.
# The right subtree is the maximum tree constructed from right part subarray divid
# ed by the maximum number.

# EXAMPLE
# Input: [3,2,1,6,0,5]
# Output: return the tree root node representing the followin
# g tree:
#
#       6
#     /   \
#    3     5
#     \    /
#      2  0
#        \
#
# 1

# 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]
```

check-if-a-string-is-a-valid-sequence-from-root-to-leaves-path-in-a-binary-tree.py

```python
# check-if-a-string-is-a-valid-sequence-from-root-to-leaves-path-in-a-binary-tree is not found.
# Time:  O(n)
# Space: O(w)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


# bfs solution
class Solution(object):
    def isValidSequence(self, root, arr):
        """
        :type root: TreeNode
        :type arr: List[int]
        :rtype: bool
        """
        q = [root]
        for depth in xrange(len(arr)):
            new_q = []
            while q:
                node = q.pop()
                if not node or node.val != arr[depth]:
                    continue
                if depth+1 == len(arr) and node.left == node.right:
                    return True
                new_q.extend(child for child in (node.left, node.right))
            q = new_q
        return False


# Time:  O(n)
# Space: O(h)
# dfs solution with stack
class Solution2(object):
    def isValidSequence(self, root, arr):
        """
        :type root: TreeNode
        :type arr: List[int]
        :rtype: bool
        """
        s = [(root, 0)]
        while s:
            node, depth = s.pop()
            if not node or depth == len(arr) or node.val != arr[depth]:
                continue
            if depth+1 == len(arr) and node.left == node.right:
                return True
            s.append((node.right, depth+1))
            s.append((node.left, depth+1))
        return False


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

```python
# dfs solution with recursion
class Solution3(object):
    def isValidSequence(self, root, arr):
        """
        :type root: TreeNode
        :type arr: List[int]
        :rtype: bool
        """
        def dfs(node, arr, depth):
            if not node or depth == len(arr) or node.val != arr[depth]:
                return False
            if depth+1 == len(arr) and node.left == node.right:
                return True
            return dfs(node.left, arr, depth+1) or dfs(node.right, arr, depth+1)

        return dfs(root, arr, 0)
```

decode-string.py

```python
# DESC
# Given an encoded string, return its decoded string.
# The encoding rule is: k[encoded_string], where the encoded_string inside the squ
# are brackets is being repeated exactly k times. Note that k is guaranteed to be
# a positive integer.
# Example 1:
# You may assume that the input string is always valid; No extra white spaces, squ
# are brackets are well-formed, etc.
# Furthermore, you may assume that the original data does not contain any digits a
# nd that digits are only for those repeat numbers, k. For example, there won't be
#  input like 3a or 2[4].
# Example 2:
# Example 4:
# k[encoded_string]
# Example 3:

# NOTE
#

# EXAMPLE
# Input: s = "3[a]2[bc]"
# Output: "aaabcbc"
# Input: s = "3[a2[c]]"
# Output: "accaccacc"
# Input: s = "abc3[cd]xyz"
# Output: "abccdcdcdxyz"
# Input: s = "2[abc]3[cd]ef"
# Output: "abcabccdcdcdef"

# 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)
```

cherry-pickup.py

```
# cherry-picku is not found.
# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def cherryPickup(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        # dp holds the max # of cherries two k-length paths can pickup.
        # The two k-length paths arrive at (i, k - i) and (j, k - j),
        # respectively.
        n = len(grid)
        dp = [[-1 for _ in xrange(n)] for _ in xrange(n)]
        dp[0][0] = grid[0][0]
        max_len = 2 * (n-1)
        directions = [(0, 0), (-1, 0), (0, -1), (-1, -1)]
        for k in xrange(1, max_len+1):
            for i in reversed(xrange(max(0, k-n+1), min(k+1, n))):  # 0 <= i < n, 0 <= k-i < n
                for j in reversed(xrange(i, min(k+1, n))):          # i <= j < n, 0 <= k-j < n
                    if grid[i][k-i] == -1 or grid[j][k-j] == -1:
                        dp[i][j] = -1
                        continue
                    cnt = grid[i][k-i]
                    if i != j:
                        cnt += grid[j][k-j]
                    max_cnt = -1
                    for direction in directions:
                        ii, jj = i+direction[0], j+direction[1]
                        if ii >= 0 and jj >= 0 and dp[ii][jj] >= 0:
                            max_cnt = max(max_cnt, dp[ii][jj]+cnt)
                    dp[i][j] = max_cnt
        return max(dp[n-1][n-1], 0)
```

smallest-string-with-swaps.py

```python
# smallest-string-with-swaps is not found.
# 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:
            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)
```

maximum-sum-circular-subarray.py

```python
# maximum-sum-circular-subarra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxSubarraySumCircular(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        total, max_sum, cur_max, min_sum, cur_min = 0, -float("inf"), 0, float("inf"), 0
        for a in A:
            cur_max = max(cur_max+a, a)
            max_sum = max(max_sum, cur_max)
            cur_min = min(cur_min+a, a)
            min_sum = min(min_sum, cur_min)
            total += a
        return max(max_sum, total-min_sum) if max_sum >= 0 else max_sum
```

combination-sum-iii.py

```python
# DESC
# Find all possible combinations of k numbers that add up to a number n, given tha
# t only numbers from 1 to 9 can be used and each combination should be a unique s
# et of numbers.
# Example 2:
# Note:
# Example 1:

# NOTE
# All numbers will be positive integers.
# The solution set must not contain duplicate combinations.

# EXAMPLE
# Input: k = 3, n = 7
# Output: [[1,2,4]]
# Input: k = 3, n = 9
# Output: [[1,2,6], [1,3,5], [2,3,4]]

# Time:  O(k * C(n, k))
# Space: O(k)

class Solution(object):
    # @param {integer} k
    # @param {integer} n
    # @return {integer[][]}
    def combinationSum3(self, k, n):
        result = []
        self.combinationSumRecu(result, [], 1, k, n)
        return result

    def combinationSumRecu(self, result, intermediate, start, k, target):
        if k == 0 and target == 0:
            result.append(list(intermediate))
        elif k < 0:
            return
        while start < 10 and start * k + k * (k - 1) / 2 <= target:
            intermediate.append(start)
            self.combinationSumRecu(result, intermediate, start + 1, k - 1, target - start)
            intermediate.pop()
            start += 1
```

optimal-division.py

```
# DESC
# Example:
# However, you can add any number of parenthesis at any position to change the pri
# ority of operations. You should find out how to add parenthesis to get the maxim
# um result, and return the corresponding expression in string format. Your expres
# sion should NOT contain redundant parenthesis.
# Note:
# Given a list of positive integers, the adjacent integers will perform the float
# division. For example, [2,3,4] -> 2 / 3 / 4.

# NOTE
# The length of the input array is [1, 10].
# There is only one optimal division for each test case.
# Elements in the given array will be in range [2, 1000].

# 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 re
# turn "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

# 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)
```

merge-two-binary-trees.py

```
# DESC
# You need to merge them into a new binary tree. The merge rule is that if two nod
# es overlap, then sum node values up as the new value of the merged node. Otherwi
# se, the NOT null node will be used as the node of new tree.
# Given two binary trees and imagine that when you put one of them to cover the ot
# her, some nodes of the two trees are overlapped while the others are not.
# Example 1:
# Note: The merging process must start from the root nodes of both trees.

# NOTE
#

# EXAMPLE
# Input:
#         Tree 1                     Tree 2
#           1
#                  2
#          / \
#    / \
#         3   2                    1   3
#
#        /                          \   \
#
#       5                            4   7
# Output:
# Merged tr
# ee:
#              3
#             / \
#            4   5
#           / \   \
#          5   4   7

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

class Solution(object):
    def mergeTrees(self, t1, t2):
        """
        :type t1: TreeNode
        :type t2: TreeNode
        :rtype: TreeNode
        """
        if t1 is None:
            return t2
        if t2 is None:
            return t1
        t1.val += t2.val
        t1.left = self.mergeTrees(t1.left, t2.left)
        t1.right = self.mergeTrees(t1.right, t2.right)
        return t1
```

remove-invalid-parentheses.py

```python
# DESC
# Remove the minimum number of invalid parentheses in order to make the input stri
# ng valid. Return all possible results.
# Note: The input string may contain letters other than the parentheses ( and ).
# Example 2:
# Example 3:
# Example 1:

# NOTE
# OR, we can discard the bracket and move on.
# We keep the bracket and add it to the expression that we are building on the fly
#  during recursion.

# EXAMPLE
# Input: "()())()"
# Output: ["()()()", "(())()"]
# Input: ")("
# Output: [""]
# Input: "(a())()"
# Output: ["(a)()()", "(a())()"]

# Time:  O(C(n, c)), try out all possible substrings with the minimum c deletion.
# Space: O(c), the depth is at most c, and it costs n at each depth

class Solution(object):
    def removeInvalidParentheses(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        # Calculate the minimum left and right parantheses to remove
        def findMinRemove(s):
            left_removed, right_removed = 0, 0
            for c in s:
                if c == '(':
                    left_removed += 1
                elif c == ')':
                    if not left_removed:
                        right_removed += 1
                    else:
                        left_removed -= 1
            return (left_removed, right_removed)

        # Check whether s is valid or not.
        def isValid(s):
            sum = 0
            for c in s:
                if c == '(':
                    sum += 1
                elif c == ')':
                    sum -= 1
                if sum < 0:
                    return False
            return sum == 0

        def removeInvalidParenthesesHelper(start, left_removed, right_removed):
            if left_removed == 0 and right_removed == 0:
                tmp = ""
```

```
            for i, c in enumerate(s):
                if i not in removed:
                    tmp += c
            if isValid(tmp):
                res.append(tmp)
            return

        for i in xrange(start, len(s)):
            if right_removed == 0 and left_removed > 0 and s[i] == '(':
                if i == start or s[i] != s[i - 1]:  # Skip duplicated.
                    removed[i] = True
                    removeInvalidParenthesesHelper(i + 1, left_removed - 1, right_removed)
                    del removed[i]
            elif right_removed > 0 and s[i] == ')':
                if i == start or s[i] != s[i - 1]:  # Skip duplicated.
                    removed[i] = True
                    removeInvalidParenthesesHelper(i + 1, left_removed, right_removed - 1)
                    del removed[i]

    res, removed = [], {}
    (left_removed, right_removed) = findMinRemove(s)
    removeInvalidParenthesesHelper(0, left_removed, right_removed)
    return res
```

construct-k-palindrome-strings.py

```python
# construct-k-palindrome-strings is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def canConstruct(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: bool
        """
        count = collections.Counter(s)
        odd = sum(v%2 for v in count.itervalues())
        return odd <= k <= len(s)
```

remove-all-adjacent-duplicates-in-string-ii.py

```
# DESC
# Return the final string after all such duplicate removals have been made.
# Example 3:
# It is guaranteed that the answer is unique.
# Given a string s, a k duplicate removal consists of choosing k adjacent and equa
# l letters from s and removing them causing the left and the right side of the de
# leted substring to concatenate together.
# We repeatedly make k duplicate removals on s until we no longer can.
# Constraints:
# Example 1:
# Example 2:

# NOTE
# 1 <= s.length <= 10^5
# s only contains lower case English letters.
# 2 <= k <= 10^4

# EXAMPLE
# Input: s = "pbbcggttciiippooaais", k = 2
# Output: "ps"
# 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"
# Input: s = "abcd", k = 2
# Output: "abcd"
# Explanation: There's nothing to delete.

# 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)
```

min-stack.py

```
# DESC
# Constraints:
# Design a stack that supports push, pop, top, and retrieving the minimum element
# in constant time.
# Example 1:

# NOTE
# getMin() -- Retrieve the minimum element in the stack.
# pop() -- Removes the element on top of the stack.
# push(x) -- Push element x onto stack.
# Methods pop, top and getMin operations will always be called on non-empty stacks.
# top() -- Get the top element.

# EXAMPLE
# Input
# ["MinStack","push","push","push","getMin","pop","top","getMin"]
# [[],[-2],[
# 0],[-3],[],[],[],[]]
#
# Output
# [null,null,null,null,-3,null,0,-2]
#
# Explanation
# Min
# Stack minStack = new MinStack();
# minStack.push(-2);
# minStack.push(0);
# minStack.p
# ush(-3);
# minStack.getMin(); // return -3
# minStack.pop();
# minStack.top();    // r
# eturn 0
# minStack.getMin(); // return -2

# Time:  O(n)
# Space: O(1)

class MinStack(object):
    def __init__(self):
        self.min = None
        self.stack = []

    # @param x, an integer
    # @return an integer
    def push(self, x):
        if not self.stack:
            self.stack.append(0)
            self.min = x
        else:
            self.stack.append(x - self.min)
            if x < self.min:
                self.min = x

    # @return nothing
    def pop(self):
        x = self.stack.pop()
        if x < 0:
```

```
            self.min = self.min - x

    # @return an integer
    def top(self):
        x = self.stack[-1]
        if x > 0:
            return x + self.min
        else:
            return self.min

    # @return an integer
    def getMin(self):
        return self.min

# Time:  O(n)
# Space: O(n)
class MinStack2(object):
    def __init__(self):
        self.stack, self.minStack = [], []
    # @param x, an integer
    # @return an integer
    def push(self, x):
        self.stack.append(x)
        if len(self.minStack):
            if x < self.minStack[-1][0]:
                self.minStack.append([x, 1])
            elif x == self.minStack[-1][0]:
                self.minStack[-1][1] += 1
        else:
            self.minStack.append([x, 1])

    # @return nothing
    def pop(self):
        x = self.stack.pop()
        if x == self.minStack[-1][0]:
            self.minStack[-1][1] -= 1
            if self.minStack[-1][1] == 0:
                self.minStack.pop()

    # @return an integer
    def top(self):
        return self.stack[-1]

    # @return an integer
    def getMin(self):
        return self.minStack[-1][0]

# time: O(1)
# space: O(n)

class MinStack3(object):

    def __init__(self):
        self.stack = []

    def push(self, x):
        if self.stack:
            current_min = min(x, self.stack[-1][0])
            self.stack.append((current_min, x))
        else:
```

```
        self.stack.append((x, x))

    def pop(self):
        return self.stack.pop()[1]

    def top(self):
        return self.stack[-1][1]

    def getMin(self):
        return self.stack[-1][0]
```

add-one-row-to-tree.py

```
# DESC
# Example 2:
# The adding rule is: given a positive integer depth d, for each NOT null tree nod
# es N in depth d-1, create two tree nodes with value v as N's left subtree root a
# nd right subtree root. And N's original left subtree should be the left subtree
# of the new left subtree root, its original right subtree should be the right sub
# tree of the new right subtree root. If depth d is 1 that means there is no depth
#  d-1 at all, then create a tree node with value v as the new root of the whole o
# riginal tree, and the original tree is the new root's left subtree.
# Given the root of a binary tree, then value v and depth d, you need to add a row
#  of nodes with value v at the given depth d. The root node is at depth 1.
# Note:
# Example 1:

# NOTE
# The given binary tree has at least one tree node.
# The given d is in range [1, maximum depth of the given tree + 1].

# EXAMPLE
# Input:
# A binary tree as following:
#       4
#      /
#     2
#    / \
#   3
# 1
#
# v = 1
#
# d = 3
#
# Output:
#       4
#      /
#     2
#    / \
#   1   1
#  /
# \
# 3       1
# Input:
# A binary tree as following:
#        4
#      /   \
#     2     6
#    / \   /
#
#   3   1 5
#
# v = 1
#
# d = 2
#
# Output:
#        4
#       / \
#      1   1
```

```
#     /     \
#
#   2       6
#   / \     /
#  3   1   5

# 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 addOneRow(self, root, v, d):
        """
        :type root: TreeNode
        :type v: int
        :type d: int
        :rtype: TreeNode
        """
        if d in (0, 1):
            node = TreeNode(v)
            if d == 1:
                node.left = root
            else:
                node.right = root
            return node
        if root and d >= 2:
            root.left = self.addOneRow(root.left,  v, d-1 if d > 2 else 1)
            root.right = self.addOneRow(root.right, v, d-1 if d > 2 else 0)
        return root
```

two-sum-iii-data-structure-design.py

```python
# two-sum-iii-data-structure-design is not found.
# Time:  O(n)
# Space: O(n)

from collections import defaultdict

class TwoSum(object):

    def __init__(self):
        """
        initialize your data structure here
        """
        self.lookup = defaultdict(int)



    def add(self, number):
        """
        Add the number to an internal data structure.
        :rtype: nothing
        """
        self.lookup[number] += 1


    def find(self, value):
        """
        Find if there exists any pair of numbers which sum is equal to the value.
        :type value: int
        :rtype: bool
        """
        for key in self.lookup:
            num = value - key
            if num in self.lookup and (num != key or self.lookup[key] > 1):
                return True
        return False
```

greatest-common-divisor-of-strings.py

```python
# DESC
# Example 1:
# Return the largest string X such that X divides str1 and X divides str2.
# Example 3:
# For strings S and T, we say "T divides S" if and only if S = T + ... + T  (T con
# catenated with itself 1 or more times)
# Note:
# Example 2:

# NOTE
# 1 <= str1.length <= 1000
# 1 <= str2.length <= 1000
# str1[i] and str2[i] are English uppercase letters.

# EXAMPLE
# Input: str1 = "ABABAB", str2 = "ABAB"
# Output: "AB"
# Input: str1 = "ABCABC", str2 = "ABC"
# Output: "ABC"
# Input: str1 = "LEET", str2 = "CODE"
# Output: ""

# Time:  O(m + n)
# Space: O(1)

class Solution(object):
    def gcdOfStrings(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """
        def check(s, common):
            i = 0
            for c in s:
                if c != common[i]:
                    return False
                i = (i+1)%len(common)
            return True

        def gcd(a, b):  # Time: O((logn)^2)
            while b:
                a, b = b, a % b
            return a

        if not str1 or not str2:
            return ""
        c = gcd(len(str1), len(str2))
        result = str1[:c]
        return result if check(str1, result) and check(str2, result) else ""
```

my-calendar-ii.py

```
# DESC
# Note:
# Example 1:
# book(int start, int end)
# Your class will have one method, book(int start, int end). Formally, this repres
# ents a booking on the half open interval [start, end), the range of real numbers
#  x such that start <= x < end.
# A triple booking happens when three events have some non-empty intersection (ie.
# , there is some time that is common to all 3 events.)
# Implement a MyCalendarTwo class to store your events. A new event can be added i
# f adding the event will not cause a triple booking.
# For each call to the method MyCalendar.book, return true if the event can be add
# ed to the calendar successfully without causing a triple booking. Otherwise, ret
# urn false and do not add the event to the calendar.

# NOTE
# In calls to MyCalendar.book(start, end), start and end are integers in the range
#  [0, 10^9].
# The number of calls to MyCalendar.book per test case will be at most 1000.

# EXAMPLE
# MyCalendar();
# MyCalendar.book(10, 20); // returns true
# MyCalendar.book(50, 60);
# // returns true
# MyCalendar.book(10, 40); // returns true
# MyCalendar.book(5, 15);
#  // returns false
# MyCalendar.book(5, 10); // returns true
# MyCalendar.book(25, 55
# ); // returns true
# Explanation:
# The first two events can be booked.  The third
# event can be double booked.
# The fourth event (5, 15) can't be booked, because it
#  would result in a triple booking.
# The fifth event (5, 10) can be booked, as it
# does not use time 10 which is already double booked.
# The sixth event (25, 55) ca
# n be booked, as the time in [25, 40) will be double booked with the third event;
#
# the time [40, 50) will be single booked, and the time [50, 55) will be double b
# ooked with the second event.

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

class MyCalendarTwo(object):

    def __init__(self):
        self.__overlaps = []
        self.__calendar = []


    def book(self, start, end):
        """
        :type start: int
        :type end: int
```

```
        :rtype: bool
        """
        for i, j in self.__overlaps:
            if start < j and end > i:
                return False
        for i, j in self.__calendar:
            if start < j and end > i:
                self.__overlaps.append((max(start, i), min(end, j)))
        self.__calendar.append((start, end))
        return True
```

ipo.py

```
# DESC
# Suppose LeetCode will start its IPO soon. In order to sell a good price of its s
# hares to Venture Capital, LeetCode would like to work on some projects to increa
# se its capital before the IPO. Since it has limited resources, it can only finis
# h at most k distinct projects before the IPO. Help LeetCode design the best way
# to maximize its total capital after finishing at most k distinct projects.
# Note:
# You are given several projects. For each project i, it has a pure profit Pi and
# a minimum capital of Ci is needed to start the corresponding project. Initially,
#  you have W capital. When you finish a project, you will obtain its pure profit
# and the profit will be added to your total capital.
# To sum up, pick a list of at most k distinct projects from given projects to max
# imize your final capital, and output your final maximized capital.
# Example 1:

# NOTE
# You may assume all numbers in the input are non-negative integers.
# The length of Profits array and Capital array will not exceed 50,000.
# The answer is guaranteed to fit in a 32-bit signed integer.

# EXAMPLE
# Input: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].
#
# Output: 4
#
# Explanation: Sinc
# e your initial capital is 0, you can only start the project indexed 0.
#
#     After finishing it you will obtain profit 1 and your capital becomes 1.
#
#          With capital 1, you can either start the project indexed 1 or the proje
# ct indexed 2.
#              Since you can choose at most 2 projects, you need to
# finish the project indexed 2 to get the maximum capital.
#              Therefore,
#  output the final maximized capital, which is 0 + 1 + 3 = 4.

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

import heapq


class Solution(object):
    def findMaximizedCapital(self, k, W, Profits, Capital):
        """
        :type k: int
        :type W: int
        :type Profits: List[int]
        :type Capital: List[int]
        :rtype: int
        """
        curr = []
        future = sorted(zip(Capital, Profits), reverse=True)
        for _ in xrange(k):
            while future and future[-1][0] <= W:
                heapq.heappush(curr, -future.pop()[1])
            if curr:
```

```
        W -= heapq.heappop(curr)
return W
```

longest-word-in-dictionary.py

```python
# longest-word-in-dictionar is not found.
# Time:  O(n), n is the total sum of the lengths of words
# Space: O(t), t is the number of nodes in trie

from collections import defaultdict
from operator import getitem


class Solution(object):
    def longestWord(self, words):
        """
        :type words: List[str]
        :rtype: str
        """
        _trie = lambda: defaultdict(_trie)
        trie = _trie()
        for i, word in enumerate(words):
            reduce(getitem, word, trie)["_end"] = i

        # DFS
        stack = trie.values()
        result = ""
        while stack:
            curr = stack.pop()
            if "_end" in curr:
                word = words[curr["_end"]]
                if len(word) > len(result) or (len(word) == len(result) and word < result):
                    result = word
                stack += [curr[letter] for letter in curr if letter != "_end"]
        return result
```

longest-substring-with-at-most-k-distinct-characters.py

```python
# longest-substring-with-at-most-k-distinct-characters is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def lengthOfLongestSubstringKDistinct(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        longest, start, distinct_count, visited = 0, 0, 0, [0 for _ in xrange(256)]
        for i, char in enumerate(s):
            if visited[ord(char)] == 0:
                distinct_count += 1
            visited[ord(char)] += 1
            while distinct_count > k:
                visited[ord(s[start])] -= 1
                if visited[ord(s[start])] == 0:
                    distinct_count -= 1
                start += 1
            longest = max(longest, i - start + 1)
        return longest


# Time:  O(n)
# Space: O(1)
from collections import Counter


class Solution2(object):
    def lengthOfLongestSubstringKDistinct(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        counter = Counter()
        left, max_length = 0, 0
        for right, char in enumerate(s):
            counter[char] += 1
            while len(counter) > k:
                counter[s[left]] -= 1
                if counter[s[left]] == 0:
                    del counter[s[left]]
                left += 1
            max_length = max(max_length, right-left+1)
        return max_length
```

clone-graph.py

```
# DESC
# Test case format:
# Constraints:
# Given a reference of a node in a connected undirected graph.
# Example 1:
# The given node will always be the first node with val = 1. You must return the c
# opy of the given node as a reference to the cloned graph.
# Example 4:
# Example 2:
# Adjacency list is a collection of unordered lists used to represent a finite gra
# ph. Each list describes the set of neighbors of a node in the graph.
# Example 3:
# For simplicity sake, each node's value is the same as the node's index (1-indexe
# d). 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.
# Each node in the graph contains a val (int) and a list (List[Node]) of its neighbors.
# Return a deep copy (clone) of the graph.

# NOTE
# The Graph is connected and all nodes can be visited starting from the given node.
# Number of Nodes will not exceed 100.
# Node.val is unique for each node.
# There is no repeated edges and no self-loops in the graph.
# 1 <= Node.val <= 100

# EXAMPLE
# class Node {
#     public int val;
#     public List<Node> neighbors;
# }
# 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 ha
# ve any neighbors.
# Input: adjList = [[2],[1]]
# Output: [[2],[1]]
# Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
# Output: [[2,4],[1,3],[2,4],[1,3]]
# Exp
# lanation: 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 n
# ode (val = 1) and 3rd node (val = 3).
# 3rd node (val = 3)'s neighbors are 2nd nod
# e (val = 2) and 4th node (val = 4).
# 4th node (val = 4)'s neighbors are 1st node
# (val = 1) and 3rd node (val = 3).
# Input: adjList = []
# Output: []
# Explanation: This an empty graph, it does not hav
# e any nodes.

# 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]
```

numbers-at-most-n-given-digit-set.py

```python
# DESC
# Now, we write numbers using these digits, using each digit as many times as we w
# ant.  For example, if D = {'1','3','5'}, we may write numbers such as '13', '551
# ', '1351315'.
# Return the number of positive integers that can be written (using the digits of
# D) that are less than or equal to N.
# Example 1:
# We have a sorted set of digits D, a non-empty subset of {'1','2','3','4','5','6'
# ,'7','8','9'}.  (Note that '0' is not included.)
# D = {'1','3','5'}
# Note:
# Example 2:

# NOTE
# D is a subset of digits '1'-'9' in sorted order.
# 1 <= N <= 10^9

# EXAMPLE
# Input: D = ["1","4","9"], N = 1000000000
# Output: 29523
# Explanation:
# We can writ
# e 3 one digit numbers, 9 two digit numbers, 27 three digit numbers,
# 81 four digi
# t numbers, 243 five digit numbers, 729 six digit numbers,
# 2187 seven digit numbe
# rs, 6561 eight digit numbers, and 19683 nine digit numbers.
# In total, this is 29
# 523 integers that can be written using the digits of D.
# Input: D = ["1","3","5","7"], N = 100
# Output: 20
# Explanation:
# The 20 numbers th
# at can be written are:
# 1, 3, 5, 7, 11, 13, 15, 17, 31, 33, 35, 37, 51, 53, 55, 5
# 7, 71, 73, 75, 77.

# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def atMostNGivenDigitSet(self, D, N):
        """
        :type D: List[str]
        :type N: int
        :rtype: int
        """
        str_N = str(N)
        set_D = set(D)
        result = sum(len(D)**i for i in xrange(1, len(str_N)))
        i = 0
        while i < len(str_N):
            result += sum(c < str_N[i] for c in D) * (len(D)**(len(str_N)-i-1))
            if str_N[i] not in set_D:
                break
            i += 1
        return result + int(i == len(str_N))
```

how-many-apples-can-you-put-into-the-basket.py

```python
# how-many-apples-can-you-put-into-the-basket is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def maxNumberOfApples(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        LIMIT = 5000
        arr.sort()
        result, total = 0, 0
        for x in arr:
            if total+x > LIMIT:
                break
            total += x
            result += 1
        return result
```

tree-diameter.py

```python
# tree-diameter is not found.
# Time:  O(|V| + |E|)
# Space: O(|E|)

import collections


class Solution(object):
    def treeDiameter(self, edges):
        """
        :type edges: List[List[int]]
        :rtype: int
        """
        graph, length = collections.defaultdict(set), 0
        for u, v in edges:
            graph[u].add(v)
            graph[v].add(u)
        curr_level = {(None, u) for u, neighbors in graph.iteritems() if len(neighbors) == 1}
        while curr_level:
            curr_level = {(u, v) for prev, u in curr_level
                          for v in graph[u] if v != prev}
            length += 1
        return max(length-1, 0)
```

wiggle-sort-ii.py

```
# DESC
# Follow Up:
#
# Can you do it in O(n) time and/or in-place with O(1) extra space?
# Note:
#
# You may assume all input has valid answer.
# Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] <
#  nums[3]....
# Example 2:
# Example 1:

# NOTE
#

# EXAMPLE
# Input: nums = [1, 3, 2, 2, 3, 1]
# Output: One possible answer is [2, 3, 1, 3, 1, 2].
# Input: nums = [1, 5, 1, 1, 6, 4]
# Output: One possible answer is [1, 4, 1, 5, 1, 6].

# 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])
```

toss-strange-coins.py

```python
# toss-strange-coins is not found.
# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def probabilityOfHeads(self, prob, target):
        """
        :type prob: List[float]
        :type target: int
        :rtype: float
        """
        dp = [0.0]*(target+1)
        dp[0] = 1.0
        for p in prob:
            for i in reversed(xrange(target+1)):
                dp[i] = (dp[i-1] if i >= 1 else 0.0)*p + dp[i]*(1-p)
        return dp[target]
```

unique-binary-search-trees-ii.py

```python
# DESC
# Constraints:
# Example:
# Given an integer n, generate all structurally unique BST's (binary search trees)
#  that store values 1 ... n.

# NOTE
# 0 <= n <= 8

# 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 uniq
# ue BST's shown below:
#
#    1         3     3      2      1
#     \       /     /
#    / \       \
#      3     2     1      1   3      2
#     /     /       \
#        \
#    2     1         2                3

# 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)

        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
```

robot-room-cleaner.py

```python
# robot-room-cleaner is not found.
# Time:  O(n), n is the number of cells
# Space: O(n)

class Solution(object):
    def cleanRoom(self, robot):
        """
        :type robot: Robot
        :rtype: None
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        def goBack(robot):
            robot.turnLeft()
            robot.turnLeft()
            robot.move()
            robot.turnRight()
            robot.turnRight()

        def dfs(pos, robot, d, lookup):
            if pos in lookup:
                return
            lookup.add(pos)

            robot.clean()
            for _ in directions:
                if robot.move():
                    dfs((pos[0]+directions[d][0],
                         pos[1]+directions[d][1]),
                        robot, d, lookup)
                    goBack(robot)
                robot.turnRight()
                d = (d+1) % len(directions)

        dfs((0, 0), robot, 0, set())
```

remove-nth-node-from-end-of-list.py

```python
# DESC
# Note:
# Example:
# Follow up:
# Given n will always be valid.
# Could you do this in one pass?
# Given a linked list, remove the n-th node from the end of list and return its head.

# NOTE
#

# EXAMPLE
# Given linked list: 1->2->3->4->5, and n = 2.
#
# After removing the second node fro
# m the end, the linked list becomes 1->2->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 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
```

shortest-palindrome.py

```
# DESC
# Example 2:
# Given a string s, you are allowed to convert it to a palindrome by adding charac
# ters in front of it. Find and return the shortest palindrome you can find by per
# forming this transformation.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: "aacecaaa"
# Output: "aaacecaaa"
# Input: "abcd"
# Output: "dcbabcd"

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

class Solution(object):
    def shortestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        def getPrefix(pattern):
            prefix = [-1] * len(pattern)
            j = -1
            for i in xrange(1, len(pattern)):
                while j > -1 and pattern[j+1] != pattern[i]:
                    j = prefix[j]
                if pattern[j+1] == pattern[i]:
                    j += 1
                prefix[i] = j
            return prefix

        if not s:
            return s

        A = s + s[::-1]
        prefix = getPrefix(A)
        i = prefix[-1]
        while i >= len(s):
            i = prefix[i]
        return s[i+1:][::-1] + s


# Time:  O(n)
# Space: O(n)
# Manacher's Algorithm
class Solution2(object):
    def shortestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        def preProcess(s):
            if not s:
```

```
        return ['^', '$']
    string = ['^']
    for c in s:
        string +=  ['#', c]
    string += ['#', '$']
    return string

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

    while string[i + 1 + palindrome[i]] == string[i - 1 - palindrome[i]]:
        palindrome[i] += 1

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

max_len = 0
for i in xrange(1, len(string) - 1):
    if i - palindrome[i] == 1:
        max_len = palindrome[i]
return s[len(s)-1:max_len-1:-1] + s
```

valid-square.py

```
# DESC
# The coordinate (x,y) of a point is represented by an integer array with two integers.
# Example:
# Given the coordinates of four points in 2D space, return whether the four points
#  could construct a square.
# Note:

# NOTE
# All the input integers are in the range [-10000, 10000].
# Input points have no order.
# A valid square has four equal sides with positive length and four equal angles (
# 90-degree angles).

# EXAMPLE
# Input: p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]
# Output: True

# 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
```

paint-house-ii.py

```
# paint-house-ii is not found.
# Time:  O(n * k)
# Space: O(k)

class Solution2(object):
    def minCostII(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        return min(reduce(self.combine, costs)) if costs else 0

    def combine(self, tmp, house):
        smallest, k, i = min(tmp), len(tmp), tmp.index(min(tmp))
        tmp, tmp[i] = [smallest] * k, min(tmp[:i] + tmp[i+1:])
        return map(sum, zip(tmp, house))


class Solution2(object):
    def minCostII(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        if not costs:
            return 0

        n = len(costs)
        k = len(costs[0])
        min_cost = [costs[0], [0] * k]
        for i in xrange(1, n):
            smallest, second_smallest = float("inf"), float("inf")
            for j in xrange(k):
                if min_cost[(i - 1) % 2][j] < smallest:
                    smallest, second_smallest = min_cost[(i - 1) % 2][j], smallest
                elif min_cost[(i - 1) % 2][j] < second_smallest:
                    second_smallest = min_cost[(i - 1) % 2][j]
            for j in xrange(k):
                min_j = smallest if min_cost[(i - 1) % 2][j] != smallest else second_smallest
                min_cost[i % 2][j] = costs[i][j] + min_j

        return min(min_cost[(n - 1) % 2])
```

water-and-jug-problem.py

```python
# DESC
# Constraints:
# Operations allowed:
# You are given two jugs with capacities x and y litres. There is an infinite amou
# nt of water supply available. You need to determine whether it is possible to me
# asure exactly z litres using these two jugs.
# Example 2:
# Example 1: (From the famous "Die Hard" example)
# If z liters of water is measurable, you must have z liters of water contained wi
# thin one or both buckets by the end.

# NOTE
# Fill any of the jugs completely with water.
# Pour water from one jug into another till the other jug is completely full or th
# e first jug itself is empty.
# 0 <= y <= 10^6
# Empty any of the jugs.
# 0 <= z <= 10^6
# 0 <= x <= 10^6

# EXAMPLE
# Input: x = 2, y = 6, z = 5
# Output: False
# Input: x = 3, y = 5, z = 4
# Output: True

# 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))
```

set-intersection-size-at-least-two.py

```
# DESC
# Find the minimum size of a set S such that for every integer interval A in inter
# vals, the intersection of S with A has size at least 2.
# Example 1:
# Example 2:
# intervals
# An integer interval [a, b] (for integers a < b) is a set of all consecutive inte
# gers from a to b, including a and b.
# Note:

# NOTE
# intervals will have length in range [1, 3000].
# intervals[i] will have length 2, representing some integer interval.
# intervals[i][j] will be an integer in [0, 10^8].

# EXAMPLE
# Input: intervals = [[1, 2], [2, 3], [2, 4], [4, 5]]
# Output: 5
# Explanation:
# An ex
# ample of a minimum sized set is {1, 2, 3, 4, 5}.
# Input: intervals = [[1, 3], [1, 4], [2, 5], [3, 5]]
# Output: 3
# Explanation:
# Consi
# der the set S = {2, 3, 4}.  For each interval, there are at least 2 elements fro
# m S in the interval.
# Also, there isn't a smaller size set that fulfills the abov
# e condition.
# Thus, we output the size of this set, which is 3.

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

class Solution(object):
    def intersectionSizeTwo(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: int
        """
        intervals.sort(key = lambda s_e: (s_e[0], -s_e[1]))
        cnts = [2] * len(intervals)
        result = 0
        while intervals:
            (start, _), cnt = intervals.pop(), cnts.pop()
            for s in xrange(start, start+cnt):
                for i in xrange(len(intervals)):
                    if cnts[i] and s <= intervals[i][1]:
                        cnts[i] -= 1
            result += cnt
        return result
```

largest-palindrome-product.py

```
# DESC
# Note:
# Since the result could be very large, you should return the largest palindrome m
# od 1337.
# Explanation: 99 x 91 = 9009, 9009 % 1337 = 987
# The range of n is [1,8].
# Find the largest palindrome made from the product of two n-digit numbers.
# Output: 987
# Example:
# Input: 2

# NOTE
#

# EXAMPLE
#

# Time:  O(10^(2n))
# Space: O(n)

class Solution_TLE(object):
    def largestPalindrome(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n == 1:
            return 9

        upper, lower = 10**n-1, 10**(n-1)
        for i in reversed(xrange(lower, upper+1)):
            candidate = int(str(i) + str(i)[::-1])
            j = upper
            while j * j >= candidate:
                if candidate % j == 0:
                    return candidate % 1337
                j -= 1
        return -1
```

rabbits-in-forest.py

```
# DESC
# Note:
# In a forest, each rabbit has some color. Some subset of rabbits (possibly all of
#  them) tell you how many other rabbits have the same color as them. Those answer
# s are placed in an array.
# Return the minimum number of rabbits that could be in the forest.

# NOTE
# answers will have length at most 1000.
# Each answers[i] will be an integer in the range [0, 999].

# EXAMPLE
# Examples:
# Input: answers = [1, 1, 2]
# Output: 5
# Explanation:
# The two rabbits that
#  answered "1" could both be the same color, say red.
# The rabbit than answered "2
# " can't be red or the answers would be inconsistent.
# Say the rabbit that answere
# d "2" was blue.
# Then there should be 2 other blue rabbits in the forest that did
# n't answer into the array.
# The smallest possible number of rabbits in the forest
#  is therefore 5: 3 that answered plus 2 that didn't.
#
# Input: answers = [10, 10,
# 10]
# Output: 11
#
# Input: answers = []
# Output: 0

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

import collections


class Solution(object):
    def numRabbits(self, answers):
        """
        :type answers: List[int]
        :rtype: int
        """
        count = collections.Counter(answers)
        return sum((((k+1)+v-1)//(k+1))*(k+1) for k, v in count.iteritems())
```

swim-in-rising-water.py

```
# DESC
# On an N x N grid, each square grid[i][j] represents the elevation at that point
# (i,j).
# Now rain starts to fall. At time t, the depth of the water everywhere is t. You
# can swim from a square to another 4-directionally adjacent square if and only if
#  the elevation of both squares individually are at most t. You can swim infinite
#  distance in zero time. Of course, you must stay within the boundaries of the gr
# id during your swim.
# You start at the top left square (0, 0). What is the least time until you can re
# ach the bottom right square (N-1, N-1)?
# Note:
# Example 1:
# Example 2:

# NOTE
# 2 <= N <= 50.
# grid[i][j] is a permutation of [0, ..., N*N - 1].

# EXAMPLE
# Input: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,
# 6]]
# Output: 16
# Explanation:
#  0  1  2  3  4
# 24 23 22 21  5
# 12 13 14 15 16
# 11 17 1
# 8 19 20
# 10  9  8  7  6
#
# The final route is marked in bold.
# We need to wait until
#  time 16 so that (0, 0) and (4, 4) are connected.
# Input: [[0,2],[1,3]]
# Output: 3
# Explanation:
# At time 0, you are in grid location
# (0, 0).
# You cannot go anywhere else because 4-directionally adjacent neighbors h
# ave a higher elevation than t = 0.
#
# You cannot reach point (1, 1) until time 3.
#
# When the depth of water is 3, we can swim anywhere inside the grid.

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

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 swimInWater(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        n = len(grid)
        positions = [None] * (n**2)
        for i in xrange(n):
            for j in xrange(n):
                positions[grid[i][j]] = (i, j)
        directions = ((-1, 0), (1, 0), (0, -1), (0, 1))

        union_find = UnionFind(n**2)
        for elevation in xrange(n**2):
            i, j = positions[elevation]
            for direction in directions:
                x, y = i+direction[0], j+direction[1]
                if 0 <= x < n and 0 <= y < n and grid[x][y] <= elevation:
                    union_find.union_set(i*n+j, x*n+y)
                    if union_find.find_set(0) == union_find.find_set(n**2-1):
                        return elevation
        return n**2-1
```

equal-rational-numbers.py

```
# DESC
# Given two strings S and T, each of which represents a non-negative rational numb
# er, return True if and only if they represent the same number. The strings may u
# se parentheses to denote the repeating part of the rational number.
# Note:
# Example 3:
# Example 2:
# 1 / 6 = 0.16666666... = 0.1(6) = 0.1666(6) = 0.166(66)
# In general a rational number can be represented using up to three parts: an inte
# ger part, a non-repeating part, and a repeating part. The number will be represe
# nted in one of the following three ways:
# The repeating portion of a decimal expansion is conventionally denoted within a
# pair of round brackets.  For example:
# Both 0.1(6) or 0.1666(6) or 0.166(66) are correct representations of 1 / 6.
# Example 1:

# NOTE
# The <IntegerPart> will not begin with 2 or more zeros.  (There is no other restr
# iction on the digits of each part.)
# <IntegerPart><.><NonRepeatingPart>  (e.g. 0.5, 1., 2.12, 2.0001)
# <IntegerPart><.><NonRepeatingPart><(><RepeatingPart><)> (e.g. 0.1(6), 0.9(9), 0.
# 00(1212))
# <IntegerPart> (e.g. 0, 12, 123)
# 1 <= <IntegerPart>.length <= 4
# Each part consists only of digits.
# 1 <= <RepeatingPart>.length <= 4
# 0 <= <NonRepeatingPart>.length <= 4

# EXAMPLE
# Input: S = "0.1666(6)", T = "0.166(66)"
# Output: true
# Input: S = "0.(52)", T = "0.5(25)"
# Output: true
# Explanation:
# Because "0.(52)" re
# presents 0.52525252..., and "0.5(25)" represents 0.52525252525..... , the string
# s represent the same number.
# Input: S = "0.9(9)", T = "1."
# Output: true
# Explanation:
# "0.9(9)" represents 0.9
# 99999999... repeated forever, which equals 1.  [See this link for an explanation
# .]
# "1." represents the number 1, which is formed correctly: (IntegerPart) = "1"
# and (NonRepeatingPart) = "".

# Time:  O(1)
# Space: O(1)

from fractions import Fraction


class Solution(object):
    def isRationalEqual(self, S, T):
        """
        :type S: str
        :type T: str
        :rtype: bool
```

```
    """
    def frac(S):
        if '.' not in S:
            return Fraction(int(S), 1)

        i = S.index('.')
        result = Fraction(int(S[:i]), 1)
        non_int_part = S[i+1:]
        if '(' not in non_int_part:
            if non_int_part:
                result += Fraction(int(non_int_part), 10**len(non_int_part))
            return result

        i = non_int_part.index('(')
        if i:
            result += Fraction(int(non_int_part[:i]), 10**i)
        repeat_part = non_int_part[i+1:-1]
        result += Fraction(int(repeat_part), 10**i * (10**len(repeat_part)-1))
        return result

    return frac(S) == frac(T)
```

delete-columns-to-make-sorted-ii.py

```
# DESC
# Example 1:
# We are given an array A of N lowercase letter strings, all of the same length.
# Suppose we chose a set of deletion indices D such that after deletions, the fina
# l array has its elements in lexicographic order (A[0] <= A[1] <= A[2] ... <= A[A
# .length - 1]).
# Example 2:
# Example 3:
# Return the minimum possible value of D.length.
# Note:
# Now, we may choose any set of deletion indices, and for each string, we delete a
# ll the characters in those indices.
# For example, if we have an array A = ["abcdef","uvwxyz"] and deletion indices {0
# , 2, 3}, then the final array after deletions is ["bef","vyz"].

# NOTE
# 1 <= A.length <= 100
# 1 <= A[i].length <= 100

# EXAMPLE
# Input: ["ca","bb","ac"]
# Output: 1
# Explanation:
# After deleting the first column,
#  A = ["a", "b", "c"].
# Now A is in lexicographic order (ie. A[0] <= A[1] <= A[2])
# .
# We require at least 1 deletion since initially A was not in lexicographic orde
# r, so the answer is 1.
# Input: ["zyx","wvu","tsr"]
# Output: 3
# Explanation:
# We have to delete every column.
# Input: ["xc","yb","za"]
# Output: 0
# Explanation:
# A is already in lexicographic or
# der, so we don't need to delete anything.
# Note that the rows of A are not necess
# arily in lexicographic order:
# ie. it is NOT necessarily true that (A[0][0] <= A[
# 0][1] <= ...)

# 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
```

inorder-successor-in-bst.py

```
# inorder-successor-in-bst is not found.
# Time:  O(h)
# Space: O(1)

class Solution(object):
    def inorderSuccessor(self, root, p):
        """
        :type root: TreeNode
        :type p: TreeNode
        :rtype: TreeNode
        """
        # If it has right subtree.
        if p and p.right:
            p = p.right
            while p.left:
                p = p.left
            return p

        # Search from root.
        successor = None
        while root and root != p:
            if root.val > p.val:
                successor = root
                root = root.left
            else:
                root = root.right

        return successor
```

kth-ancestor-of-a-tree-node.py

```python
# kth-ancestor-of-a-tree-node is not found.
# Time:  ctor: O(n * logh)
#        get:  O(logh)
# Space: O(n * logh)

# binary jump solution (frequently used in competitive programming)
# Template:
# https://github.com/kamyu104/FacebookHackerCup-2019/blob/master/Final%20Round/little_boat_on_the_sea.py
class TreeAncestor(object):

    def __init__(self, n, parent):
        """
        :type n: int
        :type parent: List[int]
        """
        par = [[p] if p != -1 else [] for p in parent]
        q = [par[i] for i, p in enumerate(parent) if p != -1]
        i = 0
        while q:
            new_q = []
            for p in q:
                if not (i < len(par[p[i]])):
                    continue
                p.append(par[p[i]][i])
                new_q.append(p)
            q = new_q
            i += 1
        self.__parent = par

    def getKthAncestor(self, node, k):
        """
        :type node: int
        :type k: int
        :rtype: int
        """
        par, i, pow_i_of_2 = self.__parent, 0, 1
        while pow_i_of_2 <= k:
            if k & pow_i_of_2:
                if not (i < len(par[node])):
                    return -1
                node = par[node][i]
            i += 1
            pow_i_of_2 *= 2
        return node
```

evaluate-reverse-polish-notation.py

```
# DESC
# Example 3:
# Example 2:
# Evaluate the value of an arithmetic expression in Reverse Polish Notation.
# Example 1:
# Note:
# 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
# Input: ["2", "1", "+", "3", "*"]
# Output: 9
# Explanation: ((2 + 1) * 3) = 9
# Input: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
# Out
# put: 22
# Explanation:
#   ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
# = ((10 * (6 / (1
# 2 * -11))) + 17) + 5
# = ((10 * (6 / -132)) + 17) + 5
# = ((10 * 0) + 17) + 5
# = (0 +
#  17) + 5
# = 17 + 5
# = 22
# Input: ["4", "13", "5", "/", "+"]
# Output: 6
# Explanation: (4 + (13 / 5)) = 6

# 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()
```

maximize-distance-to-closest-person.py

```
# DESC
# Example 1:
# Return that maximum distance to closest person.
# Alex wants to sit in the seat such that the distance between him and the closest
#  person to him is maximized.
# In a row of seats, 1 represents a person sitting in that seat, and 0 represents
# that the seat is empty.
# Example 2:
# Constraints:
# There is at least one empty seat, and at least one person sitting.

# NOTE
# 2 <= seats.length <= 20000
# seats contains only 0s or 1s, at least one 0, and at least one 1.

# EXAMPLE
# Input: [1,0,0,0,1,0,1]
# Output: 2
# Explanation:
# If Alex sits in the second open s
# eat (seats[2]), then the closest person has distance 2.
# If Alex sits in any othe
# r open seat, the closest person has distance 1.
# Thus, the maximum distance to th
# e closest person is 2.
# Input: [1,0,0,0]
# Output: 3
# Explanation:
# If Alex sits in the last seat, the clos
# est person is 3 seats away.
# This is the maximum distance possible, so the answer
#  is 3.

# Time:  O(n)
# Space: O(1)


class Solution(object):
    def maxDistToClosest(self, seats):
        """
        :type seats: List[int]
        :rtype: int
        """
        prev, result = -1, 1
        for i in xrange(len(seats)):
            if seats[i]:
                if prev < 0:
                    result = i
                else:
                    result = max(result, (i-prev)//2)
                prev = i
        return max(result, len(seats)-1-prev)
```

wiggle-sort.py

```
# wiggle-sort is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def wiggleSort(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        for i in xrange(1, len(nums)):
            if ((i % 2) and nums[i - 1] > nums[i]) or \
                (not (i % 2) and nums[i - 1] < nums[i]):
                # Swap unordered elements.
                nums[i - 1], nums[i] = nums[i], nums[i - 1]


# time: O(nlogn)
# space: O(n)
class Solution2(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]
```

increasing-decreasing-string.py

```python
# increasing-decreasing-string is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def sortString(self, s):
        """
        :type s: str
        :rtype: str
        """
        result, count = [], [0]*26
        for c in s:
            count[ord(c)-ord('a')] += 1
        while len(result) != len(s):
            for c in xrange(len(count)):
                if not count[c]:
                    continue
                result.append(chr(ord('a')+c))
                count[c] -= 1
            for c in reversed(xrange(len(count))):
                if not count[c]:
                    continue
                result.append(chr(ord('a')+c))
                count[c] -= 1
        return "".join(result)


# Time:  O(n)
# Space: O(1)
import collections


class Solution2(object):
    def sortString(self, s):
        """
        :type s: str
        :rtype: str
        """
        result, count, desc = [], collections.Counter(s), False
        while count:
            for c in sorted(count.keys(), reverse=desc):
                result.append(c)
                count[c] -= 1
                if not count[c]:
                    del count[c]
            desc = not desc
        return "".join(result)
```

validate-ip-address.py

```
# DESC
# Example 1:
# Besides, leading zeros in the IPv4 is invalid. For example, the address 172.16.2
# 54.01 is invalid.
# IPv4 addresses are canonically represented in dot-decimal notation, which consis
# ts of four decimal numbers, each ranging from 0 to 255, separated by dots ("."),
#  e.g.,172.16.254.1;
# Example 2:
# However, we don't replace a consecutive group of zero value with a single empty
# group using two consecutive colons (::) to pursue simplicity. For example, 2001:
# 0db8:85a3::8A2E:0370:7334 is an invalid IPv6 address.
# IPv6 addresses are represented as eight groups of four hexadecimal digits, each
# group representing 16 bits. The groups are separated by colons (":"). For exampl
# e, the address 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a valid one. Also, we
# could omit some leading zeros among four hexadecimal digits and some low-case ch
# aracters in the address to upper-case ones, so 2001:db8:85a3:0:0:8A2E:0370:7334
# is also a valid IPv6 address(Omit leading zeros and using upper cases).
# 172.16.254.1
# Write a function to check whether an input string is a valid IPv4 address or IPv
# 6 address or neither.
# Besides, extra leading zeros in the IPv6 is also invalid. For example, the addre
# ss 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is invalid.
# Constraints:
# Example 3:

# NOTE
# IP consists only of English letters, digits and the characters "." and ":".

# EXAMPLE
# Input: IP = "172.16.254.1"
# Output: "IPv4"
# Explanation: This is a valid IPv4 addr
# ess, return "IPv4".
# Input: IP = "256.256.256.256"
# Output: "Neither"
# Explanation: This is neither a I
# Pv4 address nor a IPv6 address.
# Input: IP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
# Output: "IPv6"
# Explanation: This
#  is a valid IPv6 address, return "IPv6".

# 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"
```

valid-palindrome.py

```python
# DESC
# Note: For the purpose of this problem, we define empty string as valid palindrome.
# Example 1:
# Constraints:
# Given a string, determine if it is a palindrome, considering only alphanumeric c
# haracters and ignoring cases.
# Example 2:

# NOTE
# s consists only of printable ASCII characters.

# EXAMPLE
# Input: "A man, a plan, a canal: Panama"
# Output: true
# Input: "race a car"
# Output: false

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param s, a string
    # @return a boolean
    def isPalindrome(self, s):
        i, j = 0, len(s) - 1
        while i < j:
            while i < j and not s[i].isalnum():
                i += 1
            while i < j and not s[j].isalnum():
                j -= 1
            if s[i].lower() != s[j].lower():
                return False
            i, j = i + 1, j - 1
        return True
```

maximum-students-taking-exam.py

```
# maximum-students-taking-exam is not found.
# Time:  O(m * n * sqrt(m * n))
# Space: O(m * n)

# the problem is the same as google codejam 2008 round 3 problem C
# https://github.com/kamyu104/GoogleCodeJam-2008/blob/master/Round%203/no_cheating.py

import collections


# Time:  O(E * sqrt(V))
# Space: O(V)
# Source code from http://code.activestate.com/recipes/123641-hopcroft-karp-bipartite-matching/
# Hopcroft-Karp bipartite max-cardinality matching and max independent set
# David Eppstein, UC Irvine, 27 Apr 2002
def bipartiteMatch(graph):
    '''Find maximum cardinality matching of a bipartite graph (U,V,E).
        The input format is a dictionary mapping members of U to a list
        of their neighbors in V.  The output is a triple (M,A,B) where M is a
        dictionary mapping members of V to their matches in U, A is the part
        of the maximum independent set in U, and B is the part of the MIS in V.
        The same object may occur in both U and V, and is treated as two
        distinct vertices if this happens.'''

    # initialize greedy matching (redundant, but faster than full search)
    matching = {}
    for u in graph:
        for v in graph[u]:
            if v not in matching:
                matching[v] = u
                break

    while 1:
        # structure residual graph into layers
        # pred[u] gives the neighbor in the previous layer for u in U
        # preds[v] gives a list of neighbors in the previous layer for v in V
        # unmatched gives a list of unmatched vertices in final layer of V,
        # and is also used as a flag value for pred[u] when u is in the first layer
        preds = {}
        unmatched = []
        pred = dict([(u,unmatched) for u in graph])
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        # repeatedly extend layering structure by another pair of layers
        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in graph[u]:
                    if v not in preds:
                        newLayer.setdefault(v,[]).append(u)
            layer = []
            for v in newLayer:
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
```

```
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in graph:
                for v in graph[u]:
                    if v not in preds:
                        unlayered[v] = None
            return (matching,list(pred),list(unlayered))

        # recursively search backward through layers to find alternating paths
        # recursion returns true if found path, false otherwise
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
                            matching[v] = u
                            return 1
            return 0

        for v in unmatched: recurse(v)


# Hopcroft-Karp bipartite matching
class Solution(object):
    def maxStudents(self, seats):
        """
        :type seats: List[List[str]]
        :rtype: int
        """
        directions = [(-1, -1), (0, -1), (1, -1), (-1, 1), (0, 1), (1, 1)]
        E, count = collections.defaultdict(list), 0
        for i in xrange(len(seats)):
            for j in xrange(len(seats[0])):
                if seats[i][j] != '.':
                    continue
                count += 1
                if j%2:
                    continue
                for dx, dy in directions:
                    ni, nj = i+dx, j+dy
                    if 0 <= ni < len(seats) and \
                       0 <= nj < len(seats[0]) and \
                       seats[ni][nj] == '.':
                        E[i*len(seats[0])+j].append(ni*len(seats[0])+nj)
        return count-len(bipartiteMatch(E)[0])


# Time:  O(|V| * |E|) = O(m^2 * n^2)
# Space: O(|V| + |E|) = O(m * n)
# Hungarian bipartite matching
class Solution2(object):
    def maxStudents(self, seats):
```

```
        """
        :type seats: List[List[str]]
        :rtype: int
        """
        directions = [(-1, -1), (0, -1), (1, -1), (-1, 1), (0, 1), (1, 1)]
        def dfs(seats, e, lookup, matching):
            i, j = e
            for dx, dy in directions:
                ni, nj = i+dx, j+dy
                if 0 <= ni < len(seats) and 0 <= nj < len(seats[0]) and \
                    seats[ni][nj] == '.' and not lookup[ni][nj]:
                    lookup[ni][nj] = True
                    if matching[ni][nj] == -1 or dfs(seats, matching[ni][nj], lookup, matching):
                        matching[ni][nj] = e
                        return True
            return False

        def Hungarian(seats):
            result = 0
            matching = [[-1]*len(seats[0]) for _ in xrange(len(seats))]
            for i in xrange(len(seats)):
                for j in xrange(0, len(seats[0]), 2):
                    if seats[i][j] != '.':
                        continue
                    lookup = [[False]*len(seats[0]) for _ in xrange(len(seats))]
                    if dfs(seats, (i, j), lookup, matching):
                        result += 1
            return result

        count = 0
        for i in xrange(len(seats)):
            for j in xrange(len(seats[0])):
                if seats[i][j] == '.':
                    count += 1
        return count-Hungarian(seats)


# Time:  O(m * 2^n * 2^n) = O(m * 4^n)
# Space: O(2^n)
# dp solution
class Solution3(object):
    def maxStudents(self, seats):
        """
        :type seats: List[List[str]]
        :rtype: int
        """
        def popcount(n):
            result = 0
            while n:
                n &= n - 1
                result += 1
            return result

        dp = {0: 0}
        for row in seats:
            invalid_mask = sum(1 << c for c, v in enumerate(row) if v == '#')
            new_dp = {}
            for mask1, v1 in dp.iteritems():
                for mask2 in xrange(1 << len(seats[0])):
                    if (mask2 & invalid_mask) or \
```

```
                (mask2 & (mask1 << 1)) or (mask2 & (mask1 >> 1)) or \
                (mask2 & (mask2 << 1)) or (mask2 & (mask2 >> 1)):
                 continue
            new_dp[mask2] = max(new_dp.get(mask2, 0), v1+popcount(mask2))
    dp = new_dp
return max(dp.itervalues()) if dp else 0
```

merge-k-sorted-lists.py

```
# DESC
# Example 3:
# Given an array of linked-lists lists, each linked list is sorted in ascending order.
# Example 2:
# Example 1:
# Constraints:
# Merge all the linked-lists into one sort linked-list and return it.

# NOTE
# -10^4 <= lists[i][j] <= 10^4
# k == lists.length
# lists[i] is sorted in ascending order.
# 0 <= k <= 10^4
# 0 <= lists[i].length <= 500
# The sum of lists[i].length won't exceed 10^4.

# EXAMPLE
# Input: lists = []
# Output: []
# Input: lists = [[1,4,5],[1,3,4],[2,6]]
# Output: [1,1,2,3,4,4,5,6]
# Explanation: Th
# e linked-lists are:
# [
#   1->4->5,
#   1->3->4,
#   2->6
# ]
# merging them into one sorte
# d list:
# 1->1->2->3->4->4->5->6
# Input: lists = [[]]
# Output: []

# Time:  O(nlogk)
# 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)


# Merge two by two solution.
class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        def mergeTwoLists(l1, l2):
            curr = dummy = ListNode(0)
            while l1 and l2:
                if l1.val < l2.val:
```

```
                    curr.next = l1
                    l1 = l1.next
                else:
                    curr.next = l2
                    l2 = l2.next
                curr = curr.next
            curr.next = l1 or l2
            return dummy.next

        if not lists:
            return None
        left, right = 0, len(lists) - 1
        while right > 0:
            if left >= right:
                left = 0
            else:
                lists[left] = mergeTwoLists(lists[left], lists[right])
                left += 1
                right -= 1
        return lists[0]


# Time:  O(nlogk)
# Space: O(logk)
# Divide and Conquer solution.
class Solution2(object):
    # @param a list of ListNode
    # @return a ListNode
    def mergeKLists(self, lists):
        def mergeTwoLists(l1, l2):
            curr = dummy = ListNode(0)
            while l1 and l2:
                if l1.val < l2.val:
                    curr.next = l1
                    l1 = l1.next
                else:
                    curr.next = l2
                    l2 = l2.next
                curr = curr.next
            curr.next = l1 or l2
            return dummy.next

        def mergeKListsHelper(lists, begin, end):
            if begin > end:
                return None
            if begin == end:
                return lists[begin]
            return mergeTwoLists(mergeKListsHelper(lists, begin, (begin + end) / 2), \
                                 mergeKListsHelper(lists, (begin + end) / 2 + 1, end))

        return mergeKListsHelper(lists, 0, len(lists) - 1)


# Time:  O(nlogk)
# Space: O(k)
# Heap solution.
import heapq
class Solution3(object):
    # @param a list of ListNode
    # @return a ListNode
```

```
def mergeKLists(self, lists):
    dummy = ListNode(0)
    current = dummy

    heap = []
    for sorted_list in lists:
        if sorted_list:
            heapq.heappush(heap, (sorted_list.val, sorted_list))

    while heap:
        smallest = heapq.heappop(heap)[1]
        current.next = smallest
        current = current.next
        if smallest.next:
            heapq.heappush(heap, (smallest.next.val, smallest.next))

    return dummy.next
```

restore-the-array.py

```python
# restore-the-arra is not found.
# Time:  O(nlogk)
# Space: O(logk)

class Solution(object):
    def numberOfArrays(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        MOD = 10**9 + 7
        klen = len(str(k))
        dp = [0]*(klen+1)
        dp[len(s)%len(dp)] = 1
        for i in reversed(xrange(len(s))):
            dp[i%len(dp)] = 0
            if s[i] == '0':
                continue
            curr = 0
            for j in xrange(i, min(i+klen, len(s))):
                curr = 10*curr + int(s[j])
                if curr > k:
                    break
                dp[i%len(dp)] = (dp[i%len(dp)] + dp[(j+1)%len(dp)])%MOD
        return dp[0]
```

sort-array-by-parity.py

```python
# sort-array-by-parit is not found.
# Time:  O(n)
# Space: O(1)

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

single-number-iii.py

```
# DESC
# Example:
# Note:
# Given an array of numbers nums, in which exactly two elements appear only once a
# nd all the other elements appear exactly twice. Find the two elements that appea
# r only once.

# NOTE
# The order of the result is not important. So in the above example, [5, 3] is als
# o correct.
# Your algorithm should run in linear runtime complexity. Could you implement it u
# sing only constant space complexity?

# EXAMPLE
# Input:  [1,2,1,3,2,5]
# Output: [3,5]

# 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]]
```

cracking-the-safe.py

```
# DESC
# While entering a password, the last n digits entered will automatically be match
# ed against the correct password.
# Return any password of minimum length that is guaranteed to open the box at some
#  point of entering it.
# Example 1:
# For example, assuming the correct password is "345", if you type "012345", the b
# ox will open because the correct password matches the suffix of the entered pass
# word.
# There is a box protected by a password. The password is a sequence of n digits w
# here each digit can be one of the first k digits 0, 1, ..., k-1.
# Note:
# Example 2:

# NOTE
# k^n will be at most 4096.
# k will be in the range [1, 10].
# n will be in the range [1, 4].

# EXAMPLE
# Input: n = 2, k = 2
# Output: "00110"
# Note: "01100", "10011", "11001" will be acce
# pted too.
# Input: n = 1, k = 2
# Output: "01"
# Note: "10" will be accepted too.

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

class Solution(object):
    def crackSafe(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        M = k**(n-1)
        P = [q*k+i for i in xrange(k) for q in xrange(M)]  # rotate: i*k^(n-1) + q => q*k + i
        result = [str(k-1)]*(n-1)
        for i in xrange(k**n):
            j = i
            # concatenation in lexicographic order of Lyndon words
            while P[j] >= 0:
                result.append(str(j//M))
                P[j], j = -1, P[j]
        return "".join(result)


# Time:  O(n * k^n)
# Space: O(n * k^n)
class Solution2(object):
    def crackSafe(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
```

```
        """
        result = [str(k-1)]*n
        lookup = {"".join(result)}
        total = k**n
        while len(lookup) < total:
            node = result[len(result)-n+1:]
            for i in xrange(k):
                neighbor = "".join(node) + str(i)
                if neighbor not in lookup:
                    lookup.add(neighbor)
                    result.append(str(i))
                    break
        return "".join(result)


# Time:  O(n * k^n)
# Space: O(n * k^n)
class Solution3(object):
    def crackSafe(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        def dfs(k, node, lookup, result):
            for i in xrange(k):
                neighbor = node + str(i)
                if neighbor not in lookup:
                    lookup.add(neighbor)
                    result.append(str(i))
                    dfs(k, neighbor[1:], lookup, result)
                    break

        result = [str(k-1)]*(n-1)
        lookup = set()
        dfs(k, "".join(result), lookup, result)
        return "".join(result)
```

binary-tree-cameras.py

```
# DESC
# Note:
# Each camera at a node can monitor its parent, itself, and its immediate children.
# Example 2:
# Example 1:
# Calculate the minimum number of cameras needed to monitor all nodes of the tree.
# Given a binary tree, we install cameras on the nodes of the tree.

# NOTE
# Every node has value 0.
# The number of nodes in the given tree will be in the range [1, 1000].

# EXAMPLE
# Input: [0,0,null,0,0]
# Output: 1
# Explanation: One camera is enough to monitor all
#  nodes if placed as shown.
# Input: [0,0,null,0,null,0,null,null,0]
# Output: 2
# Explanation: At least two camer
# as are needed to monitor all nodes of the tree. The above image shows one of the
#  valid configurations of camera placement.

# 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 minCameraCover(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        UNCOVERED, COVERED, CAMERA = range(3)
        def dfs(root, result):
            left = dfs(root.left, result) if root.left else COVERED
            right = dfs(root.right, result) if root.right else COVERED
            if left == UNCOVERED or right == UNCOVERED:
                result[0] += 1
                return CAMERA
            if left == CAMERA or right == CAMERA:
                return COVERED
            return UNCOVERED

        result = [0]
        if dfs(root, result) == UNCOVERED:
            result[0] += 1
        return result[0]
```

shuffle-the-array.py

```
# shuffle-the-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def shuffle(self, nums, n):
        """
        :type nums: List[int]
        :type n: int
        :rtype: List[int]
        """
        def dest(i, n):
            return 2*i if i < n else 2*(i-n)+1

        for i in xrange(len(nums)):
            if nums[i] < 0:
                continue
            j = i
            while True:
                j = dest(j, n)
                nums[i], nums[j] = nums[j], nums[i]
                nums[j] = -nums[j]
                if i == j:
                    break
        for i in xrange(len(nums)):
            nums[i] = -nums[i]
        return nums
```

diameter-of-binary-tree.py

```
# DESC
# Given a binary tree, you need to compute the length of the diameter of the tree.
#  The diameter of a binary tree is the length of the longest path between any two
#  nodes in a tree. This path may or may not pass through the root.
# Note:
# The length of path between two nodes is represented by the number of edges
#  between them.
# Example:
#
# Given a binary tree
# Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].

# NOTE
#

# EXAMPLE
# 1
#          / \
#         2   3
#        / \
#       4   5

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

class Solution(object):
    def diameterOfBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        return self.depth(root, 0)[1]

    def depth(self, root, diameter):
        if not root:
            return 0, diameter
        left, diameter = self.depth(root.left, diameter)
        right, diameter = self.depth(root.right, diameter)
        return 1 + max(left, right), max(diameter, left + right)
```

read-n-characters-given-read4-ii-call-multiple-times.py

```python
# read-n-characters-given-read4-ii-call-multiple-times is not found.
# Time:  O(n)
# Space: O(1)

def read4(buf):
    global file_content
    i = 0
    while i < len(file_content) and i < 4:
        buf[i] = file_content[i]
        i += 1

    if len(file_content) > 4:
        file_content = file_content[4:]
    else:
        file_content = ""
    return i

# The read4 API is already defined for you.
# @param buf, a list of characters
# @return an integer
# def read4(buf):

class Solution(object):
    def __init__(self):
        self.__buf4 = [''] * 4
        self.__i4 = 0
        self.__n4 = 0

    def read(self, buf, n):
        """
        :type buf: Destination buffer (List[str])
        :type n: Maximum number of characters to read (int)
        :rtype: The number of characters read (int)
        """
        i = 0
        while i < n:
            if self.__i4 < self.__n4:  # Any characters in buf4.
                buf[i] = self.__buf4[self.__i4]
                i += 1
                self.__i4 += 1
            else:
                self.__n4 = read4(self.__buf4)  # Read more characters.
                if self.__n4:
                    self.__i4 = 0
                else:  # Buffer has been empty.
                    break

        return i
```

add-binary.py

```
# add-binar is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param a, a string
    # @param b, a string
    # @return a string
    def addBinary(self, a, b):
        result, 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 = divmod(val, 2)
            result += str(val)
        if carry:
            result += str(carry)
        return result[::-1]


# Time:  O(n)
# Space: O(1)
from itertools import izip_longest


class Solution2(object):
    def addBinary(self, a, b):
        """
        :type a: str
        :type b: str
        :rtype: str
        """
        result = ""
        carry = 0
        for x, y in izip_longest(reversed(a), reversed(b), fillvalue="0"):
            carry, remainder = divmod(int(x)+int(y)+carry, 2)
            result += str(remainder)

        if carry:
            result += str(carry)

        return result[::-1]
```

four-divisors.py

```python
# four-divisors is not found.
# 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
```

maximum-sum-of-3-non-overlapping-subarrays.py

```
# DESC
# Return the result as a list of indices representing the starting position of eac
# h interval (0-indexed). If there are multiple answers, return the lexicographica
# lly smallest one.
# Note:
# Each subarray will be of size k, and we want to maximize the sum of all 3*k entries.
# In a given array nums of positive integers, find three non-overlapping subarrays
#  with maximum sum.
# Example:

# NOTE
# nums[i] will be between 1 and 65535.
# nums.length will be between 1 and 20000.
# k will be between 1 and floor(nums.length / 3).

# EXAMPLE
# Input: [1,2,1,2,6,7,5,1], 2
# Output: [0, 3, 5]
# Explanation: Subarrays [1, 2], [2,
#  6], [7, 5] correspond to the starting indices [0, 3, 5].
# We could have also tak
# en [2, 1], but an answer of [1, 3, 5] would be lexicographically larger.

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

class Solution(object):
    def maxSumOfThreeSubarrays(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        n = len(nums)
        accu = [0]
        for num in nums:
            accu.append(accu[-1]+num)

        left_pos = [0] * n
        total = accu[k]-accu[0]
        for i in xrange(k, n):
            if accu[i+1]-accu[i+1-k] > total:
                left_pos[i] = i+1-k
                total = accu[i+1]-accu[i+1-k]
            else:
                left_pos[i] = left_pos[i-1]

        right_pos = [n-k] * n
        total = accu[n]-accu[n-k]
        for i in reversed(xrange(n-k)):
            if accu[i+k]-accu[i] > total:
                right_pos[i] = i
                total = accu[i+k]-accu[i]
            else:
                right_pos[i] = right_pos[i+1]

        result, max_sum = [], 0
        for i in xrange(k, n-2*k+1):
```

```
    left, right = left_pos[i-1], right_pos[i+k]
    total = (accu[i+k]-accu[i]) + \
            (accu[left+k]-accu[left]) + \
            (accu[right+k]-accu[right])
    if total > max_sum:
        max_sum = total
        result = [left, i, right]
return result
```

next-permutation.py

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

# NOTE
#

# EXAMPLE
#

# 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]
```

self-crossing.py

```
# DESC
# Example 1:
# Write a one-pass algorithm with O(1) extra space to determine, if your path cros
# ses itself, or not.
# Example 3:
# Example 2:
# You are given an array x of n positive numbers. You start at point (0,0) and mov
# es x[0] metres to the north, then x[1] metres to the west, x[2] metres to the so
# uth, x[3] metres to the east and so on. In other words, after each move your dir
# ection changes counter-clockwise.

# NOTE
#

# EXAMPLE
#
#
#
#
# >
#
# Input: [1,2,3,4]
# Output: false
#
#
# >
#
# Input: [1,1,1,1]
# Output: true
#
#
# >
#
#
# Input: [2,1,1,2]
# Output: true

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isSelfCrossing(self, x):
        """
        :type x: List[int]
        :rtype: bool
        """
        if len(x) >= 5 and x[3] == x[1] and x[4] + x[0] >= x[2]:
            # Crossing in a loop:
            #     2
            # 3
            #   >1
            #   4  0  (overlapped)
            return True

        for i in xrange(3, len(x)):
            if x[i] >= x[i - 2] and x[i - 3] >= x[i - 1]:
                # Case 1:
                #    i-2
```

```
            # i-1
            #    >i
            #     i-3
            return True
        elif i >= 5 and x[i - 4] <= x[i - 2] and x[i] + x[i - 4] >= x[i - 2] and \
                        x[i - 1] <= x[i - 3] and x[i - 5] + x[i - 1] >= x[i - 3]:
            # Case 2:
            #    i-4
            #
            #    i<
            # i-3 i-5i-1
            #
            #      i-2
            return True
    return False
```

rank-transform-of-an-array.py

```python
# rank-transform-of-an-arra is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def arrayRankTransform(self, arr):
        """
        :type arr: List[int]
        :rtype: List[int]
        """
        return map({x: i+1 for i, x in enumerate(sorted(set(arr)))}.get, arr)
```

minimum-moves-to-move-a-box-to-their-target-location.py

```
# minimum-moves-to-move-a-box-to-their-target-location is not found.
# Time:  O(m^2 * n^2)
# Space: O(m^2 * n^2)

# A* Search Algorithm without heap
class Solution(object):
    def minPushBox(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        def dot(a, b):
            return a[0]*b[0]+a[1]*b[1]

        def can_reach(grid, b, p, t):
            closer, detour = [p], []
            lookup = set([b])
            while closer or detour:
                if not closer:
                    closer, detour = detour, closer
                p = closer.pop()
                if p == t:
                    return True
                if p in lookup:
                    continue
                lookup.add(p)
                for dx, dy in directions:
                    np = (p[0]+dx, p[1]+dy)
                    if not (0 <= np[0] < len(grid) and 0 <= np[1] < len(grid[0]) and
                       grid[np[0]][np[1]] != '#' and np not in lookup):
                        continue
                    (closer if dot((dx, dy), (t[0]-p[0], t[1]-p[1])) > 0 else detour).append(np)
            return False

        def g(a, b):
            return abs(a[0]-b[0])+abs(a[1]-b[1])

        def a_star(grid, b, p, t):
            f, dh = g(b, t), 2
            closer, detour = [(b, p)], []
            lookup = set()
            while closer or detour:
                if not closer:
                    f += dh
                    closer, detour = detour, closer
                b, p = closer.pop()
                if b == t:
                    return f
                if (b, p) in lookup:
                    continue
                lookup.add((b, p))
                for dx, dy in directions:
                    nb, np = (b[0]+dx, b[1]+dy), (b[0]-dx, b[1]-dy)
                    if not (0 <= nb[0] < len(grid) and 0 <= nb[1] < len(grid[0]) and
                            0 <= np[0] < len(grid) and 0 <= np[1] < len(grid[0]) and
                            grid[nb[0]][nb[1]] != '#' and grid[np[0]][np[1]] != '#' and
                            (nb, b) not in lookup and can_reach(grid, b, p, np)):
```

```
                continue
            (closer if dot((dx, dy), (t[0]-b[0], t[1]-b[1])) > 0 else detour).append((nb, b))
    return -1

b, p, t = None, None, None
for i in xrange(len(grid)):
    for j in xrange(len(grid[0])):
        if grid[i][j] == 'B':
            b = (i, j)
        elif grid[i][j] == 'S':
            p = (i, j)
        elif grid[i][j] == 'T':
            t = (i, j)
return a_star(grid, b, p, t)
```

longest-string-chain.py

```
# DESC
# A word chain is a sequence of words [word_1, word_2, ..., word_k] with k >= 1, w
# here word_1 is a predecessor of word_2, word_2 is a predecessor of word_3, and s
# o on.
# Given a list of words, each word consists of English lowercase letters.
# Example 1:
# Note:
# Return the longest possible length of a word chain with words chosen from the gi
# ven list of words.
# 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 pre
# decessor of "abac".
# word1

# NOTE
# words[i] only consists of English lowercase letters.
# 1 <= words.length <= 1000
# 1 <= words[i].length <= 16

# EXAMPLE
# Input: ["a","b","ba","bca","bda","bdca"]
# Output: 4
# Explanation: one of the longe
# st word chain is "a","ba","bda","bdca".

# 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

```
# DESC
# Return the minimum size of the set so that at least half of the integers of the
# array are removed.
# Constraints:
# Given an array arr.  You can choose a set of integers and remove all the occurre
# nces of these integers in the array.
# Example 5:
# Example 1:
# Example 2:
# Example 4:
# Example 3:

# NOTE
# 1 <= arr.length <= 10^5
# 1 <= arr[i] <= 10^5
# arr.length is even.

# EXAMPLE
# Input: arr = [1,2,3,4,5,6,7,8,9,10]
# Output: 5
# Input: arr = [1,9]
# Output: 1
# Input: arr = [3,3,3,3,5,5,5,2,2,7]
# Output: 2
# Explanation: Choosing {3,7} will ma
# ke 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 g
# reater than half of the size of the old array.
# Input: arr = [1000,1000,3,7]
# Output: 1
# 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.

# 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
```

all-paths-from-source-to-target.py

```python
# DESC
# Given a directed acyclic graph of N nodes. Find all possible paths from node 0 t
# o node N-1, and return them in any order.
# Constraints:
# The graph is given as follows:  the nodes are 0, 1, ..., graph.length - 1.  grap
# h[i] is a list of all nodes j for which the edge (i, j) exists.

# NOTE
# The number of nodes in the graph will be in the range [2, 15].
# You can print different paths in any order, but you should keep the order of nod
# es inside one path.

# EXAMPLE
# Example:
# Input: [[1,2],[3],[3],[]]
# Output: [[0,1,3],[0,2,3]]
# Explanation: The gr
# aph looks like this:
# 0--->1
# |    |
# v    v
# 2--->3
# There are two paths: 0 -> 1 ->
# 3 and 0 -> 2 -> 3.

# Time:  O(p + r * n), p is the count of all the possible paths in graph,
#                      r is the count of the result.
# Space: O(n)

class Solution(object):
    def allPathsSourceTarget(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: List[List[int]]
        """
        def dfs(graph, curr, path, result):
            if curr == len(graph)-1:
                result.append(path[:])
                return
            for node in graph[curr]:
                path.append(node)
                dfs(graph, node, path, result)
                path.pop()

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

two-city-scheduling.py

```
# DESC
# Example 1:
# Note:
# Return the minimum cost to fly every person to a city such that exactly N people
#  arrive in each city.
# There are 2N people a company is planning to interview. The cost of flying the i
# -th person to city A is costs[i][0], and the cost of flying the i-th person to c
# ity B is costs[i][1].

# NOTE
# 1 <= costs[i][0], costs[i][1] <= 1000
# 1 <= costs.length <= 100
# It is guaranteed that costs.length is even.

# EXAMPLE
# Input: [[10,20],[30,200],[400,50],[30,20]]
# Output: 110
# Explanation:
# The first p
# erson goes to city A for a cost of 10.
# The second person goes to city A for a co
# st of 30.
# The third person goes to city B for a cost of 50.
# The fourth person go
# es to city B for a cost of 20.
#
# The total minimum cost is 10 + 30 + 50 + 20 = 11
# 0 to have half the people interviewing in each city.

# Time:  O(n) ~ O(n^2), O(n) on average.
# Space: O(1)

import random


# quick select solution
class Solution(object):
    def twoCitySchedCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        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:
                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

kthElement(costs, len(costs)//2, lambda a, b: a[0]-a[1] < b[0]-b[1])
result = 0
for i in xrange(len(costs)):
    result += costs[i][0] if i < len(costs)//2 else costs[i][1]
return result
```

construct-target-array-with-multiple-sums.py

```
# DESC
# Given an array of integers target. From a starting array, A consisting of all 1'
# s, you may perform the following procedure :
# Example 1:
# Return True if it is possible to construct the target array from A otherwise ret
# urn False.
# target
# Example 3:
# Example 2:
# Constraints:

# NOTE
# let x be the sum of all elements currently in your array.
# 1 <= target.length <= 5 * 10^4
# N == target.length
# You may repeat this procedure as many times as needed.
# 1 <= target[i] <= 10^9
# choose index i, such that 0 <= i < target.size and set the value of A at index i
#  to x.

# EXAMPLE
# Input: target = [1,1,1,2]
# Output: false
# Explanation: Impossible to create target
#  array from [1,1,1,1].
# Input: target = [8,5]
# Output: true
# Input: target = [9,3,5]
# Output: true
# Explanation: Start with [1, 1, 1]
# [1, 1, 1
# ], sum = 3 choose index 1
# [1, 3, 1], sum = 5 choose index 2
# [1, 3, 5], sum = 9 c
# hoose index 0
# [9, 3, 5] Done

# Time:  O(log(max(t)) * logn)
# Space: O(n)

import heapq


class Solution(object):
    def isPossible(self, target):
        """
        :type target: List[int]
        :rtype: bool
        """
        # (1) x + remain = y
        # (2) y + remain = total
        # (1) - (2) => x - y = y - total
        #           => x = 2*y - total
        total = sum(target)
        max_heap = [-x for x in target]
        heapq.heapify(max_heap)
        while total != len(target):
            y = -heapq.heappop(max_heap)
```

```
        remain = total-y
        x = y-remain
        if x <= 0:
            return False
        if x > remain:  # for case [1, 1000000000]
            x = x%remain + remain
        heapq.heappush(max_heap, -x)
        total = x+remain
    return True
```

ugly-number-ii.py

```
# DESC
# Write a program to find the n-th ugly number.
# Example:
# Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.
# 2, 3, 5
# Note:

# NOTE
# n does not exceed 1690.
# 1 is typically treated as an ugly number.

# EXAMPLE
# Input: n = 10
# Output: 12
# Explanation: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequ
# ence of the first 10 ugly numbers.

# 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
            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

```
# DESC
# Example:
# Design your implementation of the circular double-ended queue (deque).
# Your implementation should support following operations:
# Note:

# NOTE
# isEmpty(): Checks whether Deque is empty or not.
# getFront(): Gets the front item from the Deque. If the deque is empty, return -1.
# All values will be in the range of [0, 1000].
# Please do not use the built-in Deque library.
# getRear(): Gets the last item from Deque. If the deque is empty, return -1.
# 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.
# MyCircularDeque(k): Constructor, set the size of the deque to be k.
# The number of operations will be in the range of [1, 1000].
# deleteLast(): Deletes an item from the rear of Deque. Return true if the operati
# on is successful.
# deleteFront(): Deletes an item from the front of Deque. Return true if the opera
# tion is successful.
# 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.insertF
# ront(4);                         // return false, the queue is full
# circularDeque.getRear();                           // re
# turn 2
# circularDeque.isFull();                                 // return true
# circularDeque.deleteLast();                          /
# / return true
# circularDeque.insertFront(4);                          // return true
# circularDeque.getFr
# ont();                       // return 4

# 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)
```

frog-position-after-t-seconds.py

```
# frog-position-after-t-seconds is not found.
# Time:  O(n)
# Space: O(n)

import collections


# bfs solution with better precision
class Solution(object):
    def frogPosition(self, n, edges, t, target):
        """
        :type n: int
        :type edges: List[List[int]]
        :type t: int
        :type target: int
        :rtype: float
        """
        G = collections.defaultdict(list)
        for u, v in edges:
            G[u].append(v)
            G[v].append(u)

        stk = [(t, 1, 0, 1)]
        while stk:
            new_stk = []
            while stk:
                t, node, parent, choices = stk.pop()
                if not t or not (len(G[node])-(parent != 0)):
                    if node == target:
                        return 1.0/choices
                    continue
                for child in G[node]:
                    if child == parent:
                        continue
                    new_stk.append((t-1, child, node,
                                    choices*(len(G[node])-(parent != 0))))
            stk = new_stk
        return 0.0


# Time:  O(n)
# Space: O(n)
# dfs solution with stack with better precision
class Solution2(object):
    def frogPosition(self, n, edges, t, target):
        """
        :type n: int
        :type edges: List[List[int]]
        :type t: int
        :type target: int
        :rtype: float
        """
        G = collections.defaultdict(list)
        for u, v in edges:
            G[u].append(v)
            G[v].append(u)

        stk = [(t, 1, 0, 1)]
```

```
        while stk:
            t, node, parent, choices = stk.pop()
            if not t or not (len(G[node])-(parent != 0)):
                if node == target:
                    return 1.0/choices
                continue
            for child in G[node]:
                if child == parent:
                    continue
                stk.append((t-1, child, node,
                            choices*(len(G[node])-(parent != 0))))
        return 0.0


# Time:  O(n)
# Space: O(n)
# dfs solution with recursion with better precision
class Solution3(object):
    def frogPosition(self, n, edges, t, target):
        """
        :type n: int
        :type edges: List[List[int]]
        :type t: int
        :type target: int
        :rtype: float
        """
        def dfs(G, target, t, node, parent):
            if not t or not (len(G[node])-(parent != 0)):
                return int(node == target)
            result = 0
            for child in G[node]:
                if child == parent:
                    continue
                result = dfs(G, target, t-1, child, node)
                if result:
                    break
            return result*(len(G[node])-(parent != 0))

        G = collections.defaultdict(list)
        for u, v in edges:
            G[u].append(v)
            G[v].append(u)
        choices = dfs(G, target, t, 1, 0)
        return 1.0/choices if choices else 0.0


# Time:  O(n)
# Space: O(n)
# dfs solution with recursion
class Solution4(object):
    def frogPosition(self, n, edges, t, target):
        """
        :type n: int
        :type edges: List[List[int]]
        :type t: int
        :type target: int
        :rtype: float
        """
        def dfs(G, target, t, node, parent):
            if not t or not (len(G[node])-(parent != 0)):
```

```
        return float(node == target)
    for child in G[node]:
        if child == parent:
            continue
        result = dfs(G, target, t-1, child, node)
        if result:
            break
    return result/(len(G[node])-(parent != 0))

G = collections.defaultdict(list)
for u, v in edges:
    G[u].append(v)
    G[v].append(u)
return dfs(G, target, t, 1, 0)
```

smallest-integer-divisible-by-k.py

```
# DESC
# Return the length of N.  If there is no such N, return -1.
# Example 1:
# Example 2:
# Given a positive integer K, you need find the smallest positive integer N such t
# hat N is divisible by K, and N only contains the digit 1.
# Example 3:
# Note:

# NOTE
# 1 <= K <= 10^5

# EXAMPLE
# Input: 1
# Output: 1
# Explanation: The smallest answer is N = 1, which has length 1.
# Input: 2
# Output: -1
# Explanation: There is no such positive integer N divisible by 2.
# Input: 3
# Output: 3
# Explanation: The smallest answer is N = 111, which has length 3.

# 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):
            result = (result*10+1) % K
            if not result:
                return N
        assert(False)
        return -1  # never reach
```

range-sum-query-2d-immutable.py

```
# range-sum-query-2d-immutable is not found.
# Time:  ctor:   O(m * n),
#        lookup: O(1)
# Space: O(m * n)

class NumMatrix(object):
    def __init__(self, matrix):
        """
        initialize your data structure here.
        :type matrix: List[List[int]]
        """
        if not matrix:
            return

        m, n = len(matrix), len(matrix[0])
        self.__sums = [[0 for _ in xrange(n+1)] for _ in xrange(m+1)]
        for i in xrange(1, m+1):
            for j in xrange(1, n+1):
                self.__sums[i][j] = self.__sums[i][j-1] + self.__sums[i-1][j] - \
                                    self.__sums[i-1][j-1] + matrix[i-1][j-1]

    def sumRegion(self, row1, col1, row2, col2):
        """
        sum of elements matrix[(row1,col1)..(row2,col2)], inclusive.
        :type row1: int
        :type col1: int
        :type row2: int
        :type col2: int
        :rtype: int
        """
        return self.__sums[row2+1][col2+1] - self.__sums[row2+1][col1] - \
               self.__sums[row1][col2+1] + self.__sums[row1][col1]
```

greatest-sum-divisible-by-three.py

```python
# greatest-sum-divisible-by-three is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxSumDivThree(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        dp = [0, 0, 0]
        for num in nums:
            for i in [num+x for x in dp]:
                dp[i%3] = max(dp[i%3], i)
        return dp[0]
```

minimum-moves-to-reach-target-with-rotations.py

```python
# minimum-moves-to-reach-target-with-rotations is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def minimumMoves(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        level, q, lookup = 0, [(0, 0, False)], set()
        while q:
            next_q = []
            for r, c, is_vertical in q:
                if (r, c, is_vertical) in lookup:
                    continue
                if (r, c, is_vertical) == (len(grid)-1, len(grid)-2, False):
                    return level
                lookup.add((r, c, is_vertical))
                if not is_vertical:
                    if c+2 != len(grid[0]) and grid[r][c+2] == 0:
                        next_q.append((r, c+1, is_vertical))
                    if r+1 != len(grid) and grid[r+1][c] == 0 and grid[r+1][c+1] == 0:
                        next_q.append((r+1, c, is_vertical))
                        next_q.append((r, c, not is_vertical))
                else:
                    if r+2 != len(grid) and grid[r+2][c] == 0:
                        next_q.append((r+1, c, is_vertical))
                    if c+1 != len(grid) and grid[r][c+1] == 0 and grid[r+1][c+1] == 0:
                        next_q.append((r, c+1, is_vertical))
                        next_q.append((r, c, not is_vertical))
            q = next_q
            level += 1
        return -1
```

count-good-triplets.py

```python
# count-good-triplets is not found.
# Time:  O(n^3)
# Space: O(1)

class Solution(object):
    def countGoodTriplets(self, arr, a, b, c):
        """
        :type arr: List[int]
        :type a: int
        :type b: int
        :type c: int
        :rtype: int
        """
        return sum(abs(arr[i]-arr[j]) <= a and
                   abs(arr[j]-arr[k]) <= b and
                   abs(arr[k]-arr[i]) <= c
                   for i in xrange(len(arr)-2)
                       for j in xrange(i+1, len(arr)-1)
                           for k in xrange(j+1, len(arr)))
```

dota2-senate.py

```
# DESC
# In the world of Dota2, there are two parties: the Radiant and the Dire.
# Note:
# Suppose every senator is smart enough and will play the best strategy for his ow
# n party, you need to predict which party will finally announce the victory and m
# ake the change in the Dota2 game. The output should be Radiant or Dire.
# The round-based procedure starts from the first senator to the last senator in t
# he given order. This procedure will last until the end of voting. All the senato
# rs who have lost their rights will be skipped during the procedure.
# Example 2:
# Example 1:
# The Dota2 senate consists of senators coming from two parties. Now the senate wa
# nts to make a decision about a change in the Dota2 game. The voting for this cha
# nge is a round-based procedure. In each round, each senator can exercise one of
# the two rights:
# Given a string representing each senator's party belonging. The character 'R' an
# d '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.

# NOTE
# Ban one senator's right:
#
#         A senator can make another senator lose all his right
# s 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 d
# ecision about the change in the game.
# The length of the given string will in the range [1, 10,000].

# EXAMPLE
# 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 sen
# ator 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 on
# ly guy in the senate who can vote.
# Input: "RDD"
# Output: "Dire"
# Explanation:
# The first senator comes from Radiant a
# nd he can just ban the next senator's right in the round 1.
# And the second sena
# tor 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 ro
# und 1.
# And in the round 2, the third senator can just announce the victory sinc
# e he is the only guy in the senate who can vote.

# 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

```
# DESC
# [1,7,4,9,2,5]
# Example 2:
# Example 1:
# 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.
# A sequence of numbers is called a wiggle sequence if the differences between suc
# cessive numbers strictly alternate between positive and negative. The first diff
# erence (if one exists) may be either positive or negative. A sequence with fewer
#  than two elements is trivially a wiggle sequence.
# Follow up:
#
# Can you do it in O(n) time?
# 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 eleme
# nts (eventually, also zero) from the original sequence, leaving the remaining el
# ements in their original order.
# Example 3:

# NOTE
#

# EXAMPLE
# Input: [1,7,4,9,2,5]
# Output: 6
# Explanation: The entire sequence is a wiggle sequence.
# Input: [1,2,3,4,5,6,7,8,9]
# Output: 2
# Input: [1,17,5,10,13,15,10,5,16,8]
# Output: 7
# Explanation: There are several subs
# equences that achieve this length. One is [1,17,10,13,10,16,8].

# 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

```
# DESC
# Example 1:
# Return the starting gas station's index if you can travel around the circuit onc
# e in the clockwise direction, otherwise return -1.
# 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 ta
# nk at one of the gas stations.
# There are N gas stations along a circular route, where the amount of gas at stat
# ion i is gas[i].
# Example 2:
# cost[i]
# Note:

# NOTE
# Both input arrays are non-empty and have the same length.
# Each element in the input arrays is a non-negative integer.
# If there exists a solution, it is guaranteed to be unique.

# EXAMPLE
# 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 enoug
# h to travel back to station 3.
# Therefore, return 3 as the starting index.
# 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
# Trav
# el 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 w
# here you start.
```

```python
# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @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
# DESC
# Example 1:
# Constraints:

# NOTE
# The value of nodes is between 1 and 100.
# The number of nodes in the tree is between 1 and 10^4.

# EXAMPLE
# Input: root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
# Output: 15

# 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)
```

counting-elements.py

```
# counting-elements is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def countElements(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        lookup = set(arr)
        return sum(1 for x in arr if x+1 in lookup)


# Time:  O(nlogn)
# Space: O(1)
class Solution(object):
    def countElements(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        arr.sort()
        result, l = 0, 1
        for i in xrange(len(arr)-1):
            if arr[i] == arr[i+1]:
                l += 1
                continue
            if arr[i]+1 == arr[i+1]:
                result += l
            l = 1
        return result
```

largest-component-size-by-common-factor.py

```
# DESC
# Example 2:
# Note:
# Example 1:
# Given a non-empty array of unique positive integers A, consider the following graph:
# Return the size of the largest connected component in the graph.
# Example 3:

# NOTE
# 1 <= A[i] <= 100000
# There is an edge between A[i] and A[j] if and only if A[i] and A[j] share a comm
# on factor greater than 1.
# There are A.length nodes, labelled A[0] to A[A.length - 1];
# 1 <= A.length <= 20000

# EXAMPLE
# Input: [20,50,9,63]
# Output: 2
# Input: [4,6,15,35]
# Output: 4
# Input: [2,3,6,7,4,12,21,39]
# Output: 8

# Time:  O(f * n), f is the max number of unique prime factors
# Space: O(p + n), p is the total number of unique primes

import collections


class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)
        self.size = [1]*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)
        self.size[max(x_root, y_root)] += self.size[min(x_root, y_root)]
        return True


class Solution(object):
    def largestComponentSize(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        def primeFactors(i):  # prime factor decomposition
            result = []
            d = 2
            if i%d == 0:
```

```
            while i%d == 0:
                i //= d
            result.append(d)
        d = 3
        while d*d <= i:
            if i%d == 0:
                while i%d == 0:
                    i //= d
                result.append(d)
            d += 2
        if i > 2:
            result.append(i)
        return result

    union_find = UnionFind(len(A))
    nodesWithCommonFactor = collections.defaultdict(int)
    for i in xrange(len(A)):
        for factor in primeFactors(A[i]):
            if factor not in nodesWithCommonFactor:
                nodesWithCommonFactor[factor] = i
            union_find.union_set(nodesWithCommonFactor[factor], i)
    return max(union_find.size)
```

construct-string-from-binary-tree.py

```
# DESC
# The null node needs to be represented by empty parenthesis pair "()". And you ne
# ed to omit all the empty parenthesis pairs that don't affect the one-to-one mapp
# ing relationship between the string and the original binary tree.
# You need to construct a string consists of parenthesis and integers from a binar
# y tree with the preorder traversing way.
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: Binary tree: [1,2,3,null,4]
#        1
#      /   \
#     2     3
#      \
#
#   4
#
# Output: "1(2()(4))(3)"
#
# Explanation: Almost the same as the first example,
#
# except we can't omit the first parenthesis pair to break the one-to-one mappin
# g relationship between the input and the output.
# Input: Binary tree: [1,2,3,4]
#        1
#      /   \
#     2     3
#    /
#   4
#
#
# Output: "1(2(4))(3)"
#
# Explanation: Originallay it needs to be "1(2(4())(3()())
# ",
# but you need to omit all the unnecessary empty parenthesis pairs.
# And it wi
# ll be "1(2(4))(3)".

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

class Solution(object):
    def tree2str(self, t):
        """
        :type t: TreeNode
        :rtype: str
        """
        if not t: return ""
        s = str(t.val)
        if t.left or t.right:
            s += "(" + self.tree2str(t.left) + ")"
        if t.right:
            s += "(" + self.tree2str(t.right) + ")"
```

```
    return s
```

clone-n-ary-tree.py

```
# clone-n-ary-tree is not found.
# Time:  O(n)
# Space: O(h)

# Definition for a Node.
class Node(object):
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []


class Solution(object):
    def cloneTree(self, root):
        """
        :type root: Node
        :rtype: Node
        """
        result = [None]
        stk = [(1, (root, result))]
        while stk:
            step, params = stk.pop()
            if step == 1:
                node, ret = params
                if not node:
                    continue
                ret[0] = Node(node.val)
                for child in reversed(node.children):
                    ret1 = [None]
                    stk.append((2, (ret1, ret)))
                    stk.append((1, (child, ret1)))
            else:
                ret1, ret = params
                ret[0].children.append(ret1[0])
        return result[0]


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def cloneTree(self, root):
        """
        :type root: Node
        :rtype: Node
        """
        def dfs(node):
            if not node:
                return None
            copy = Node(node.val)
            for child in node.children:
                copy.children.append(dfs(child))
            return copy

        return dfs(root)
```

binary-tree-longest-consecutive-sequence.py

```python
# binary-tree-longest-consecutive-sequence is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def longestConsecutive(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        self.max_len = 0

        def longestConsecutiveHelper(root):
            if not root:
                return 0

            left_len = longestConsecutiveHelper(root.left)
            right_len = longestConsecutiveHelper(root.right)

            cur_len = 1
            if root.left and root.left.val == root.val + 1:
                cur_len = max(cur_len, left_len + 1)
            if root.right and root.right.val == root.val + 1:
                cur_len = max(cur_len, right_len + 1)

            self.max_len = max(self.max_len, cur_len)

            return cur_len

        longestConsecutiveHelper(root)
        return self.max_len
```

super-egg-drop.py

```python
# super-egg-dro is not found.
# Time:  O(klogn)
# Space: O(1)

class Solution(object):
    def superEggDrop(self, K, N):
        """
        :type K: int
        :type N: int
        :rtype: int
        """
        def check(n, K, N):
            # Each combination of n moves with k broken eggs could represent a unique F.
            # Thus, the range size of F that all cominations can cover
            # is the sum of C(n, k), k = 1..K
            total, c = 0, 1
            for k in xrange(1, K+1):
                c *= n-k+1
                c //= k
                total += c
                if total >= N:
                    return True
            return False

        left, right = 1, N
        while left <= right:
            mid = left + (right-left)//2
            if check(mid, K, N):
                right = mid-1
            else:
                left = mid+1
        return left
```

check-if-word-is-valid-after-substitutions.py

```
# DESC
# Example 4:
# We are given that the string "abc" is valid.
# Return true if and only if the given string S is valid.
# If for example S = "abc", then examples of valid strings are: "abc", "aabcbc", "
# abcabc", "abcabcababcc".  Examples of invalid strings are: "abccba", "ab", "caba
# bc", "bac".
# Example 1:
# Example 2:
# Example 3:
# From any valid string V, we may split V into two pieces X and Y such that X + Y
# (X concatenated with Y) is equal to V.  (X or Y may be empty.)  Then, X + "abc"
# + Y is also valid.
# Note:

# NOTE
# S[i] is 'a', 'b', or 'c'
# 1 <= S.length <= 20000

# EXAMPLE
# Input: "abcabcababcc"
# Output: true
# Explanation:
# "abcabcabc" is valid after cons
# ecutive insertings of "abc".
# Then we can insert "abc" before the last letter, re
# sulting in "abcabcab" + "abc" + "c" which is "abcabcababcc".
# Input: "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".
# Input: "abccba"
# Output: false
# Input: "cababc"
# Output: false

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

class Solution(object):
    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
```

minimum-number-of-taps-to-open-to-water-a-garden.py

```python
# minimum-number-of-taps-to-open-to-water-a-garden is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def minTaps(self, n, ranges):
        """
        :type n: int
        :type ranges: List[int]
        :rtype: int
        """
        def jump_game(A):
            jump_count, reachable, curr_reachable = 0, 0, 0
            for i, length in enumerate(A):
                if i > reachable:
                    return -1
                if i > curr_reachable:
                    curr_reachable = reachable
                    jump_count += 1
                reachable = max(reachable, i+length)
            return jump_count

        max_range = [0]*(n+1)
        for i, r in enumerate(ranges):
            left, right = max(i-r, 0), min(i+r, n)
            max_range[left] = max(max_range[left], right-left)
        return jump_game(max_range)
```

the-earliest-moment-when-everyone-become-friends.py

```python
# the-earliest-moment-when-everyone-become-friends is not found.
# Time:  O(nlogn)
# Space: O(n)

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:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        self.count -= 1
        return True


class Solution(object):
    def earliestAcq(self, logs, N):
        """
        :type logs: List[List[int]]
        :type N: int
        :rtype: int
        """
        logs.sort()
        union_find = UnionFind(N)
        for t, a, b in logs:
            union_find.union_set(a, b)
            if union_find.count == 1:
                return t
        return -1
```

find-the-kth-smallest-sum-of-a-matrix-with-sorted-rows.py

```python
# find-the-kth-smallest-sum-of-a-matrix-with-sorted-rows is not found.
# Time:  O(m * klogk)
# Space: O(k)

import heapq


class Solution(object):
    def kthSmallest(self, mat, k):
        """
        :type mat: List[List[int]]
        :type k: int
        :rtype: int
        """
        def kSmallestPairs(nums1, nums2, k):
            result, min_heap = [], []
            for c in xrange(min(len(nums1), k)):
                heapq.heappush(min_heap, (nums1[c]+nums2[0], 0))
                c += 1
            while len(result) != k and min_heap:
                total, c = heapq.heappop(min_heap)
                result.append(total)
                if c+1 == len(nums2):
                    continue
                heapq.heappush(min_heap, (total-nums2[c]+nums2[c+1], c+1))
            return result

        result = mat[0]
        for r in xrange(1, len(mat)):
            result = kSmallestPairs(result, mat[r], k)
        return result[k-1]


# Time:  O((k + m) * log(m * MAX_NUM)) ~ O(k * m * log(m * MAX_NUM))
# Space: O(m)
class Solution2(object):
    def kthSmallest(self, mat, k):
        """
        :type mat: List[List[int]]
        :type k: int
        :rtype: int
        """
        def countArraysHaveSumLessOrEqual(mat, k, r, target):  # Time: O(k + m) ~ O(k * m)
            if target < 0:
                return 0
            if r == len(mat):
                return 1
            result = 0
            for c in xrange(len(mat[0])):
                cnt = countArraysHaveSumLessOrEqual(mat, k-result, r+1, target-mat[r][c])
                if not cnt:
                    break
                result += cnt
                if result > k:
                    break
            return result

        MAX_NUM = 5000
```

```
left, right = len(mat), len(mat)*MAX_NUM
while left <= right:
    mid = left + (right-left)//2
    cnt = countArraysHaveSumLessOrEqual(mat, k, 0, mid)
    if cnt >= k:
        right = mid-1
    else:
        left = mid+1
return left
```

minimum-time-to-build-blocks.py

```python
# minimum-time-to-build-blocks is not found.
# Time:  O(nlogn)
# Space: O(n)

import heapq


class Solution(object):
    def minBuildTime(self, blocks, split):
        """
        :type blocks: List[int]
        :type split: int
        :rtype: int
        """
        heapq.heapify(blocks)
        while len(blocks) != 1:
            x, y = heapq.heappop(blocks), heapq.heappop(blocks)
            heapq.heappush(blocks, y+split)
        return heapq.heappop(blocks)
```

insert-into-a-cyclic-sorted-list.py

```
# insert-into-a-cyclic-sorted-list is not found.
# Time:  O(n)
# Space: O(1)

class Node(object):
    def __init__(self, val, next):
        self.val = val
        self.next = next


class Solution(object):
    def insert(self, head, insertVal):
        """
        :type head: Node
        :type insertVal: int
        :rtype: Node
        """
        def insertAfter(node, val):
            node.next = Node(val, node.next)

        if not head:
            node = Node(insertVal, None)
            node.next = node
            return node

        curr = head
        while True:
            if curr.val < curr.next.val:
                if curr.val <= insertVal and \
                   insertVal <= curr.next.val:
                    insertAfter(curr, insertVal)
                    break
            elif curr.val > curr.next.val:
                if curr.val <= insertVal or \
                   insertVal <= curr.next.val:
                    insertAfter(curr, insertVal)
                    break
            else:
                if curr.next == head:
                    insertAfter(curr, insertVal)
                    break
            curr = curr.next
        return head
```

substring-with-concatenation-of-all-words.py

```
# DESC
# Example 2:
# Example 1:
# You are given a string, s, and a list of words, words, that are all of the same
# length. Find all starting indices of substring(s) in s that is a concatenation o
# f each word in words exactly once and without any intervening characters.

# NOTE
#

# EXAMPLE
# Input:
#   s = "barfoothefoobarman",
#   words = ["foo","bar"]
# Output: [0,9]
# Explana
# tion: Substrings starting at index 0 and 9 are "barfoo" and "foobar" respectivel
# y.
# The output order does not matter, returning [9,0] is fine too.
# Input:
#   s = "wordgoodgoodgoodbestword",
#   words = ["word","good","best","word"]
#
# Output: []

# Time:  O((m + n) * k), where m is string length, n is dictionary size, k is word length
# Space: O(n * k)

import collections


class Solution(object):
    def findSubstring(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: List[int]
        """
        if not words:
            return []

        result, m, n, k = [], len(s), len(words), len(words[0])
        if m < n*k:
            return result

        lookup = collections.defaultdict(int)
        for i in words:
            lookup[i] += 1                # Space: O(n * k)

        for i in xrange(k):               # Time:  O(k)
            left, count = i, 0
            tmp = collections.defaultdict(int)
            for j in xrange(i, m-k+1, k): # Time:  O(m / k)
                s1 = s[j:j+k]             # Time:  O(k)
                if s1 in lookup:
                    tmp[s1] += 1
                    count += 1
                    while tmp[s1] > lookup[s1]:
```

```
                    tmp[s[left:left+k]] -= 1
                    count -= 1
                    left += k
                if count == n:
                    result.append(left)
            else:
                tmp = collections.defaultdict(int)
                count = 0
                left = j+k
        return result


# Time:  O(m * n * k), where m is string length, n is dictionary size, k is word length
# Space: O(n * k)
class Solution2(object):
    def findSubstring(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: List[int]
        """
        result, m, n, k = [], len(s), len(words), len(words[0])
        if m < n*k:
            return result

        lookup = collections.defaultdict(int)
        for i in words:
            lookup[i] += 1                              # Space: O(n * k)

        for i in xrange(m+1-k*n):                       # Time: O(m)
            cur, j = collections.defaultdict(int), 0
            while j < n:                                # Time: O(n)
                word = s[i+j*k:i+j*k+k]                 # Time: O(k)
                if word not in lookup:
                    break
                cur[word] += 1
                if cur[word] > lookup[word]:
                    break
                j += 1
            if j == n:
                result.append(i)

        return result
```

sentence-similarity-ii.py

```python
# sentence-similarity-ii is not found.
# Time:  O(n + p)
# Space: O(p)

import itertools


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 areSentencesSimilarTwo(self, words1, words2, pairs):
        """
        :type words1: List[str]
        :type words2: List[str]
        :type pairs: List[List[str]]
        :rtype: bool
        """
        if len(words1) != len(words2): return False

        lookup = {}
        union_find = UnionFind(2 * len(pairs))
        for pair in pairs:
            for p in pair:
                if p not in lookup:
                    lookup[p] = len(lookup)
            union_find.union_set(lookup[pair[0]], lookup[pair[1]])

        return all(w1 == w2 or
                   w1 in lookup and w2 in lookup and
                   union_find.find_set(lookup[w1]) == union_find.find_set(lookup[w2])
                   for w1, w2 in itertools.izip(words1, words2))
```

is-subsequence.py

```
# DESC
# "ace"
# A subsequence of a string is a new string which is formed from the original stri
# ng by deleting some (can be none) of the characters without disturbing the relat
# ive positions of the remaining characters. (ie, "ace" is a subsequence of "abcde
# " while "aec" is not).
# Given a string s and a string t, check if s is subsequence of t.
# Example 1:
# Constraints:
# Follow up:
#
# If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B,
#  and you want to check one by one to see if T has its subsequence. In this scena
# rio, how would you change your code?
# Example 2:
# Credits:
#
# Special thanks to @pbrother for adding this problem and creating all t
# est cases.

# NOTE
# 0 <= t.length <= 10^4
# 0 <= s.length <= 100
# Both strings consists only of lowercase characters.

# EXAMPLE
# Input: s = "axc", t = "ahbgdc"
# Output: false
# Input: s = "abc", t = "ahbgdc"
# Output: true

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isSubsequence(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        if not s:
            return True

        i = 0
        for c in t:
            if c == s[i]:
                i += 1
            if i == len(s):
                break
        return i == len(s)
```

reverse-nodes-in-k-group.py

```python
# reverse-nodes-in-k-grou is not found.
# 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 k, an integer
    # @return a ListNode
    def reverseKGroup(self, head, k):
        dummy = ListNode(-1)
        dummy.next = head

        cur, cur_dummy = head, dummy
        length = 0

        while cur:
            next_cur = cur.next
            length = (length + 1) % k

            if length == 0:
                next_dummy = cur_dummy.next
                self.reverse(cur_dummy, cur.next)
                cur_dummy = next_dummy

            cur = next_cur

        return dummy.next

    def reverse(self, begin, end):
            first = begin.next
            cur = first.next

            while cur != end:
                first.next = cur.next
                cur.next = begin.next
                begin.next = cur
                cur = first.next
```

subsets-ii.py

```
# DESC
# Given a collection of integers that might contain duplicates, nums, return all p
# ossible subsets (the power set).
# Note: The solution set must not contain duplicate subsets.
# Example:

# NOTE
#

# 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)
            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

```
# DESC
# Note:
# Example 3:
# Delete all insufficient nodes simultaneously, and return the root of the resulti
# ng binary tree.
# Given the root of a binary tree, consider all root to leaf paths: paths from the
#  root to any leaf.  (A leaf is a node with no children.)
# Example 2:
# A node is insufficient if every such root to leaf path intersecting this node ha
# s sum strictly less than limit.
# Example 1:

# NOTE
# The given tree will have between 1 and 5000 nodes.
# -10^5 <= node.val <= 10^5
# -10^9 <= limit <= 10^9

# EXAMPLE
# 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]
# Input: root = [1,2,3,4,-99,-99,7,8,9,-99,-99,12,13,-99,14], limit = 1
#
# Output: [
# 1,2,3,4,null,null,7,8,9,null,14]
# 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

```
# DESC
# Example 1:
# Example 3:
# 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:
# Return Construct(B).
# Example 2:
# Note that we were not given A directly, only a root node root = Construct(A).
# Constraints:
# Suppose B is a copy of A with the value val appended to it.  It is guaranteed th
# at B has unique values.
# We are given the root node of a maximum tree: a tree where every node has a valu
# e greater than any other value in its subtree.

# NOTE
# 1 <= B.length <= 100
# The right child of root will be Construct([A[i+1], A[i+2], ..., A[A.length - 1]])
# Return root.
# If A is empty, return null.
# The left child of root will be Construct([A[0], A[1], ..., A[i-1]])
# Otherwise, let A[i] be the largest element of A.  Create a root node with value A[i].

# EXAMPLE
# 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]
# 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]
# Input: root = [4,1,3,null,null,2], val = 5
# Output: [5,4,null,1,3,null,null,2]
# Ex
# planation: A = [1,4,2,3], B = [1,4,2,3,5]

# Time:  O(h)
# Space: O(1)

# 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
```

handshakes-that-dont-cross.py

```
# handshakes-that-dont-cross is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numberOfWays(self, num_people):
        """
        :type num_people: int
        :rtype: int
        """
        MOD = 10**9+7
        def inv(x, m):  # Euler's Theorem
            return pow(x, m-2, m)  # O(logMOD) = O(1)

        def nCr(n, k, m):
            if n-k < k:
                return nCr(n, n-k, m)
            result = 1
            for i in xrange(1, k+1):
                result = result*(n-k+i)*inv(i, m)%m
            return result

        n = num_people//2
        return nCr(2*n, n, MOD)*inv(n+1, MOD) % MOD  # Catalan number


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def numberOfWays(self, num_people):
        """
        :type num_people: int
        :rtype: int
        """
        MOD = 10**9+7
        dp = [0]*(num_people//2+1)
        dp[0] = 1
        for k in xrange(1, num_people//2+1):
            for i in xrange(k):
                dp[k] = (dp[k] + dp[i]*dp[k-1-i]) % MOD
        return dp[num_people//2]
```

binary-tree-postorder-traversal.py

```
# binary-tree-postorder-traversal is not found.
# 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 postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        dummy = TreeNode(0)
        dummy.left = root
        result, cur = [], dummy
        while cur:
            if cur.left is None:
                cur = cur.right
            else:
                node = cur.left
                while node.right and node.right != cur:
                    node = node.right

                if node.right is None:
                    node.right = cur
                    cur = cur.left
                else:
                    result += self.traceBack(cur.left, node)
                    node.right = None
                    cur = cur.right

        return result

    def traceBack(self, frm, to):
        result, cur = [], frm
        while cur is not to:
            result.append(cur.val)
            cur = cur.right
        result.append(to.val)
        result.reverse()
        return result


# Time:  O(n)
# Space: O(h)
# Stack Solution
class Solution2(object):
    def postorderTraversal(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, True))
        stack.append((root.right, False))
        stack.append((root.left, False))
return result
```

walls-and-gates.py

```
# walls-and-gates is not found.
# Time:  O(m * n)
# Space: O(g)

from collections import deque

class Solution(object):
    def wallsAndGates(self, rooms):
        """
        :type rooms: List[List[int]]
        :rtype: void Do not return anything, modify rooms in-place instead.
        """
        INF = 2147483647
        q = deque([(i, j) for i, row in enumerate(rooms) for j, r in enumerate(row) if not r])
        while q:
            (i, j) = q.popleft()
            for I, J in (i+1, j), (i-1, j), (i, j+1), (i, j-1):
                if 0 <= I < len(rooms) and 0 <= J < len(rooms[0]) and \
                   rooms[I][J] == INF:
                    rooms[I][J] = rooms[i][j] + 1
                    q.append((I, J))
```

lfu-cache.py

```
# DESC
# 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 reaches its capacity, it should
# invalidate the least frequently used item before inserting a new item. For the p
# urpose of this problem, when there is a tie (i.e., two or more keys that have th
# e same frequency), the least recently used key would be evicted.
# Note that the number of times an item is used is the number of calls to the get
# and put functions for that item since it was inserted. This number is set to zer
# o when the item is removed.
# Design and implement a data structure for Least Frequently Used (LFU) cache. It
# should support the following operations: get and put.
# get(key)
# Example:
# Follow up:
#
# Could you do both operations in O(1) time complexity?

# NOTE
#

# EXAMPLE
# LFUCache cache = new LFUCache( 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.get(3);       // returns 3.
# cache
# .put(4, 4);    // evicts key 1.
# cache.get(1);       // returns -1 (not found)
# ca
# che.get(3);       // returns 3
# cache.get(4);       // returns 4

# Time:  O(1), per operation
# Space: O(k), k is the capacity of cache

import collections


class ListNode(object):
    def __init__(self, key, value, freq):
        self.key = key
        self.val = value
        self.freq = freq
        self.next = None
        self.prev = None


class LinkedList(object):
    def __init__(self):
```

```
        self.head = None
        self.tail = None

    def append(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:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        node.next, node.prev = None, None  # make node clean


class LFUCache(object):

    def __init__(self, capacity):
        """
        :type capacity: int
        """
        self.__capa = capacity
        self.__size = 0
        self.__min_freq = 0
        self.__freq_to_nodes = collections.defaultdict(LinkedList)
        self.__key_to_node = {}


    def get(self, key):
        """
        :type key: int
        :rtype: int
        """
        if key not in self.__key_to_node:
            return -1

        old_node = self.__key_to_node[key]
        self.__key_to_node[key] = ListNode(key, old_node.val, old_node.freq)
        self.__freq_to_nodes[old_node.freq].delete(old_node)
        if not self.__freq_to_nodes[self.__key_to_node[key].freq].head:
            del self.__freq_to_nodes[self.__key_to_node[key].freq]
            if self.__min_freq == self.__key_to_node[key].freq:
                self.__min_freq += 1

        self.__key_to_node[key].freq += 1
        self.__freq_to_nodes[self.__key_to_node[key].freq].append(self.__key_to_node[key])

        return self.__key_to_node[key].val


    def put(self, key, value):
```

```
        """
        :type key: int
        :type value: int
        :rtype: void
        """
        if self.__capa <= 0:
            return

        if self.get(key) != -1:
            self.__key_to_node[key].val = value
            return

        if self.__size == self.__capa:
            del self.__key_to_node[self.__freq_to_nodes[self.__min_freq].head.key]
            self.__freq_to_nodes[self.__min_freq].delete(self.__freq_to_nodes[self.__min_freq].head)
            if not self.__freq_to_nodes[self.__min_freq].head:
                del self.__freq_to_nodes[self.__min_freq]
            self.__size -= 1

        self.__min_freq = 1
        self.__key_to_node[key] = ListNode(key, value, self.__min_freq)
        self.__freq_to_nodes[self.__key_to_node[key].freq].append(self.__key_to_node[key])
        self.__size += 1
```

number-of-equivalent-domino-pairs.py

```
# DESC
# Constraints:
# Return the number of pairs (i, j) for which 0 <= i < j < dominoes.length, and do
# minoes[i] is equivalent to dominoes[j].
# (i, j)
# Example 1:
# Given a list of dominoes, dominoes[i] = [a, b] is equivalent to dominoes[j] = [c
# , d] if and only if either (a==c and b==d), or (a==d and b==c) - that is, one do
# mino can be rotated to be equal to another domino.

# NOTE
# 1 <= dominoes[i][j] <= 9
# 1 <= dominoes.length <= 40000

# EXAMPLE
# Input: dominoes = [[1,2],[2,1],[3,4],[5,6]]
# Output: 1

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

import collections


class Solution(object):
    def numEquivDominoPairs(self, dominoes):
        """
        :type dominoes: List[List[int]]
        :rtype: int
        """
        counter = collections.Counter((min(x), max(x)) for x in dominoes)
        return sum(v*(v-1)//2 for v in counter.itervalues())
```

shifting-letters.py

```
# DESC
# Call the shift of a letter, the next letter in the alphabet, (wrapping around so
#  that 'z' becomes 'a').
# Example 1:
# We have a string S of lowercase letters, and an integer array shifts.
# Note:
# For example, shift('a') = 'b', shift('t') = 'u', and shift('z') = 'a'.
# Return the final string after all such shifts to S are applied.
# Now for each shifts[i] = x, we want to shift the first i+1 letters of S, x times.

# NOTE
# 0 <= shifts[i] <= 10 ^ 9
# 1 <= S.length = shifts.length <= 20000

# EXAMPLE
# Input: S = "abc", shifts = [3,5,9]
# Output: "rpl"
# Explanation:
# We start with "ab
# c".
# After shifting the first 1 letters of S by 3, we have "dbc".
# After shifting
# the first 2 letters of S by 5, we have "igc".
# After shifting the first 3 letters
#  of S by 9, we have "rpl", the answer.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def shiftingLetters(self, S, shifts):
        """
        :type S: str
        :type shifts: List[int]
        :rtype: str
        """
        result = []
        times = sum(shifts) % 26
        for i, c in enumerate(S):
            index = ord(c) - ord('a')
            result.append(chr(ord('a') + (index+times) % 26))
            times = (times-shifts[i]) % 26
        return "".join(result)
```

count-and-say.py

```python
# count-and-sa is not found.
# Time:  O(n * 2^n)
# Space: O(2^n)

class Solution(object):
    # @return a string
    def countAndSay(self, n):
        seq = "1"
        for i in xrange(n - 1):
            seq = self.getNext(seq)
        return seq

    def getNext(self, seq):
        i, next_seq = 0, ""
        while i < len(seq):
            cnt = 1
            while i < len(seq) - 1 and seq[i] == seq[i + 1]:
                cnt += 1
                i += 1
            next_seq += str(cnt) + seq[i]
            i += 1
        return next_seq
```

longest-substring-with-at-least-k-repeating-characters.py

```python
# DESC
# Example 1:
# Find the length of the longest substring T of a given string (consists of lowerc
# ase letters only) such that every character in T appears no less than k times.
# Example 2:

# NOTE
#

# EXAMPLE
# Input:
# s = "aaabb", k = 3
#
# Output:
# 3
#
# The longest substring is "aaa", as 'a' is
# repeated 3 times.
# 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))
```

surface-area-of-3d-shapes.py

```python
# DESC
# Return the total surface area of the resulting shapes.
# Example 2:
# Note:
# On a N * N grid, we place some 1 * 1 * 1 cubes.
# Example 4:
# Example 5:
# v = grid[i][j]
# Example 1:
# Each value v = grid[i][j] represents a tower of v cubes placed on top of grid ce
# ll (i, j).
# Example 3:

# NOTE
# 1 <= N <= 50
# 0 <= grid[i][j] <= 50

# EXAMPLE
# Input: [[2,2,2],[2,1,2],[2,2,2]]
# Output: 46
# Input: [[1,2],[3,4]]
# Output: 34
# Input: [[1,1,1],[1,0,1],[1,1,1]]
# Output: 32
# Input: [[2]]
# Output: 10
# Input: [[1,0],[0,2]]
# Output: 16

# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def surfaceArea(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        result = 0
        for i in xrange(len(grid)):
            for j in xrange(len(grid)):
                if grid[i][j]:
                    result += 2 + grid[i][j]*4
                if i:
                    result -= min(grid[i][j], grid[i-1][j])*2
                if j:
                    result -= min(grid[i][j], grid[i][j-1])*2
        return result
```

probability-of-a-two-boxes-having-the-same-number-of-distinct-balls.py

```
# probability-of-a-two-boxes-having-the-same-number-of-distinct-balls is not found.
# Time:  O(k^3 * n^2)
# Space: O(k^2 * n)

import collections


class Solution(object):
    def getProbability(self, balls):
        """
        :type balls: List[int]
        :rtype: float
        """
        def nCrs(n):  # Time: O(n), Space: O(1)
            c = 1
            for k in xrange(n+1):
                yield c
                c *= n-(k+1)+1
                c //= k+1

        def nCr(n, r):  # Time: O(n), Space: O(1)
            if n-r < r:
                return nCr(n, n-r)
            c = 1
            for k in xrange(1, r+1):
                c *= n-k+1
                c //= k
            return c

        dp = collections.defaultdict(int)
        dp[0, 0] = 1  # dp[i, j] is the number of ways with number difference i and color difference j
        for n in balls:  # O(k) times
            new_dp = collections.defaultdict(int)
            for (ndiff, cdiff), count in dp.iteritems():  # O(k^2 * n) times
                for k, new_count in enumerate(nCrs(n)):  # O(n) times
                    new_ndiff = ndiff+(k-(n-k))
                    new_cdiff = cdiff-1 if k == 0 else (cdiff+1 if k == n else cdiff)
                    new_dp[new_ndiff, new_cdiff] += count*new_count
            dp = new_dp
        total = sum(balls)
        return float(dp[0, 0])/nCr(total, total//2)
```

best-time-to-buy-and-sell-stock-iv.py

```
# DESC
# Example 1:
# Example 2:
# Note:
#
# You may not engage in multiple transactions at the same time (ie, you mus
# t sell the stock before you buy again).
# Say you have an array for which the i-th element is the price of a given stock o
# n day i.
# Design an algorithm to find the maximum profit. You may complete at most k trans
# actions.

# NOTE
#

# EXAMPLE
# Input: [3,2,6,5,0,3], k = 2
# Output: 7
# Explanation: Buy on day 2 (price = 2) and
# sell on day 3 (price = 6), profit = 6-2 = 4.
#              Then buy on day 5 (pri
# ce = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
# Input: [2,4,1], k = 2
# Output: 2
# Explanation: Buy on day 1 (price = 2) and sell o
# n day 2 (price = 4), profit = 4-2 = 2.

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


class Solution(object):
    # @return an integer as the maximum profit
    def maxProfit(self, k, prices):
        if k >= len(prices) / 2:
            return self.maxAtMostNPairsProfit(prices)

        return self.maxAtMostKPairsProfit(prices, k)

    def maxAtMostNPairsProfit(self, prices):
        profit = 0
        for i in xrange(len(prices) - 1):
            profit += max(0, prices[i + 1] - prices[i])
        return profit

    def maxAtMostKPairsProfit(self, prices, k):
        max_buy = [float("-inf") for _ in xrange(k + 1)]
        max_sell = [0 for _ in xrange(k + 1)]

        for i in xrange(len(prices)):
            for j in xrange(1, min(k, i/2+1) + 1):
                max_buy[j] = max(max_buy[j], max_sell[j-1] - prices[i])
                max_sell[j] = max(max_sell[j], max_buy[j] + prices[i])

        return max_sell[k]
```

diagonal-traverse.py

```
# DESC
# Example:
# 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.
# Note:
# The total number of elements of the given matrix will not exceed 10,000.

# NOTE
#

# EXAMPLE
# Input:
# [
#  [ 1, 2, 3 ],
#  [ 4, 5, 6 ],
#  [ 7, 8, 9 ]
# ]
#
# Output:  [1,2,4,7,5,3,6,8,9
# ]
#
# Explanation:

# 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
```

sudoku-solver.py

```
# DESC
# A sudoku solution must satisfy all of the following rules:
# Write a program to solve a Sudoku puzzle by filling the empty cells.
# Empty cells are indicated by the character '.'.
# ...and its solution numbers marked in red.
# Note:
# A sudoku puzzle...

# NOTE
# You may assume that the given Sudoku puzzle will have a single unique solution.
# The given board contain only digits 1-9 and the character '.'.
# Each of the digits 1-9 must occur exactly once in each column.
# Each of the digits 1-9 must occur exactly once in each row.
# The given board size is always 9x9.
# Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxe
# s of the grid.

# EXAMPLE
#

# Time:  ((9!)^9)
# Space: (1)

class Solution(object):
    # @param board, a 9x9 2D array
    # Solve the Sudoku by modifying the input board in-place.
    # Do not return any value.
    def solveSudoku(self, board):
        def isValid(board, x, y):
            for i in xrange(9):
                if i != x and board[i][y] == board[x][y]:
                    return False
            for j in xrange(9):
                if j != y and board[x][j] == board[x][y]:
                    return False
            i = 3 * (x / 3)
            while i < 3 * (x / 3 + 1):
                j = 3 * (y / 3)
                while j < 3 * (y / 3 + 1):
                    if (i != x or j != y) and board[i][j] == board[x][y]:
                        return False
                    j += 1
                i += 1
            return True

        def solver(board):
            for i in xrange(len(board)):
                for j in xrange(len(board[0])):
                    if(board[i][j] == '.'):
                        for k in xrange(9):
                            board[i][j] = chr(ord('1') + k)
                            if isValid(board, i, j) and solver(board):
                                return True
                            board[i][j] = '.'
                        return False
            return True

        solver(board)
```

evaluate-division.py

```
# DESC
# Equations are given in the format A / B = k, where A and B are variables represe
# nted as strings, and k is a real number (floating point number). Given some quer
# ies, return the answers. If the answer does not exist, return -1.0.
# The input is:  vector<pair<string, string>> equations, vector<double>& values, v
# ector<pair<string, string>> queries , where equations.size() == values.size(), a
# nd the values are positive. This represents the equations. Return  vector<double
# >.
# a / b = 2.0, b / c = 3.0.
# The input is always valid. You may assume that evaluating the queries will resul
# t in no division by zero and there is no contradiction.
# According to the example above:
# 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 ].

# NOTE
#

# EXAMPLE
# equations = [ ["a", "b"], ["b", "c"] ],
# values = [2.0, 3.0],
# queries = [ ["a", "
# c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

# Time:  O(e + q * |V|!), |V| is the number of variables
# 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 = []
    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
```

insert-interval.py

```
# DESC
# Example 1:
# Given a set of non-overlapping intervals, insert a new interval into the interva
# ls (merge if necessary).
# Example 2:
# NOTE: input types have been changed on April 15, 2019. Please reset to default c
# ode definition to get new method signature.
# You may assume that the intervals were initially sorted according to their start
#  times.

# NOTE
#

# EXAMPLE
# Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
# Outpu
# t: [[1,2],[3,10],[12,16]]
# Explanation: Because the new interval [4,8] overlaps w
# ith [3,5],[6,7],[8,10].
# Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
# Output: [[1,5],[6,9]]

# Time:  O(n)
# 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 insert(self, intervals, newInterval):
        """
        :type intervals: List[Interval]
        :type newInterval: Interval
        :rtype: List[Interval]
        """
        result = []
        i = 0
        while i < len(intervals) and newInterval.start > intervals[i].end:
            result += intervals[i],
            i += 1
        while i < len(intervals) and newInterval.end >= intervals[i].start:
            newInterval = Interval(min(newInterval.start, intervals[i].start), \
                                   max(newInterval.end, intervals[i].end))
            i += 1
        result += newInterval,
        result += intervals[i:]
        return result
```

non-decreasing-array.py

```python
# non-decreasing-arra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def checkPossibility(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        modified, prev = False, nums[0]
        for i in xrange(1, len(nums)):
            if prev > nums[i]:
                if modified:
                    return False
                if i-2 < 0 or nums[i-2] <= nums[i]:
                    prev = nums[i]    # nums[i-1] = nums[i], prev = nums[i]
#                else:
#                    prev = nums[i-1]  # nums[i] = nums[i-1], prev = nums[i]
                modified = True
            else:
                prev = nums[i]
        return True
```

distance-between-bus-stops.py

```python
# distance-between-bus-stops is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def distanceBetweenBusStops(self, distance, start, destination):
        """
        :type distance: List[int]
        :type start: int
        :type destination: int
        :rtype: int
        """
        if start > destination:
            start, destination = destination, start
        s_to_d = sum(itertools.islice(distance, start, destination))
        d_to_s = sum(itertools.islice(distance, 0, start)) + \
                 sum(itertools.islice(distance, destination, len(distance)))
        return min(s_to_d, d_to_s)
```

split-bst.py

```python
# split-bst is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def splitBST(self, root, V):
        """
        :type root: TreeNode
        :type V: int
        :rtype: List[TreeNode]
        """
        if not root:
            return None, None
        elif root.val <= V:
            result = self.splitBST(root.right, V)
            root.right = result[0]
            return root, result[1]
        else:
            result = self.splitBST(root.left, V)
            root.left = result[1]
            return result[0], root
```

rank-teams-by-votes.py

```python
# rank-teams-by-votes is not found.
# Time:  O(m * (n + mlogm)), n is the number of votes
#                          , m is the length of vote
# Space: O(m^2)

class Solution(object):
    def rankTeams(self, votes):
        """
        :type votes: List[str]
        :rtype: str
        """
        count = {v: [0]*len(votes[0]) + [v] for v in votes[0]}
        for vote in votes:
            for i, v in enumerate(vote):
                count[v][i] -= 1
        return "".join(sorted(votes[0], key=count.__getitem__))
```

network-delay-time.py

```
# DESC
# Now, we send a signal from a certain node K. How long will it take for all nodes
#  to receive the signal? If it is impossible, return -1.
# Given times, a list of travel times as directed edges times[i] = (u, v, w), wher
# e u is the source node, v is the target node, and w is the time it takes for a s
# ignal to travel from source to target.
# Example 1:
# times
# There are N network nodes, labelled 1 to N.
# Note:

# NOTE
# K will be in the range [1, N].
# The length of times will be in the range [1, 6000].
# N will be in the range [1, 100].
# All edges times[i] = (u, v, w) will have 1 <= u, v <= N and 0 <= w <= 100.

# EXAMPLE
# Input: times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2
# Output: 2

# Time:  O((|E| + |V|) * log|V|) = O(|E| * log|V|) by using binary heap,
#        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 heapq

# Dijkstra's algorithm
class Solution(object):
    def networkDelayTime(self, times, N, K):
        """
        :type times: List[List[int]]
        :type N: int
        :type K: int
        :rtype: int
        """
        adj = [[] for _ in xrange(N)]
        for u, v, w in times:
            adj[u-1].append((v-1, w))

        result = 0
        lookup = set()
        best = collections.defaultdict(lambda: float("inf"))
        min_heap = [(0, K-1)]
        while min_heap and len(lookup) != N:
            result, u = heapq.heappop(min_heap)
            lookup.add(u)
            if best[u] < result:
                continue
            for v, w in adj[u]:
                if v in lookup: continue
                if result+w < best[v]:
                    best[v] = result+w
                    heapq.heappush(min_heap, (result+w, v))
        return result if len(lookup) == N else -1
```

number-of-nodes-in-the-sub-tree-with-the-same-label.py

```python
# number-of-nodes-in-the-sub-tree-with-the-same-label is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def countSubTrees(self, n, edges, labels):
        """
        :type n: int
        :type edges: List[List[int]]
        :type labels: str
        :rtype: List[int]
        """
        def iter_dfs(labels, adj, node, parent, result):
            stk = [(1, (node, parent, [0]*26))]
            while stk:
                step, params = stk.pop()
                if step == 1:
                    node, parent, ret = params
                    stk.append((4, (node, ret)))
                    stk.append((2, (node, parent, reversed(adj[node]), ret)))
                elif step == 2:
                    node, parent, it, ret = params
                    child = next(it, None)
                    if not child or child == parent:
                        continue
                    ret2 = [0]*26
                    stk.append((2, (node, parent, it, ret)))
                    stk.append((3, (ret2, ret)))
                    stk.append((1, (child, node, ret2)))
                elif step == 3:
                    ret2, ret = params
                    for k in xrange(len(ret2)):
                        ret[k] += ret2[k]
                else:
                    node, ret = params
                    ret[ord(labels[node]) - ord('a')] += 1
                    result[node] += ret[ord(labels[node]) - ord('a')]

        adj = [[] for _ in xrange(n)]
        for u, v in edges:
            adj[u].append(v)
            adj[v].append(u)
        result = [0]*n
        iter_dfs(labels, adj, 0, -1, result)
        return result


# Time:  O(n)
# Space: O(h)
import collections


class Solution2(object):
    def countSubTrees(self, n, edges, labels):
        """
        :type n: int
        :type edges: List[List[int]]
        :type labels: str
```

```
    :rtype: List[int]
    """
    def dfs(labels, adj, node, parent, result):
        count = [0]*26
        for child in adj[node]:
            if child == parent:
                continue
            new_count = dfs(labels, adj, child, node, result)
            for k in xrange(len(new_count)):
                count[k] += new_count[k]
        count[ord(labels[node]) - ord('a')] += 1
        result[node] = count[ord(labels[node]) - ord('a')]
        return count

    adj = [[] for _ in xrange(n)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)
    result = [0]*n
    dfs(labels, adj, 0, -1, result)
    return result
```

perform-string-shifts.py

```python
# perform-string-shifts is not found.
# Time:  O(n + l)
# Space: O(l)

class Solution(object):
    def stringShift(self, s, shift):
        """
        :type s: str
        :type shift: List[List[int]]
        :rtype: str
        """
        left_shifts = 0
        for direction, amount in shift:
            if not direction:
                left_shifts += amount
            else:
                left_shifts -= amount
        left_shifts %= len(s)
        return s[left_shifts:] + s[:left_shifts]
```

minimum-time-difference.py

```python
# DESC
# Example 1:
# Note:

# NOTE
# The input time is legal and ranges from 00:00 to 23:59.
# The number of time points in the given list is at least 2 and won't exceed 20000.

# EXAMPLE
# Input: ["23:59","00:00"]
# Output: 1

# 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]))
```

interleaving-string.py

```
# DESC
# Example 2:
# Example 1:
# Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

# NOTE
#

# EXAMPLE
# Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
# Output: true
# Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
# Output: false

# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    # @return a boolean
    def isInterleave(self, s1, s2, s3):
        if len(s1) + len(s2) != len(s3):
            return False
        if len(s1) > len(s2):
            return self.isInterleave(s2, s1, s3)
        match = [False for i in xrange(len(s1) + 1)]
        match[0] = True
        for i in xrange(1, len(s1) + 1):
            match[i] = match[i -1] and s1[i - 1] == s3[i - 1]
        for j in xrange(1, len(s2) + 1):
            match[0] = match[0] and s2[j - 1] == s3[j - 1]
            for i in xrange(1, len(s1) + 1):
                match[i] = (match[i - 1] and s1[i - 1] == s3[i + j - 1]) \
                                    or (match[i] and s2[j - 1] == s3[i + j - 1])
        return match[-1]

# Time:  O(m * n)
# Space: O(m * n)
# Dynamic Programming
class Solution2(object):
    # @return a boolean
    def isInterleave(self, s1, s2, s3):
        if len(s1) + len(s2) != len(s3):
            return False
        match = [[False for i in xrange(len(s2) + 1)] for j in xrange(len(s1) + 1)]
        match[0][0] = True
        for i in xrange(1, len(s1) + 1):
            match[i][0] = match[i - 1][0] and s1[i - 1] == s3[i - 1]
        for j in xrange(1, len(s2) + 1):
            match[0][j] = match[0][j - 1] and s2[j - 1] == s3[j - 1]
        for i in xrange(1, len(s1) + 1):
            for j in xrange(1, len(s2) + 1):
                match[i][j] = (match[i - 1][j] and s1[i - 1] == s3[i + j - 1]) \
                                    or (match[i][j - 1] and s2[j - 1] == s3[i + j - 1])
        return match[-1][-1]

# Time:  O(m * n)
# Space: O(m * n)
# Recursive + Hash
```

```python
class Solution3(object):
    # @return a boolean
    def isInterleave(self, s1, s2, s3):
        self.match = {}
        if len(s1) + len(s2) != len(s3):
            return False
        return self.isInterleaveRecu(s1, s2, s3, 0, 0, 0)

    def isInterleaveRecu(self, s1, s2, s3, a, b, c):
        if repr([a, b]) in self.match.keys():
            return self.match[repr([a, b])]

        if c == len(s3):
            return True

        result = False
        if a < len(s1) and s1[a] == s3[c]:
            result = result or self.isInterleaveRecu(s1, s2, s3, a + 1, b, c + 1)
        if b < len(s2) and s2[b] == s3[c]:
            result = result or self.isInterleaveRecu(s1, s2, s3, a, b + 1, c + 1)

        self.match[repr([a, b])] = result

        return result
```

binary-tree-zigzag-level-order-traversal.py

```python
# binary-tree-zigzag-level-order-traversal is not found.
# 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

```
# DESC
# Given N, consider a convex N-sided polygon with vertices labelled A[0], A[i], ..
# ., A[N-1] in clockwise order.
# Example 1:
# Example 2:
# Example 3:
# 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 tot
# al 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 triangul
# ation of the polygon.
# Note:

# NOTE
# 1 <= A[i] <= 100
# 3 <= A.length <= 50

# EXAMPLE
# Input: [1,2,3]
# Output: 6
# Explanation: The polygon is already triangulated, and t
# he score of the only triangle is 6.
# Input: [3,7,4,5]
# Output: 144
# Explanation: There are two triangulations, with pos
# sible scores: 3*7*5 + 4*5*7 = 245, or 3*4*5 + 3*4*7 = 144.  The minimum score is
#  144.
# 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.

# 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

```
# DESC
# Note:
# Example 2:
# Example 1:
# Given two non-negative integers num1 and num2 represented as strings, return the
#  product of num1 and num2, also represented as a string.

# NOTE
# The length of both num1 and num2 is < 110.
# You must not use any built-in BigInteger library or convert the inputs to intege
# r directly.
# Both num1 and num2 contain only digits 0-9.
# Both num1 and num2 do not contain any leading zero, except the number 0 itself.

# EXAMPLE
# Input: num1 = "123", num2 = "456"
# Output: "56088"
# Input: num1 = "2", num2 = "3"
# Output: "6"

# 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))
```

minimum-number-of-frogs-croaking.py

```python
# minimum-number-of-frogs-croaking is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minNumberOfFrogs(self, croakOfFrogs):
        """
        :type croakOfFrogs: str
        :rtype: int
        """
        S = "croak"
        lookup = [0]*len(S)
        result = 0
        for c in croakOfFrogs:
            i = S.find(c)
            lookup[i] += 1
            if lookup[i-1]:
                lookup[i-1] -= 1
            elif i == 0:
                result += 1
            else:
                return -1
        return result if result == lookup[-1] else -1
```

circular-array-loop.py

```python
# circular-array-loo is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def circularArrayLoop(self, nums):
        """
        :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

```
# DESC
# Constraints:
# Example 2:
# Given a 2D grid of 0s and 1s, return the number of elements in the largest squar
# e subgrid that has all 1s on its border, or 0 if such a subgrid doesn't exist in
#  the grid.
# Example 1:

# NOTE
# grid[i][j] is 0 or 1
# 1 <= grid.length <= 100
# 1 <= grid[0].length <= 100

# EXAMPLE
# Input: grid = [[1,1,1],[1,0,1],[1,1,1]]
# Output: 9
# Input: grid = [[1,1,0,0]]
# Output: 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
```

image-smoother.py

```
# DESC
# Note:
# Given a 2D integer matrix M representing the gray scale of an image, you need to
#  design a smoother to make the gray scale of each cell becomes the average gray
# scale (rounding down) of all the 8 surrounding cells and itself.  If a cell has
# less than 8 surrounding cells, then use as many as you can.
# Example 1:

# NOTE
# The length and width of the given matrix are in the range of [1, 150].
# The value in the given matrix is in the range of [0, 255].

# EXAMPLE
# Input:
# [[1,1,1],
#  [1,0,1],
#  [1,1,1]]
# Output:
# [[0, 0, 0],
#  [0, 0, 0],
#  [0, 0, 0]]
#
# Explanation:
# For the point (0,0), (0,2), (2,0), (2,2): floor(3/4) = floor(0.75)
#  = 0
# For the point (0,1), (1,0), (1,2), (2,1): floor(5/6) = floor(0.83333333) =
# 0
# For the point (1,1): floor(8/9) = floor(0.88888889) = 0

# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def imageSmoother(self, M):
        """
        :type M: List[List[int]]
        :rtype: List[List[int]]
        """
        def getGray(M, i, j):
            total, count = 0, 0.0
            for r in xrange(-1, 2):
                for c in xrange(-1, 2):
                    ii, jj = i + r, j + c
                    if 0 <= ii < len(M) and 0 <= jj < len(M[0]):
                        total += M[ii][jj]
                        count += 1.0
            return int(total / count)

        result = [[0 for _ in xrange(len(M[0]))] for _ in xrange(len(M))]
        for i in xrange(len(M)):
            for j in xrange(len(M[0])):
                result[i][j] = getGray(M, i, j)
        return result
```

check-if-a-string-contains-all-binary-codes-of-size-k.py

```
# DESC
# Example 4:
# Return True if every binary code of length k is a substring of s. Otherwise, ret
# urn False.
# Constraints:
# Example 3:
# Example 2:
# Example 1:
# Example 5:
# Given a binary string s and an integer k.

# NOTE
# 1 <= s.length <= 5 * 10^5
# s consists of 0's and 1's only.
# 1 <= k <= 20

# EXAMPLE
# 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.
# Input: s = "00110110", k = 2
# Output: true
# Explanation: The binary codes of lengt
# h 2 are "00", "01", "10" and "11". They can be all found as substrings at indici
# es 0, 1, 3 and 2 respectively.
# Input: s = "0000000001011100", k = 4
# Output: false
# Input: s = "00110", k = 2
# Output: true
# Input: s = "0110", k = 2
# Output: false
# Explanation: The binary code "00" is of l
# ength 2 and doesn't exist in the array.

# 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
        """
        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

```
# DESC
# Example 2:
# A partially filled sudoku which is valid.
# Example 1:
# Note:
# Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be valid
# ated according to the following rules:
# The Sudoku board could be partially filled, where empty cells are filled with th
# e character '.'.

# NOTE
# A Sudoku board (partially filled) could be valid but is not necessarily solvable.
# 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 repe
# tition.
# Each row must contain the digits 1-9 without repetition.
# The given board contain only digits 1-9 and the character '.'.
# The given board size is always 9x9.
# Only the filled cells need to be validated according to the mentioned rules.

# EXAMPLE
# 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
# 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
# Explana
# tion: Same as Example 1, except with the 5 in the top left corner being
#     mod
```

```
# ified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

# 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)
```

my-calendar-i.py

```
# DESC
# Your class will have the method, book(int start, int end). Formally, this repres
# ents a booking on the half open interval [start, end), the range of real numbers
#  x such that start <= x < end.
# A double booking happens when two events have some non-empty intersection (ie.,
# there is some time that is common to both events.)
# Note:
# book(int start, int end)
# For each call to the method MyCalendar.book, return true if the event can be add
# ed to the calendar successfully without causing a double booking. Otherwise, ret
# urn false and do not add the event to the calendar.
# Implement a MyCalendar class to store your events. A new event can be added if a
# dding the event will not cause a double booking.
# Example 1:

# NOTE
# In calls to MyCalendar.book(start, end), start and end are integers in the range
#  [0, 10^9].
# The number of calls to MyCalendar.book per test case will be at most 1000.

# EXAMPLE
# MyCalendar();
# MyCalendar.book(10, 20); // returns true
# MyCalendar.book(15, 25);
# // returns false
# MyCalendar.book(20, 30); // returns true
# Explanation:
# The firs
# t event can be booked.  The second can't because time 15 is already booked by an
# other event.
# The third event can be booked, as the first event takes every time
# less than 20, but not including 20.

# Time:  O(nlogn) on average, O(n^2) on worst case
# Space: O(n)

class Node(object):
    def __init__(self, start, end):
        self.__start = start
        self.__end = end
        self.__left = None
        self.__right = None


    def insert(self, node):
        if node.__start >= self.__end:
            if not self.__right:
                self.__right = node
                return True
            return self.__right.insert(node)
        elif node.__end <= self.__start:
            if not self.__left:
                self.__left = node
                return True
            return self.__left.insert(node)
        else:
            return False
```

```
class MyCalendar(object):
    def __init__(self):
        self.__root = None


    def book(self, start, end):
        """
        :type start: int
        :type end: int
        :rtype: bool
        """
        if self.__root is None:
            self.__root = Node(start, end)
            return True
        return self.root.insert(Node(start, end))


# Time:  O(n^2)
# Space: O(n)
class MyCalendar2(object):

    def __init__(self):
        self.__calendar = []


    def book(self, start, end):
        """
        :type start: int
        :type end: int
        :rtype: bool
        """
        for i, j in self.__calendar:
            if start < j and end > i:
                return False
        self.__calendar.append((start, end))
        return True
```

can-i-win.py

```
# DESC
# For example, two players might take turns drawing from a common pool of numbers
# of 1..15 without replacement until they reach a total >= 100.
# Example
# You can always assume that maxChoosableInteger will not be larger than 20 and de
# siredTotal will not be larger than 300.
# Given an integer maxChoosableInteger and another integer desiredTotal, determine
#  if the first player to move can force a win, assuming both players play optimal
# ly.
# What if we change the game so that players cannot re-use integers?
# In the "100 game," two players take turns adding, to a running total, any intege
# r from 1..10. The player who first causes the running total to reach or exceed 1
# 00 wins.

# NOTE
#

# EXAMPLE
# Input:
# maxChoosableInteger = 10
# desiredTotal = 11
#
# Output:
# false
#
# Explanation:
# N
# o 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 wit
# h other integers chosen by the first player, the second player will always win.

# 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

```
# DESC
# Constraints:
# 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.
# Follow up:
#
# How would you handle overflow for very large input integers?
# Note: Numbers in the additive sequence cannot have leading zeros, so sequence 1,
#  2, 03 or 1, 02, 3 is invalid.
# Example 1:
# Additive number is a string whose digits can form additive sequence.
# Example 2:

# NOTE
# 1 <= num.length <= 35
# num consists only of digits '0'-'9'.

# EXAMPLE
# Input: "112358"
# Output: true
# Explanation: The digits can form an additive sequen
# ce: 1, 1, 2, 3, 5, 8.
#              1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
# Input: "199100199"
# Output: true
# Explanation: The additive sequence is: 1, 99, 10
# 0, 199.
#              1 + 99 = 100, 99 + 100 = 199

# 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
                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
```

remove-boxes.py

```
# DESC
# Example 1:
# Constraints:
# Given several boxes with different colors represented by different positive numb
# ers.
#
# You may experience several rounds to remove boxes until there is no box le
# ft. Each time you can choose some continuous boxes with the same color (composed
#  of k boxes, k >= 1), remove them and get k*k points.
#
# Find the maximum points y
# ou can get.

# NOTE
# 1 <= boxes.length <= 100
# 1 <= boxes[i] <= 100

# EXAMPLE
# Input: boxes = [1,3,2,2,2,3,4,3,1]
# Output: 23
# Explanation:
# [1, 3, 2, 2, 2, 3, 4,
#  3, 1]
# ----> [1, 3, 3, 4, 3, 1] (3*3=9 points)
# ----> [1, 3, 3, 3, 1] (1*1=1 po
# ints)
# ----> [1, 1] (3*3=9 points)
# ----> [] (2*2=4 points)

# Time:  O(n^3) ~ O(n^4)
# Space: O(n^3)

class Solution(object):
    def removeBoxes(self, boxes):
        """
        :type boxes: List[int]
        :rtype: int
        """
        def dfs(boxes, l, r, k, lookup):
            if l > r: return 0
            if lookup[l][r][k]: return lookup[l][r][k]

            ll, kk = l, k
            while l < r and boxes[l+1] == boxes[l]:
                l += 1
                k += 1
            result = dfs(boxes, l+1, r, 0, lookup) + (k+1) ** 2
            for i in xrange(l+1, r+1):
                if boxes[i] == boxes[l]:
                    result = max(result, dfs(boxes, l+1, i-1, 0, lookup) + dfs(boxes, i, r, k+1, lookup))
            lookup[ll][r][kk] = result
            return result

        lookup = [[[0]*len(boxes) for _ in xrange(len(boxes)) ] for _ in xrange(len(boxes)) ]
        return dfs(boxes, 0, len(boxes)-1, 0, lookup)
```

data-stream-as-disjoint-intervals.py

```
# DESC
# For example, suppose the integers from the data stream are 1, 3, 7, 2, 6, ..., t
# hen the summary will be:
# What if there are lots of merges and the number of disjoint intervals are small
# compared to the data stream's size?
# Given a data stream input of non-negative integers a1, a2, ..., an, ..., summari
# ze the numbers seen so far as a list of disjoint intervals.
# Follow up:

# NOTE
#

# EXAMPLE
# [1, 1]
# [1, 1], [3, 3]
# [1, 1], [3, 3], [7, 7]
# [1, 3], [7, 7]
# [1, 3], [6, 7]

# Time:  addNum: O(n), getIntervals: O(n), n is the number of disjoint intervals.
# Space: O(n)

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


class SummaryRanges(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__intervals = []

    def addNum(self, val):
        """
        :type val: int
        :rtype: void
        """
        def upper_bound(nums, target):
            left, right = 0, len(nums) - 1
            while left <= right:
                mid = left + (right - left) / 2
                if nums[mid].start > target:
                    right = mid - 1
                else:
                    left = mid + 1
            return left

        i = upper_bound(self.__intervals, val)
        start, end = val, val
        if i != 0 and self.__intervals[i-1].end + 1 >= val:
            i -= 1
        while i != len(self.__intervals) and \
              end + 1 >= self.__intervals[i].start:
            start = min(start, self.__intervals[i].start)
```

```
            end = max(end, self.__intervals[i].end)
            del self.__intervals[i]
        self.__intervals.insert(i, Interval(start, end))

    def getIntervals(self):
        """
        :rtype: List[Interval]
        """
        return self.__intervals
```

best-time-to-buy-and-sell-stock-iii.py

```
# DESC
# Design an algorithm to find the maximum profit. You may complete at most two tra
# nsactions.
# Example 3:
# Example 1:
# Example 2:
# Note: You may not engage in multiple transactions at the same time (i.e., you mu
# st sell the stock before you buy again).
# Say you have an array for which the ith element is the price of a given stock on
#  day i.

# NOTE
#

# EXAMPLE
# Input: [7,6,4,3,1]
# Output: 0
# Explanation: In this case, no transaction is done,
# i.e. max profit = 0.
# Input: [1,2,3,4,5]
# Output: 4
# Explanation: Buy on day 1 (price = 1) and sell on d
# ay 5 (price = 5), profit = 5-1 = 4.
#              Note that you cannot buy on day
#  1, buy on day 2 and sell them later, as you are
#              engaging multiple
# transactions at the same time. You must sell before buying again.
# Input: [3,3,5,0,0,3,1,4]
# Output: 6
# Explanation: Buy on day 4 (price = 0) and sel
# l on day 6 (price = 3), profit = 3-0 = 3.
#              Then buy on day 7 (price
# = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

# Time:  O(n)
# Space: O(1)


class Solution(object):
    # @param prices, a list of integer
    # @return an integer
    def maxProfit(self, prices):
        hold1, hold2 = float("-inf"), float("-inf")
        release1, release2 = 0, 0
        for i in prices:
            release2 = max(release2, hold2 + i)
            hold2 = max(hold2, release1 - i)
            release1 = max(release1, hold1 + i)
            hold1 = max(hold1, -i)
        return release2


# Time:  O(k * n)
# Space: O(k)
class Solution2(object):
    # @param prices, a list of integer
    # @return an integer
    def maxProfit(self, prices):
```

```
        return self.maxAtMostKPairsProfit(prices, 2)

    def maxAtMostKPairsProfit(self, prices, k):
        max_buy = [float("-inf") for _ in xrange(k + 1)]
        max_sell = [0 for _ in xrange(k + 1)]

        for i in xrange(len(prices)):
            for j in xrange(1, min(k, i/2+1) + 1):
                max_buy[j] = max(max_buy[j], max_sell[j-1] - prices[i])
                max_sell[j] = max(max_sell[j], max_buy[j] + prices[i])

        return max_sell[k]


# Time:  O(n)
# Space: O(n)
class Solution3(object):
    # @param prices, a list of integer
    # @return an integer
    def maxProfit(self, prices):
        min_price, max_profit_from_left, max_profits_from_left = \
            float("inf"), 0, []
        for price in prices:
            min_price = min(min_price, price)
            max_profit_from_left = max(max_profit_from_left, price - min_price)
            max_profits_from_left.append(max_profit_from_left)

        max_price, max_profit_from_right, max_profits_from_right = 0, 0, []
        for i in reversed(range(len(prices))):
            max_price = max(max_price, prices[i])
            max_profit_from_right = max(max_profit_from_right,
                                        max_price - prices[i])
            max_profits_from_right.insert(0, max_profit_from_right)

        max_profit = 0
        for i in range(len(prices)):
            max_profit = max(max_profit,
                             max_profits_from_left[i] +
                             max_profits_from_right[i])

        return max_profit
```

design-a-file-sharing-system.py

```
# design-a-file-sharing-system is not found.
# Time:  ctor:    O(1)
#        join:    O(logu + c), u is the number of total joined users
#        leave:   O(logu + c), c is the number of chunks
#        request: O(u)
# Space: O(u * c)

import heapq


# "u ~= n" solution, n is the average number of users who own the chunk
class FileSharing(object):

    def __init__(self, m):
        """
        :type m: int
        """
        self.__users = []
        self.__lookup = set()
        self.__min_heap = []

    def join(self, ownedChunks):
        """
        :type ownedChunks: List[int]
        :rtype: int
        """
        if self.__min_heap:
            userID = heapq.heappop(self.__min_heap)
        else:
            userID = len(self.__users)+1
            self.__users.append(set())
        self.__users[userID-1] = set(ownedChunks)
        self.__lookup.add(userID)
        return userID

    def leave(self, userID):
        """
        :type userID: int
        :rtype: None
        """
        if userID not in self.__lookup:
            return
        self.__lookup.remove(userID)
        self.__users[userID-1] = []
        heapq.heappush(self.__min_heap, userID)

    def request(self, userID, chunkID):
        """
        :type userID: int
        :type chunkID: int
        :rtype: List[int]
        """
        result = []
        for u, chunks in enumerate(self.__users, 1):
            if chunkID not in chunks:
                continue
            result.append(u)
        if not result:
```

```
            return
        self.__users[userID-1].add(chunkID)
        return result


# Time:  ctor:    O(1)
#        join:    O(logu + c), u is the number of total joined users
#        leave:   O(logu + c), c is the number of chunks
#        request: O(nlogn)   , n is the average number of users who own the chunk
# Space: O(u * c + m), m is the total number of unique chunks
import collections
import heapq


# "u >> n" solution
class FileSharing2(object):

    def __init__(self, m):
        """
        :type m: int
        """
        self.__users = []
        self.__lookup = set()
        self.__chunks = collections.defaultdict(set)
        self.__min_heap = []

    def join(self, ownedChunks):
        """
        :type ownedChunks: List[int]
        :rtype: int
        """
        if self.__min_heap:
            userID = heapq.heappop(self.__min_heap)
        else:
            userID = len(self.__users)+1
            self.__users.append(set())
        self.__users[userID-1] = set(ownedChunks)
        self.__lookup.add(userID)
        for c in ownedChunks:
            self.__chunks[c].add(userID)
        return userID

    def leave(self, userID):
        """
        :type userID: int
        :rtype: None
        """
        if userID not in self.__lookup:
            return
        for c in self.__users[userID-1]:
            self.__chunks[c].remove(userID)
        self.__lookup.remove(userID)
        self.__users[userID-1] = []
        heapq.heappush(self.__min_heap, userID)

    def request(self, userID, chunkID):
        """
        :type userID: int
        :type chunkID: int
        :rtype: List[int]
```

```
    """
    result = sorted(self.__chunks[chunkID])
    if not result:
        return
    self.__users[userID-1].add(chunkID)
    self.__chunks[chunkID].add(userID)
    return result
```

least-number-of-unique-integers-after-k-removals.py

```python
# least-number-of-unique-integers-after-k-removals is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findLeastNumOfUniqueInts(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: int
        """
        count = collections.Counter(arr)
        result, count_count = len(count), collections.Counter(count.itervalues())
        for c in xrange(1, len(arr)+1):
            if k < c*count_count[c]:
                result -= k//c
                break
            k -= c*count_count[c]
            result -= count_count[c]
        return result
```

distribute-candies-to-people.py

```
# DESC
# We distribute some number of candies, to a row of n = num_people people in the f
# ollowing way:
# Then, we go back to the start of the row, giving n + 1 candies to the first pers
# on, n + 2 candies to the second person, and so on until we give 2 * n candies to
#  the last person.
# Constraints:
# Example 2:
# Return an array (of length num_people and sum candies) that represents the final
#  distribution of candies.
# We then give 1 candy to the first person, 2 candies to the second person, and so
#  on until we give n candies to the last person.
# Example 1:
# This process repeats (with us giving one more candy each time, and moving to the
#  start of the row after we reach the end) until we run out of candies.  The last
#  person will receive all of our remaining candies (not necessarily one more than
#  the previous gift).

# NOTE
# 1 <= num_people <= 1000
# 1 <= candies <= 10^9

# EXAMPLE
# Input: candies = 7, num_people = 4
# Output: [1,2,3,1]
# Explanation:
# On the first t
# urn, ans[0] += 1, and the array is [1,0,0,0].
# On the second turn, ans[1] += 2, a
# nd the array is [1,2,0,0].
# On the third turn, ans[2] += 3, and the array is [1,2
# ,3,0].
# On the fourth turn, ans[3] += 1 (because there is only one candy left), a
# nd the final array is [1,2,3,1].
# Input: candies = 10, num_people = 3
# Output: [5,2,3]
# Explanation:
# On the first t
# urn, ans[0] += 1, and the array is [1,0,0].
# On the second turn, ans[1] += 2, and
#  the array is [1,2,0].
# On the third turn, ans[2] += 3, and the array is [1,2,3].
#
# On the fourth turn, ans[0] += 4, and the final array is [5,2,3].

# Time:  O(n + logc), c is the number of candies
# Space: O(1)

class Solution(object):
    def distributeCandies(self, candies, num_people):
        """
        :type candies: int
        :type num_people: int
        :rtype: List[int]
        """
        # find max integer p s.t. sum(1 + 2 + ... + p) <= C
        # => remaining : 0 <= C-(1+p)*p/2 < p+1
        # => -2p-2 < p^2+p-2C <= 0
```

```
        # => 2C+1/4 < (p+3/2)^2 and (p+1/2)^2 <= 2C+1/4
        # => sqrt(2C+1/4)-3/2 < p <= sqrt(2C+1/4)-1/2
        # => p = floor(sqrt(2C+1/4)-1/2)
        p = int((2*candies + 0.25)**0.5 - 0.5)
        remaining = candies - (p+1)*p//2
        rows, cols = divmod(p, num_people)

        result = [0]*num_people
        for i in xrange(num_people):
            result[i] = (i+1)*(rows+1) + (rows*(rows+1)//2)*num_people if i < cols else \
                        (i+1)*rows + ((rows-1)*rows//2)*num_people
        result[cols] += remaining
        return result


# Time:  O(n + logc), c is the number of candies
# Space: O(1)
class Solution2(object):
    def distributeCandies(self, candies, num_people):
        """
        :type candies: int
        :type num_people: int
        :rtype: List[int]
        """
        # find max integer p s.t. sum(1 + 2 + ... + p) <= C
        left, right = 1, candies
        while left <= right:
            mid = left + (right-left)//2
            if not ((mid <= candies*2 // (mid+1))):
                right = mid-1
            else:
                left = mid+1
        p = right
        remaining = candies - (p+1)*p//2
        rows, cols = divmod(p, num_people)

        result = [0]*num_people
        for i in xrange(num_people):
            result[i] = (i+1)*(rows+1) + (rows*(rows+1)//2)*num_people if i < cols else \
                        (i+1)*rows + ((rows-1)*rows//2)*num_people
        result[cols] += remaining
        return result


# Time:  O(sqrt(c)), c is the number of candies
# Space: O(1)
class Solution3(object):
    def distributeCandies(self, candies, num_people):
        """
        :type candies: int
        :type num_people: int
        :rtype: List[int]
        """
        result = [0]*num_people
        i = 0
        while candies != 0:
            result[i % num_people] += min(candies, i+1)
            candies -= min(candies, i+1)
            i += 1
        return result
```

3sum.py

```
# 3sum is not found.
# 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
```

longest-subarray-of-1s-after-deleting-one-element.py

```python
# longest-subarray-of-1s-after-deleting-one-element is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def longestSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        count, left = 0, 0
        for right in xrange(len(nums)):
            count += (nums[right] == 0)
            if count >= 2:
                count -= (nums[left] == 0)
                left += 1
        return (right-left+1)-1


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def longestSubarray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, count, left = 0, 0, 0
        for right in xrange(len(nums)):
            count += (nums[right] == 0)
            while count >= 2:
                count -= (nums[left] == 0)
                left += 1
            result = max(result, right-left+1)
        return result-1
```

minimum-cost-tree-from-leaf-values.py

```
# DESC
# Example 1:
# Constraints:
# Given an array arr of positive integers, consider all binary trees such that:
# 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.

# NOTE
# 2 <= arr.length <= 40
# Each node has either 0 or 2 children;
# 1 <= arr[i] <= 15
# It is guaranteed that the answer fits into a 32-bit signed integer (ie. it is le
# ss than 2^31).
# The value of each non-leaf node is equal to the product of the largest leaf valu
# e in its left and right subtree respectively.
# 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.)

# EXAMPLE
# 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

# 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
# DESC
# Constraints:
# Example 2:
#
# Input:
# Example 1:
#
# Input:
# Given a string s, find the longest palindromic subsequence's length in s. You ma
# y assume that the maximum length of s is 1000.

# NOTE
# 1 <= s.length <= 1000
# s consists only of lowercase English letters.

# EXAMPLE
# "cbbd"
# "bbbab"

# 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]
```

ip-to-cidr.py

```
# ip-to-cidr is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def ipToCIDR(self, ip, n):
        """
        :type ip: str
        :type n: int
        :rtype: List[str]
        """
        def ipToInt(ip):
            result = 0
            for i in ip.split('.'):
                result = 256 * result + int(i)
            return result

        def intToIP(n):
            return ".".join(str((n >> i) % 256) \
                            for i in (24, 16, 8, 0))

        start = ipToInt(ip)
        result = []
        while n:
            mask = max(33-(start & ~(start-1)).bit_length(), \
                       33-n.bit_length())
            result.append(intToIP(start) + '/' + str(mask))
            start += 1 << (32-mask)
            n -= 1 << (32-mask)
        return result
```

split-array-largest-sum.py

```
# DESC
# Examples:
# Note:
#
# If n is the length of array, assume the following constraints are satisfied:
# Given an array which consists of non-negative integers and an integer m, you can
#  split the array into m non-empty continuous subarrays. Write an algorithm to mi
# nimize the largest sum among these m subarrays.

# NOTE
# 1  m  min(50, n)
# 1  n  1000

# EXAMPLE
# Input:
# nums = [7,2,5,10,8]
# m = 2
#
# Output:
# 18
#
# Explanation:
# There are four ways t
# o split nums into two subarrays.
# The best way is to split it into [7,2,5] and [1
# 0,8],
# where the largest sum among the two subarrays is only 18.

# Time:  O(nlogs), s is the sum of nums
# Space: O(1)

# Minimize the largest subarray sum.
class Solution(object):
    def splitArray(self, nums, m):
        """
        :type nums: List[int]
        :type m: int
        :rtype: int
        """
        def canSplit(nums, m, s):
            cnt, curr_sum = 1, 0
            for num in nums:
                curr_sum += num
                if curr_sum > s:
                    curr_sum = num
                    cnt += 1
            return cnt <= m

        left, right = max(nums), sum(nums)
        while left <= right:
            mid = left + (right - left) / 2
            if canSplit(nums, m, mid):
                right = mid - 1
            else:
                left = mid + 1
        return left
```

```python
# Time:  O(nlogs), s is the sum of nums
# Space: O(1)

# Maximize the smallest subarray sum.
class SolutionFindMaxMin(object):
    def splitArray(self, nums, m):
        """
        :type nums: List[int]
        :type m: int
        :rtype: int
        """
        def canSplit(nums, m, s):
            cnt, curr_sum = 1, 0
            for num in nums:
                curr_sum += num
                if curr_sum > s:
                    curr_sum = 0
                    cnt += 1
            return cnt <= m

        left, right = min(nums), sum(nums)
        while left <= right:
            mid = left + (right - left) / 2
            x = canSplit(nums, m, mid)
            if x:
                right = mid - 1
            else:
                left = mid + 1
        return left


# Time:  O(nlogs), s is the sum of nums
# Space: O(1)

# Maximize the smallest subarray sum.
class SolutionFindMaxMin2(object):
    def splitArray(self, nums, m):
        """
        :type nums: List[int]
        :type m: int
        :rtype: int
        """
        def canSplit(nums, m, s):
            cnt, curr_sum = 0, 0
            for num in nums:
                curr_sum += num
                if curr_sum >= s:
                    curr_sum = 0
                    cnt += 1
            return cnt < m

        left, right = min(nums), sum(nums)
        for i in xrange(left, right+1):
            canSplit(nums, m, i)
        while left <= right:
            mid = left + (right - left) / 2
            x = canSplit(nums, m, mid)
            if x:
                right = mid - 1
            else:
```

```
        left = mid + 1
    return left - 1
```

dinner-plate-stacks.py

```python
# dinner-plate-stacks is not found.
# Time:  push: O(logn)
#        pop:  O(1), amortized
#        popAtStack: O(logn)
# Space: O(n * c)

import heapq


class DinnerPlates(object):

    def __init__(self, capacity):
        """
        :type capacity: int
        """
        self.__stks = []
        self.__c = capacity
        self.__min_heap = []

    def push(self, val):
        """
        :type val: int
        :rtype: None
        """
        if self.__min_heap:
            l = heapq.heappop(self.__min_heap)
            if l < len(self.__stks):
                self.__stks[l].append(val)
                return
            self.__min_heap = []  # nothing is valid in min heap
        if not self.__stks or len(self.__stks[-1]) == self.__c:
            self.__stks.append([])
        self.__stks[-1].append(val)

    def pop(self):
        """
        :rtype: int
        """
        while self.__stks and not self.__stks[-1]:
            self.__stks.pop()
        if not self.__stks:
            return -1
        return self.__stks[-1].pop()

    def popAtStack(self, index):
        """
        :type index: int
        :rtype: int
        """
        if index >= len(self.__stks) or not self.__stks[index]:
            return -1
        heapq.heappush(self.__min_heap, index)
        return self.__stks[index].pop()
```

largest-rectangle-in-histogram.py

```python
# DESC
# Given n non-negative integers representing the histogram's bar height where the
# width of each bar is 1, find the area of largest rectangle in the histogram.
# Example:
# The largest rectangle is shown in the shaded area, which has area = 10 unit.
# Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

# NOTE
#

# EXAMPLE
# Input: [2,1,5,6,2,3]
# Output: 10

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

class Solution(object):
    # @param height, a list of integer
    # @return an integer
    def largestRectangleArea(self, height):
        increasing, area, i = [], 0, 0
        while i <= len(height):
            if not increasing or (i < len(height) and height[i] > height[increasing[-1]]):
                increasing.append(i)
                i += 1
            else:
                last = increasing.pop()
                if not increasing:
                    area = max(area, height[last] * i)
                else:
                    area = max(area, height[last] * (i - increasing[-1] - 1 ))
        return area
```

relative-sort-array.py

```
# relative-sort-arra is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def relativeSortArray(self, arr1, arr2):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :rtype: List[int]
        """
        lookup = {v: i for i, v in enumerate(arr2)}
        return sorted(arr1, key=lambda i: lookup.get(i, len(arr2)+i))
```

expressive-words.py

```
# DESC
# Sometimes people repeat letters to represent extra feeling, such as "hello" -> "
# heeellooo", "hi" -> "hiiii".  In these strings like "heeellooo", we have groups
# of adjacent letters that are all the same:  "h", "eee", "ll", "ooo".
# For example, starting with "hello", we could do an extension on the group "o" to
#  get "hellooo", but we cannot get "helloo" since the group "oo" has size less th
# an 3.  Also, we could do another extension like "ll" -> "lllll" to get "hellllo
# oo".  If S = "hellllllooo", then the query word "hello" would be stretchy because
#  of these two extension operations: query = "hello" -> "hellooo" -> "hellllllooo"
#  = S.
# For some given string S, a query word is stretchy if it can be made to be equal
# to S by any number of applications of the following extension operation: choose
# a group consisting of characters c, and add some number of characters c to the g
# roup so that the size of the group is 3 or more.
# Given a list of query words, return the number of words that are stretchy.
# Constraints:

# NOTE
# S and all words in words consist only of lowercase letters
# 0 <= len(words[i]) <= 100.
# 0 <= len(words) <= 100.
# 0 <= len(S) <= 100.

# EXAMPLE
# Example:
# Input:
# S = "heeellooo"
# words = ["hello", "hi", "helo"]
# Output: 1
# Expla
# nation:
# We can extend "e" and "o" in the word "hello" to get "heeellooo".
# We ca
# n't extend "helo" to get "heeellooo" because the group "ll" is not size 3 or mor
# e.

# Time:  O(n + s), n is the sum of all word lengths, s is the length of S
# Space: O(l + s), l is the max word length

import itertools


class Solution(object):
    def expressiveWords(self, S, words):
        """
        :type S: str
        :type words: List[str]
        :rtype: int
        """
        # Run length encoding
        def RLE(S):
            return itertools.izip(*[(k, len(list(grp)))
                                    for k, grp in itertools.groupby(S)])

        R, count = RLE(S)
        result = 0
        for word in words:
            R2, count2 = RLE(word)
```

```
    if R2 != R:
        continue
    result += all(c1 >= max(c2, 3) or c1 == c2
                  for c1, c2 in itertools.izip(count, count2))
return result
```

intersection-of-two-arrays.py

```
# DESC
# Example 2:
# Note:
# Given two arrays, write a function to compute their intersection.
# Example 1:

# NOTE
# The result can be in any order.
# Each element in the result must be unique.

# EXAMPLE
# Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
# Output: [9,4]
# Input: nums1 = [1,2,2,1], nums2 = [2,2]
# Output: [2]

# Time:  O(m + n)
# Space: O(min(m, n))

class Solution(object):
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        if len(nums1) > len(nums2):
            return self.intersection(nums2, nums1)

        lookup = set()
        for i in nums1:
            lookup.add(i)

        res = []
        for i in nums2:
            if i in lookup:
                res += i,
                lookup.discard(i)

        return res

    def intersection2(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        return list(set(nums1) & set(nums2))


# Time:  O(max(m, n) * log(max(m, n)))
# Space: O(1)
# Binary search solution.
class Solution2(object):
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
```

```
        :rtype: List[int]
        """
        if len(nums1) > len(nums2):
            return self.intersection(nums2, nums1)

        def binary_search(compare, nums, left, right, target):
            while left < right:
                mid = left + (right - left) / 2
                if compare(nums[mid], target):
                    right = mid
                else:
                    left = mid + 1
            return left

        nums1.sort(), nums2.sort()

        res = []
        left = 0
        for i in nums1:
            left = binary_search(lambda x, y: x >= y, nums2, left, len(nums2), i)
            if left != len(nums2) and nums2[left] == i:
                res += i,
                left = binary_search(lambda x, y: x > y, nums2, left, len(nums2), i)

        return res


# Time:  O(max(m, n) * log(max(m, n)))
# Space: O(1)
# Two pointers solution.
class Solution3(object):
    def intersection(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        nums1.sort(), nums2.sort()
        res = []

        it1, it2 = 0, 0
        while it1 < len(nums1) and it2 < len(nums2):
            if nums1[it1] < nums2[it2]:
                it1 += 1
            elif nums1[it1] > nums2[it2]:
                it2 += 1
            else:
                if not res or res[-1] != nums1[it1]:
                    res += nums1[it1],
                it1 += 1
                it2 += 1

        return res
```

closest-divisors.py

```python
# closest-divisors is not found.
# 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
        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
```

last-moment-before-all-ants-fall-out-of-a-plank.py

```python
# last-moment-before-all-ants-fall-out-of-a-plank is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def getLastMoment(self, n, left, right):
        """
        :type n: int
        :type left: List[int]
        :type right: List[int]
        :rtype: int
        """
        return max(max(left or [0]), n-min(right or [n]))
```

powerful-integers.py

```
# DESC
# Example 2:
# Note:
# Return a list of all powerful integers that have value less than or equal to bound.
# Example 1:
# Given two positive integers x and y, an integer is powerful if it is equal to x^
# i + y^j for some integers i >= 0 and j >= 0.
# bound
# You may return the answer in any order.  In your answer, each value should occur
#  at most once.

# NOTE
# 1 <= x <= 100
# 0 <= bound <= 10^6
# 1 <= y <= 100

# EXAMPLE
# Input: x = 2, y = 3, bound = 10
# Output: [2,3,4,5,7,9,10]
# Explanation:
# 2 = 2^0 +
#  3^0
# 3 = 2^1 + 3^0
# 4 = 2^0 + 3^1
# 5 = 2^1 + 3^1
# 7 = 2^2 + 3^1
# 9 = 2^3 + 3^0
# 10 =
# 2^0 + 3^2
# Input: x = 3, y = 5, bound = 15
# Output: [2,4,6,8,10,14]

# Time:  O((logn)^2), n is the bound
# Space: O(r), r is the size of the result

import math


class Solution(object):
    def powerfulIntegers(self, x, y, bound):
        """
        :type x: int
        :type y: int
        :type bound: int
        :rtype: List[int]
        """
        result = set()
        log_x = int(math.floor(math.log(bound) / math.log(x)))+1 if x != 1 else 1
        log_y = int(math.floor(math.log(bound) / math.log(y)))+1 if y != 1 else 1
        pow_x = 1
        for i in xrange(log_x):
            pow_y = 1
            for j in xrange(log_y):
                val = pow_x + pow_y
                if val <= bound:
                    result.add(val)
                pow_y *= y
            pow_x *= x
```

```
    return list(result)
```

minimum-moves-to-equal-array-elements-ii.py

```
# DESC
# Example:
# You may assume the array's length is at most 10,000.
# Given a non-empty integer array, find the minimum number of moves required to ma
# ke all array elements equal, where a move is incrementing a selected element by
# 1 or decrementing a selected element by 1.

# NOTE
#

# 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)
```

4sum-ii.py

```
# 4sum-ii is not found.
# Time:  O(n^2)
# Space: O(n^2)

import collections


class Solution(object):
    def fourSumCount(self, A, B, C, D):
        """
        :type A: List[int]
        :type B: List[int]
        :type C: List[int]
        :type D: List[int]
        :rtype: int
        """
        A_B_sum = collections.Counter(a+b for a in A for b in B)
        return sum(A_B_sum[-c-d] for c in C for d in D)
```

flatten-2d-vector.py

```python
# flatten-2d-vector is not found.
# Time:  O(1)
# Space: O(1)

from collections import deque


class Vector2D(object):

    def __init__(self, vec2d):
        """
        Initialize your data structure here.
        :type vec2d: List[List[int]]
        """
        self.stack = deque((len(v), iter(v)) for v in vec2d if v)

    def next(self):
        """
        :rtype: int
        """
        length, iterator = self.stack.popleft()
        if length > 1:
            self.stack.appendleft((length-1, iterator))
        return next(iterator)

    def hasNext(self):
        """
        :rtype: bool
        """
        return bool(self.stack)
```

minimum-swaps-to-make-sequences-increasing.py

```
# DESC
# Given A and B, return the minimum number of swaps to make both sequences strictl
# y increasing.  It is guaranteed that the given input always makes it possible.
# Note:
# We are allowed to swap elements A[i] and B[i].  Note that both elements are in t
# he same index position in their respective sequences.
# At the end of some number of swaps, A and B are both strictly increasing.  (A se
# quence is strictly increasing if and only if A[0] < A[1] < A[2] < ... < A[A.leng
# th - 1].)
# We have two integer sequences A and B of the same non-zero length.

# NOTE
# A[i], B[i] are integer values in the range [0, 2000].
# A, B are arrays with the same length, and that length will be in the range [1, 1000].

# EXAMPLE
# Example:
# Input: A = [1,3,5,4], B = [1,2,3,7]
# Output: 1
# Explanation:
# Swap A[3] a
# nd B[3].  Then the sequences are:
# A = [1, 3, 5, 7] and B = [1, 2, 3, 4]
# which ar
# e both strictly increasing.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minSwap(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        dp_no_swap, dp_swap = [0]*2, [1]*2
        for i in xrange(1, len(A)):
            dp_no_swap[i%2], dp_swap[i%2] = float("inf"), float("inf")
            if A[i-1] < A[i] and B[i-1] < B[i]:
                dp_no_swap[i%2] = min(dp_no_swap[i%2], dp_no_swap[(i-1)%2])
                dp_swap[i%2] = min(dp_swap[i%2], dp_swap[(i-1)%2]+1)
            if A[i-1] < B[i] and B[i-1] < A[i]:
                dp_no_swap[i%2] = min(dp_no_swap[i%2], dp_swap[(i-1)%2])
                dp_swap[i%2] = min(dp_swap[i%2], dp_no_swap[(i-1)%2]+1)
        return min(dp_no_swap[(len(A)-1)%2], dp_swap[(len(A)-1)%2])
```

robot-return-to-origin.py

```
# DESC
# Example 2:
# There is a robot starting at position (0, 0), the origin, on a 2D plane. Given a
#  sequence of its moves, judge if this robot ends up at (0, 0) after it completes
#  its moves.
# The move sequence is represented by a string, and the character moves[i] represe
# nts its ith move. Valid moves are R (right), L (left), U (up), and D (down). If
# the robot returns to the origin after it finishes all of its moves, return true.
#  Otherwise, return false.
# Note: The way that the robot is "facing" is irrelevant. "R" will always make the
#  robot move to the right once, "L" will always make it move left, etc. Also, ass
# ume that the magnitude of the robot's movement is the same for each move.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: "LL"
# Output: false
# Explanation: The robot moves left twice. It ends up tw
# o "moves" to the left of the origin. We return false because it is not at the or
# igin at the end of its moves.
# Input: "UD"
# Output: true
# Explanation: The robot moves up once, and then down on
# ce. All moves have the same magnitude, so it ended up at the origin where it sta
# rted. Therefore, we return true.

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def judgeCircle(self, moves):
        """
        :type moves: str
        :rtype: bool
        """
        count = collections.Counter(moves)
        return count['L'] == count['R'] and count['U'] == count['D']


# Time:  O(n)
# Space: O(1)
class Solution(object):
    def judgeCircle(self, moves):
        """
        :type moves: str
        :rtype: bool
        """
        v, h = 0, 0
        for move in moves:
            if move == 'U':
                v += 1
            elif move == 'D':
```

```
            v -= 1
        elif move == 'R':
            h += 1
        elif move == 'L':
            h -= 1
    return v == 0 and h == 0
```

distinct-subsequences.py

```
# DESC
# It's guaranteed the answer fits on a 32-bit signed integer.
# Given a string S and a string T, count the number of distinct subsequences of S
# which equals T.
# Example 2:
# Example 1:
# A subsequence of a string is a new string which is formed from the original stri
# ng by deleting some (can be none) of the characters without disturbing the relat
# ive positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE
# " while "AEC" is not).
# "ACE"

# NOTE
#

# EXAMPLE
# Input: S = "babgbag", T = "bag"
# Output: 5
# Explanation:
# As shown below, there are
#  5 ways you can generate "bag" from S.
# (The caret symbol ^ means the chosen lett
# ers)
#
# babgbag
# ^^ ^
# babgbag
# ^^    ^
# babgbag
# ^    ^^
# babgbag
#   ^  ^^
# babgbag
#     ^
# ^^
# Input: S = "rabbbit", T = "rabbit"
# Output: 3
# Explanation:
# As shown below, there
# are 3 ways you can generate "rabbit" from S.
# (The caret symbol ^ means the chose
# n letters)
#
# rabbbit
# ^^^^ ^^
# rabbbit
# ^^ ^^^^
# rabbbit
# ^^^ ^^^

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

class Solution(object):
    # @return an integer
    def numDistinct(self, S, T):
        ways = [0 for _ in xrange(len(T) + 1)]
        ways[0] = 1
```

```
for S_char in S:
    for j, T_char in reversed(list(enumerate(T))):
        if S_char == T_char:
            ways[j + 1] += ways[j]
return ways[len(T)]
```

smallest-subsequence-of-distinct-characters.py

```
# DESC
# Constraints:
# Return the lexicographically smallest subsequence of text that contains all the
# distinct characters of text exactly once.
# Example 4:
# Example 2:
# Example 3:
# Example 1:
# Note: This question is the same as 316: https://leetcode.com/problems/remove-dup
# licate-letters/

# NOTE
# text consists of lowercase English letters.
# 1 <= text.length <= 1000

# EXAMPLE
# Input: "cdadabcc"
# Output: "adbc"
# Input: "abcd"
# Output: "abcd"
# Input: "ecbacba"
# Output: "eacb"
# Input: "leetcode"
# Output: "letcod"

# 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)
```

string-to-integer-atoi.py

```python
# string-to-integer-atoi is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def myAtoi(self, str):
        """
        :type str: str
        :rtype: int
        """
        INT_MAX =  2147483647
        INT_MIN = -2147483648
        result = 0

        if not str:
            return result

        i = 0
        while i < len(str) and str[i].isspace():
            i += 1

        if len(str) == i:
            return result

        sign = 1
        if str[i] == "+":
            i += 1
        elif str[i] == "-":
            sign = -1
            i += 1

        while i < len(str) and '0' <= str[i] <= '9':
            if result > (INT_MAX - int(str[i])) / 10:
                return INT_MAX if sign > 0 else INT_MIN
            result = result * 10 + int(str[i])
            i += 1

        return sign * result
```

short-encoding-of-words.py

```
# DESC
# Example:
# Then for each index, we will recover the word by reading from the reference stri
# ng from that index until we reach a "#" character.
# What is the length of the shortest reference string S possible that encodes the
# given words?
# Given a list of words, we may encode it by writing a reference string S and a li
# st of indexes A.
# For example, if the list of words is ["time", "me", "bell"], we can write it as
# S = "time#bell#" and indexes = [0, 2, 5].
# Note:
# ["time", "me", "bell"]

# NOTE
# Each word has only lowercase letters.
# 1 <= words.length <= 2000.
# 1 <= words[i].length <= 7.

# EXAMPLE
# Input: words = ["time", "me", "bell"]
# Output: 10
# Explanation: S = "time#bell#" a
# nd indexes = [0, 2, 5].

# Time:  O(n), n is the total sum of the lengths of words
# Space: O(t), t is the number of nodes in trie

import collections
import functools


class Solution(object):
    def minimumLengthEncoding(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        words = list(set(words))
        _trie = lambda: collections.defaultdict(_trie)
        trie = _trie()

        nodes = [functools.reduce(dict.__getitem__, word[::-1], trie)
                 for word in words]

        return sum(len(word) + 1
                   for i, word in enumerate(words)
                   if len(nodes[i]) == 0)
```

car-fleet.py

```
# DESC
# The distance between these two cars is ignored - they are assumed to have the sa
# me position.
# Example 1:
# N cars are going to the same destination along a one lane road.  The destination
#  is target miles away.
# A car fleet is some non-empty set of cars driving at the same position and same
# speed.  Note that a single car is also a car fleet.
# How many car fleets will arrive at the destination?
# If a car catches up to a car fleet right at the destination point, it will still
#  be considered as one car fleet.
# A car can never pass another car ahead of it, but it can catch up to it, and dri
# ve bumper to bumper at the same speed.
# Note:
# Each car i has a constant speed speed[i] (in miles per hour), and initial positi
# on position[i] miles towards the target along the road.

# NOTE
# 0 < speed[i] <= 10 ^ 6
# 0 <= N <= 10 ^ 4
# All initial positions are different.
# 0 <= position[i] < target
# 0 < target <= 10 ^ 6

# EXAMPLE
# Input: target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]
# Output: 3
# Expla
# nation:
# The cars starting at 10 and 8 become a fleet, meeting each other at 12.
#
# The car starting at 0 doesn't catch up to any other car, so it is a fleet by its
# elf.
# The cars starting at 5 and 3 become a fleet, meeting each other at 6.
# Note
# that no other cars meet these fleets before the destination, so the answer is 3.

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

class Solution(object):
    def carFleet(self, target, position, speed):
        """
        :type target: int
        :type position: List[int]
        :type speed: List[int]
        :rtype: int
        """
        times = [float(target-p)/s for p, s in sorted(zip(position, speed))]
        result, curr = 0, 0
        for t in reversed(times):
            if t > curr:
                result += 1
                curr = t
        return result
```

find-bottom-left-tree-value.py

```
# DESC
# Example 2:
# Example 1:
# Given a binary tree, find the leftmost value in the last row of the tree.
# Note:
# You may assume the tree (i.e., the given root node) is not NULL.

# NOTE
#

# EXAMPLE
# Input:
#
#     2
#    / \
#   1   3
#
# Output:
# 1
# Input:
#
#         1
#        / \
#       2   3
#      /   / \
#     4   5   6
#        /
#
#    7
#
# Output:
# 7

# 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)
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

```
# DESC
# Given two integers n and k, return all possible combinations of k numbers out of
#  1 ... n.
# Example:

# NOTE
#

# EXAMPLE
# Input: n = 4, k = 2
# Output:
# [
#   [2,4],
#   [3,4],
#   [2,3],
#   [1,2],
#   [1,3],
#   [1,4],
# ]

# 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


# 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
```

inorder-successor-in-bst-ii.py

```
# inorder-successor-in-bst-ii is not found.
# Time:  O(h)
# Space: O(1)

# Definition for a Node.
class Node(object):
    def __init__(self, val, left, right, parent):
        self.val = val
        self.left = left
        self.right = right
        self.parent = parent


class Solution(object):
    def inorderSuccessor(self, node):
        """
        :type node: Node
        :rtype: Node
        """
        if not node:
            return None

        if node.right:
            node = node.right
            while node.left:
                node = node.left
            return node

        while node.parent and node.parent.right is node:
            node = node.parent
        return node.parent
```

meeting-scheduler.py

```python
# meeting-scheduler is not found.
# Time:  O(n) ~ O(nlogn)
# Space: O(n)

import heapq


class Solution(object):
    def minAvailableDuration(self, slots1, slots2, duration):
        """
        :type slots1: List[List[int]]
        :type slots2: List[List[int]]
        :type duration: int
        :rtype: List[int]
        """
        min_heap = list(filter(lambda slot: slot[1] - slot[0] >= duration, slots1 + slots2))
        heapq.heapify(min_heap)  # Time: O(n)
        while len(min_heap) > 1:
            left = heapq.heappop(min_heap)  # Time: O(logn)
            right = min_heap[0]
            if left[1]-right[0] >= duration:
                return [right[0], right[0]+duration]
        return []


# Time:  O(nlogn)
# Space: O(n)
class Solution2(object):
    def minAvailableDuration(self, slots1, slots2, duration):
        """
        :type slots1: List[List[int]]
        :type slots2: List[List[int]]
        :type duration: int
        :rtype: List[int]
        """
        slots1.sort(key = lambda x: x[0])
        slots2.sort(key = lambda x: x[0])
        i, j = 0, 0
        while i < len(slots1) and j < len(slots2):
            left = max(slots1[i][0], slots2[j][0])
            right = min(slots1[i][1], slots2[j][1])
            if left+duration <= right:
                return [left, left+duration]
            if slots1[i][1] < slots2[j][1]:
                i += 1
            else:
                j += 1
        return []
```

capacity-to-ship-packages-within-d-days.py

```
# DESC
# 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.
# Example 1:
# Example 3:
# A conveyor belt has packages that must be shipped from one port to another withi
# n D days.
# Return the least weight capacity of the ship that will result in all the package
# s on the conveyor belt being shipped within D days.
# Constraints:
# Example 2:

# NOTE
# 1 <= D <= weights.length <= 50000
# 1 <= weights[i] <= 500

# EXAMPLE
# 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
# 2n
# d day: 2, 4
# 3rd day: 1, 4
# Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5
# Output: 15
# Explanation:
# A ship c
# apacity of 15 is the minimum to ship all the packages in 5 days like this:
# 1st d
# ay: 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 sp
# litting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is
# not allowed.
# 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

# 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
```

cheapest-flights-within-k-stops.py

```
# DESC
# Now given all the cities and flights, together with starting city src and the de
# stination dst, your task is to find the cheapest price from src to dst with up t
# o k stops. If there is no such route, output -1.
# Constraints:
# There are n cities connected by m flights. Each flight starts from city u and ar
# rives at v with a price w.

# NOTE
# The number of nodes n will be in range [1, 100], with nodes labeled from 0 to n - 1.
# k is in the range of [0, n - 1].
# There will not be any duplicated flights or self cycles.
# The price of each flight will be in the range [1, 10000].
# The format of each flight will be (src, dst, price).
# The size of flights will be in range [0, n * (n - 1) / 2].

# EXAMPLE
# Example 1:
# Input:
# n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
# src = 0, dst =
#  2, k = 1
# Output: 200
# Explanation:
# The graph looks like this:
#
#
# The cheapest pr
# ice from city 0 to city 2 with at most 1 stop costs 200, as marked red in the pi
# cture.
# Example 2:
# Input:
# n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
# src = 0, dst =
#  2, k = 0
# Output: 500
# Explanation:
# The graph looks like this:
#
#
# The cheapest pr
# ice from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the p
# icture.

# Time:  O((|E| + |V|) * log|V|) = O(|E| * log|V|),
#        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 heapq


class Solution(object):
    def findCheapestPrice(self, n, flights, src, dst, K):
        """
        :type n: int
        :type flights: List[List[int]]
        :type src: int
```

```
        :type dst: int
        :type K: int
        :rtype: int
        """
        adj = collections.defaultdict(list)
        for u, v, w in flights:
            adj[u].append((v, w))
        best = collections.defaultdict(lambda: collections.defaultdict(lambda: float("inf")))
        min_heap = [(0, src, K+1)]
        while min_heap:
            result, u, k = heapq.heappop(min_heap)
            if k < 0 or best[u][k] < result:
                continue
            if u == dst:
                return result
            for v, w in adj[u]:
                if result+w < best[v][k-1]:
                    best[v][k-1] = result+w
                    heapq.heappush(min_heap, (result+w, v, k-1))
        return -1
```

game-of-life.py

```
# DESC
# According to the Wikipedia's article: "The Game of Life, also known simply as Li
# fe, is a cellular automaton devised by the British mathematician John Horton Con
# way in 1970."
# Example:
# 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 simult
# aneously to every cell in the current state, where births and deaths occur simul
# taneously.
# Follow up:
# 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, diagon
# al) using the following four rules (taken from the above Wikipedia article):

# NOTE
# Any live cell with more than three live neighbors dies, as if by over-population..
# Any live cell with fewer than two live neighbors dies, as if caused by under-pop
# ulation.
# Any dead cell with exactly three live neighbors becomes a live cell, as if by re
# production.
# Any live cell with two or three live neighbors lives on to the next generation.
# In this question, we represent the board using a 2D array. In principle, the boa
# rd is infinite, which would cause problems when the active area encroaches the b
# order of the array. How would you address these problems?
# 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.

# 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]
# ]

# 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.
```

minimum-cost-to-make-at-least-one-valid-path-in-a-grid.py

```
# minimum-cost-to-make-at-least-one-valid-path-in-a-grid is not found.
# Time:  O(m * n)
# Space: O(m * n)

import collections


# A* Search Algorithm without heap
class Solution(object):
    def minCost(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        def a_star(grid, b, t):
            R, C = len(grid), len(grid[0])
            f, dh = 0, 1
            closer, detour = [b], []
            lookup = set()
            while closer or detour:
                if not closer:
                    f += dh
                    closer, detour = detour, closer
                b = closer.pop()
                if b == t:
                    return f
                if b in lookup:
                    continue
                lookup.add(b)
                for nd, (dr, dc) in enumerate(directions, 1):
                    nb = (b[0]+dr, b[1]+dc)
                    if not (0 <= nb[0] < R and 0 <= nb[1] < C and nb not in lookup):
                        continue
                    (closer if nd == grid[b[0]][b[1]] else detour).append(nb)
            return -1

        return a_star(grid, (0, 0), (len(grid)-1, len(grid[0])-1))


# Time:  O(m * n)
# Space: O(m * n)
#  0-1 bfs solution
class Solution2(object):
    def minCost(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        R, C = len(grid), len(grid[0])
        b, t = (0, 0), (R-1, C-1)
        dq = collections.deque([(b, 0)])
        lookup = {b: 0}
        while dq:
            b, d = dq.popleft()
            if b == t:
                return d
```

```
        if lookup[b] < d:
            continue
        for nd, (dr, dc) in enumerate(directions, 1):
            nb = (b[0]+dr, b[1]+dc)
            cost = 1 if nd != grid[b[0]][b[1]] else 0
            if not (0 <= nb[0] < R and 0 <= nb[1] < C and
                    (nb not in lookup or lookup[nb] > d+cost)):
                continue
            lookup[nb] = d+cost
            if not cost:
                dq.appendleft((nb, d))
            else:
                dq.append((nb, d+cost))
    return -1  # never reach here
```

online-stock-span.py

```python
# DESC
# Write a class StockSpanner which collects daily price quotes for some stock, and
#  returns the span of that stock's price for the current day.
# Note:
# The span of the stock's price today is defined as the maximum number of consecut
# ive days (starting from today and going backwards) for which the price of the st
# ock was less than or equal to today's price.
# For example, if the price of a stock over the next 7 days were [100, 80, 60, 70,
#  60, 75, 85], then the stock spans would be [1, 1, 1, 2, 1, 4, 6].
# Example 1:

# NOTE
# 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.
# Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.
# The total time limit for this problem has been reduced by 75% for C++, and 50% f
# or all other languages.

# EXAMPLE
# Input: ["StockSpanner","next","next","next","next","next","next","next"], [[],[1
# 00],[80],[60],[70],[60],[75],[85]]
# Output: [null,1,1,1,2,1,4,6]
# Explanation:
# Fi
# rst, 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.ne
# xt(70) is called and returns 2,
# S.next(60) is called and returns 1,
# S.next(75) i
# s called and returns 4,
# S.next(85) is called and returns 6.
#
# Note that (for exam
# ple) S.next(75) returned 4, because the last 4 prices
# (including today's price o
# f 75) were less than or equal to today's price.

# 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
```

find-common-characters.py

```
# DESC
# Note:
# Given an array A of strings made only from lowercase letters, return a list of a
# ll characters that show up in all strings within the list (including duplicates)
# .  For example, if a character occurs 3 times in all strings but not 4 times, yo
# u need to include that character three times in the final answer.
# You may return the answer in any order.
# Example 1:
# Example 2:

# NOTE
# 1 <= A[i].length <= 100
# A[i][j] is a lowercase letter
# 1 <= A.length <= 100

# EXAMPLE
# Input: ["cool","lock","cook"]
# Output: ["c","o"]
# Input: ["bella","label","roller"]
# Output: ["e","l","l"]

# Time:  O(n * l)
# Space: O(1)

import collections


class Solution(object):
    def commonChars(self, A):
        """
        :type A: List[str]
        :rtype: List[str]
        """
        result = collections.Counter(A[0])
        for a in A:
            result &= collections.Counter(a)
        return list(result.elements())
```

interval-list-intersections.py

```
# DESC
# Note:
# (Formally, a closed interval [a, b] (with a <= b) denotes the set of real number
# s 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].)
# Return the intersection of these two interval lists.
# Given two lists of closed intervals, each list of intervals is pairwise disjoint
#  and in sorted order.
# Example 1:

# NOTE
# 0 <= A.length < 1000
# 0 <= B.length < 1000
# 0 <= A[i].start, A[i].end, B[i].start, B[i].end < 10^9

# EXAMPLE
# Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
# Ou
# tput: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]

# 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
# DESC
# 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.

# NOTE
#

# EXAMPLE
#

# 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
```

consecutive-numbers-sum.py

```
# DESC
# Note: 1 <= N <= 10 ^ 9.
# Given a positive integer N, how many ways can we write it as a sum of consecutiv
# e positive integers?
# Example 1:
# Example 2:
# Example 3:

# NOTE
#

# EXAMPLE
# Input: 5
# Output: 2
# Explanation: 5 = 5 = 2 + 3
# Input: 15
# Output: 4
# Explanation: 15 = 15 = 8 + 7 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5
# Input: 9
# Output: 3
# Explanation: 9 = 9 = 4 + 5 = 2 + 3 + 4

# Time:  O(sqrt(n))
# Space: O(1)

class Solution(object):
    def consecutiveNumbersSum(self, N):
        """
        :type N: int
        :rtype: int
        """
        # x + x+1 + x+2 + ... + x+l-1 = N = 2^k * M, where M is odd
        # => l*x + (l-1)*l/2 = 2^k * M
        # => x = (2^k * M -(l-1)*l/2)/l= 2^k * M/l - (l-1)/2 is integer
        # => l could be 2 or any odd factor of M (excluding M)
        #    s.t. x = 2^k * M/l - (l-1)/2 is integer, and also unique
        # => the answer is the number of all odd factors of M
        # if prime factorization of N is 2^k * p1^a * p2^b * ..
        # => answer is the number of all odd factors = (a+1) * (b+1) * ...
        result = 1
        while N % 2 == 0:
            N /= 2
        i = 3
        while i*i <= N:
            count = 0
            while N % i == 0:
                N /= i
                count += 1
            result *= count+1
            i += 2
        if N > 1:
            result *= 2
        return result
```

russian-doll-envelopes.py

```
# DESC
# Note:
#
# Rotation is not allowed.
# Example:
# You have a number of envelopes with widths and heights given as a pair of intege
# rs (w, h). One envelope can fit into another if and only if both the width and h
# eight of one envelope is greater than the width and height of the other envelope
# .
# What is the maximum number of envelopes can you Russian doll? (put one inside other)

# NOTE
#

# EXAMPLE
# Input: [[5,4],[6,4],[6,7],[2,3]]
# Output: 3
# Explanation: The maximum number of e
# nvelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

# Time:  O(nlogn + nlogk) = O(nlogn), k is the length of the result.
# Space: O(1)

class Solution(object):
    def maxEnvelopes(self, envelopes):
        """
        :type envelopes: List[List[int]]
        :rtype: int
        """
        def insert(target):
            left, right = 0, len(result) - 1
            while left <= right:
                mid = left + (right - left) / 2
                if result[mid] >= target:
                    right = mid - 1
                else:
                    left = mid + 1
            if left == len(result):
                result.append(target)
            else:
                result[left] = target

        result = []

        envelopes.sort(lambda x, y: y[1] - x[1] if x[0] == y[0] else \
                                    x[0] - y[0])
        for envelope in envelopes:
            insert(envelope[1])

        return len(result)
```

reach-a-number.py

```
# DESC
# Example 1:
# Return the minimum number of steps required to reach the destination.
# Example 2:
# Note:
# You are standing at position 0 on an infinite number line.  There is a goal at p
# osition target.
# On each move, you can either go left or right.  During the n-th move (starting f
# rom 1), you take n steps.

# NOTE
# target will be a non-zero integer in the range [-10^9, 10^9].

# EXAMPLE
# Input: target = 2
# Output: 3
# Explanation:
# On the first move we step from 0 to 1.
#
# On the second move we step  from 1 to -1.
# On the third move we step from -1 to 2
# .
# Input: target = 3
# Output: 2
# Explanation:
# On the first move we step from 0 to 1.
#
# On the second step we step from 1 to 3.

# Time:  O(logn)
# Space: O(1)

import math


class Solution(object):
    def reachNumber(self, target):
        """
        :type target: int
        :rtype: int
        """
        target = abs(target)
        k = int(math.ceil((-1+math.sqrt(1+8*target))/2))
        target -= k*(k+1)/2
        return k if target%2 == 0 else k+1+k%2


# Time:  O(sqrt(n))
# Space: O(1)
class Solution2(object):
    def reachNumber(self, target):
        """
        :type target: int
        :rtype: int
        """
        target = abs(target)
        k = 0
        while target > 0:
```

```
        k += 1
        target -= k
    return k if target%2 == 0 else k+1+k%2
```

number-of-ships-in-a-rectangle.py

```python
# number-of-ships-in-a-rectangle is not found.
# Time:  O(s * log(m * n)), s is the max number of ships, which is 10 in this problem
# Space: O(log(m * n))

# """
# This is Sea's API interface.
# You should not implement it, or speculate about its implementation
# """
class Sea(object):
   def hasShips(self, topRight, bottomLeft):
       """
       :type topRight: Point
                 :type bottomLeft: Point
       :rtype bool
       """
       pass

class Point(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y

class Solution(object):
    def countShips(self, sea, topRight, bottomLeft):
        """
        :type sea: Sea
        :type topRight: Point
        :type bottomLeft: Point
        :rtype: integer
        """
        result = 0
        if topRight.x >= bottomLeft.x and \
           topRight.y >= bottomLeft.y and \
           sea.hasShips(topRight, bottomLeft):
            if (topRight.x, topRight.y) == (bottomLeft.x, bottomLeft.y):
                return 1
            mid_x, mid_y = (topRight.x+bottomLeft.x)//2, (topRight.y+bottomLeft.y)//2
            result += self.countShips(sea, topRight, Point(mid_x+1, mid_y+1))
            result += self.countShips(sea, Point(mid_x, topRight.y), Point(bottomLeft.x, mid_y+1))
            result += self.countShips(sea, Point(topRight.x, mid_y), Point(mid_x+1, bottomLeft.y))
            result += self.countShips(sea, Point(mid_x, mid_y), bottomLeft)
        return result
```

most-frequent-subtree-sum.py

```
# DESC
# Examples 2
#
# Input:
# Given the root of a tree, you are asked to find the most frequent subtree sum. T
# he 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 h
# ighest frequency in any order.
# Note:
# You may assume the sum of values in any subtree is in the range of 32-bit
# signed integer.
# Examples 1
#
# Input:

# NOTE
#

# EXAMPLE
# 5
#  /  \
# 2   -5
# 5
#  /  \
# 2   -3

# 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

```
# DESC
# Example 2:
# You may return the answer in any order.  The answer is guaranteed to be unique (
# except for the order that it is in.)
# Note:
# (Here, the distance between two points on a plane is the Euclidean distance.)
# We have a list of points on the plane.  Find the K closest points to the origin
# (0, 0).
# Example 1:

# NOTE
# -10000 < points[i][0] < 10000
# -10000 < points[i][1] < 10000
# 1 <= K <= points.length <= 10000

# EXAMPLE
# 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.)
# Input: points = [[1,3],[-2,2]], K = 1
# Output: [[-2,2]]
# Explanation:
# The distanc
# e between (1, 3) and the origin is sqrt(10).
# The distance between (-2, 2) and th
# e 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]].

# 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:
                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))]
```

generate-a-string-with-characters-that-have-odd-counts.py

```python
# generate-a-string-with-characters-that-have-odd-counts is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def generateTheString(self, n):
        """
        :type n: int
        :rtype: str
        """
        result = ['a']*(n-1)
        result.append('a' if n%2 else 'b')
        return "".join(result)
```

design-linked-list.py

```
# DESC
# Example:
# Design your implementation of the linked list. You can choose to use the singly
# linked list or the doubly linked list. A node in a singly linked list should hav
# e two attributes: val and next. val is the value of the current node, and next i
# s a pointer/reference to the next node. If you want to use the doubly linked lis
# t, you will need one more attribute prev to indicate the previous node in the li
# nked list. Assume all nodes in the linked list are 0-indexed.
# Constraints:
# Implement these functions in your linked list class:

# NOTE
# 0 <= index,val <= 1000
# addAtHead(val) : Add a node of value val before the first element of the linked
# list. After the insertion, the new node will be the first node of the linked lis
# t.
# deleteAtIndex(index) : Delete the index-th node in the linked list, if the index
#  is valid.
# addAtIndex(index, val) : Add a node of value val before the index-th node in the
#  linked list. If index equals to the length of linked list, the node will be app
# ended to the end of linked list. If index is greater than the length, the node w
# ill not be inserted.
# addAtTail(val) : Append a node of value val to the last element of the linked list.
# get(index) : Get the value of the index-th node in the linked list. If the index
#  is invalid, return -1.
# Please do not use the built-in LinkedList library.
# At most 2000 calls will be made to get, addAtHead, addAtTail,  addAtIndex and de
# leteAtIndex.

# EXAMPLE
# Input:
# ["MyLinkedList","addAtHead","addAtTail","addAtIndex","get","deleteAtInde
# x","get"]
# [[],[1],[3],[1,2],[1],[1],[1]]
# Output:
# [null,null,null,null,2,null,3
# ]
#
# Explanation:
# MyLinkedList linkedList = new MyLinkedList(); // Initialize empt
# y LinkedList
# linkedList.addAtHead(1);
# linkedList.addAtTail(3);
# linkedList.addAtI
# ndex(1, 2);  // linked list becomes 1->2->3
# linkedList.get(1);            // ret
# urns 2
# linkedList.deleteAtIndex(1);  // now the linked list is 1->3
# linkedList.g
# et(1);            // returns 3

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

class Node(object):
    def __init__(self, value):
        self.val = value
        self.next = self.prev = None
```

```python
class MyLinkedList(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__head = self.__tail = Node(-1)
        self.__head.next = self.__tail
        self.__tail.prev = self.__head
        self.__size = 0

    def get(self, index):
        """
        Get the value of the index-th node in the linked list. If the index is invalid, return -1.
        :type index: int
        :rtype: int
        """
        if 0 <= index <= self.__size // 2:
            return self.__forward(0, index, self.__head.next).val
        elif self.__size // 2 < index < self.__size:
            return self.__backward(self.__size, index, self.__tail).val
        return -1

    def addAtHead(self, val):
        """
        Add a node of value val before the first element of the linked list.
        After the insertion, the new node will be the first node of the linked list.
        :type val: int
        :rtype: void
        """
        self.__add(self.__head, val)

    def addAtTail(self, val):
        """
        Append a node of value val to the last element of the linked list.
        :type val: int
        :rtype: void
        """
        self.__add(self.__tail.prev, val)

    def addAtIndex(self, index, val):
        """
        Add a node of value val before the index-th node in the linked list.
        If index equals to the length of linked list,
        the node will be appended to the end of linked list.
        If index is greater than the length, the node will not be inserted.
        :type index: int
        :type val: int
        :rtype: void
        """
        if 0 <= index <= self.__size // 2:
            self.__add(self.__forward(0, index, self.__head.next).prev, val)
        elif self.__size // 2 < index <= self.__size:
            self.__add(self.__backward(self.__size, index, self.__tail).prev, val)

    def deleteAtIndex(self, index):
        """
        Delete the index-th node in the linked list, if the index is valid.
```

```
        :type index: int
        :rtype: void
        """
        if 0 <= index <= self.__size // 2:
            self.__remove(self.__forward(0, index, self.__head.next))
        elif self.__size // 2 < index < self.__size:
            self.__remove(self.__backward(self.__size, index, self.__tail))

    def __add(self, preNode, val):
        node = Node(val)
        node.prev = preNode
        node.next = preNode.next
        node.prev.next = node.next.prev = node
        self.__size += 1

    def __remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev
        self.__size -= 1

    def __forward(self, start, end, curr):
        while start != end:
            start += 1
            curr = curr.next
        return curr

    def __backward(self, start, end, curr):
        while start != end:
            start -= 1
            curr = curr.prev
        return curr
```

concatenated-words.py

```
# DESC
# Example:
# A concatenated word is defined as a string that is comprised entirely of at leas
# t two shorter words in the given array.
# Note:

# NOTE
# The returned elements order does not matter.
# The number of elements of the given array will not exceed 10,000
# All the input string will only include lower case letters.
# The length sum of elements in the given array will not exceed 600,000.

# EXAMPLE
# Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ra
# tcatdogcat"]
#
# Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
#
# Explanation:
# "catsdogcats" can be concatenated by "cats", "dog" and "cats";
#  "dogcatsdog" ca
# n be concatenated by "dog", "cats" and "dog";
# "ratcatdogcat" can be concatenate
# d by "rat", "cat", "dog" and "cat".

# Time:  O(n * l^2)
# Space: O(n * l)

class Solution(object):
    def findAllConcatenatedWordsInADict(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        lookup = set(words)
        result = []
        for word in words:
            dp = [False] * (len(word)+1)
            dp[0] = True
            for i in xrange(len(word)):
                if not dp[i]:
                    continue

                for j in xrange(i+1, len(word)+1):
                    if j - i < len(word) and word[i:j] in lookup:
                        dp[j] = True

                if dp[len(word)]:
                    result.append(word)
                    break

        return result
```

design-circular-queue.py

```
# DESC
# One of the benefits of the circular queue is that we can make use of the spaces
# in front of the queue. In a normal queue, once the queue becomes full, we cannot
#  insert the next element even if there is a space in front of the queue. But usi
# ng the circular queue, we can use the space to store new values.
# Example:
# Design your implementation of the circular queue. The circular queue is a linear
#  data structure in which the operations are performed based on FIFO (First In Fi
# rst Out) principle and the last position is connected back to the first position
#  to make a circle. It is also called "Ring Buffer".
# Your implementation should support following operations:
# Note:

# NOTE
# isEmpty(): Checks whether the circular queue is empty or not.
# The number of operations will be in the range of [1, 1000].
# Please do not use the built-in Queue library.
# Rear: Get the last item from the queue. If the queue is empty, return -1.
# isFull(): Checks whether the circular queue is full or not.
# Front: Get the front item from the queue. If the queue is empty, return -1.
# deQueue(): Delete an element from the circular queue. Return true if the operati
# on is successful.
# All values will be in the range of [0, 1000].
# MyCircularQueue(k): Constructor, set the size of the queue to be k.
# enQueue(value): Insert an element into the circular queue. Return true if the op
# eration is successful.

# EXAMPLE
# MyCircularQueue circularQueue = new MyCircularQueue(3); // set the size to be 3
#
# circularQueue.enQueue(1);  // return true
# circularQueue.enQueue(2);  // return t
# rue
# circularQueue.enQueue(3);  // return true
# circularQueue.enQueue(4);  // retu
# rn false, the queue is full
# circularQueue.Rear();  // return 3
# circularQueue.isF
# ull();  // return true
# circularQueue.deQueue();  // return true
# circularQueue.en
# Queue(4);  // return true
# circularQueue.Rear();  // return 4

# Time:  O(1)
# Space: O(k)

class MyCircularQueue(object):

    def __init__(self, k):
        """
        Initialize your data structure here. Set the size of the queue to be k.
        :type k: int
        """
        self.__start = 0
        self.__size = 0
        self.__buffer = [0] * k
```

```
    def enQueue(self, value):
        """
        Insert an element into the circular queue. 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 deQueue(self):
        """
        Delete an element from the circular queue. 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 Front(self):
        """
        Get the front item from the queue.
        :rtype: int
        """
        return -1 if self.isEmpty() else self.__buffer[self.__start]

    def Rear(self):
        """
        Get the last item from the queue.
        :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 queue is empty or not.
        :rtype: bool
        """
        return self.__size == 0

    def isFull(self):
        """
        Checks whether the circular queue is full or not.
        :rtype: bool
        """
        return self.__size == len(self.__buffer)
```

smallest-range-ii.py

```
# DESC
# Given an array A of integers, for each integer A[i] we need to choose either x =
#  -K or x = K, and add x to A[i] (only once).
# Return the smallest possible difference between the maximum value of B and the m
# inimum value of B.
# Note:
# Example 1:
# Example 3:
# Example 2:
# After this process, we have some array B.

# NOTE
# 0 <= A[i] <= 10000
# 1 <= A.length <= 10000
# 0 <= K <= 10000

# EXAMPLE
# Input: A = [1], K = 0
# Output: 0
# Explanation: B = [1]
# Input: A = [1,3,6], K = 3
# Output: 3
# Explanation: B = [4,6,3]
# Input: A = [0,10], K = 2
# Output: 6
# Explanation: B = [2,8]

# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def smallestRangeII(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        A.sort()
        result = A[-1]-A[0]
        for i in xrange(len(A)-1):
            result = min(result,
                         max(A[-1]-K, A[i]+K) -
                         min(A[0]+K, A[i+1]-K))
        return result
```

implement-magic-dictionary.py

```
# implement-magic-dictionar is not found.
# 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)
        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)
```

peak-index-in-a-mountain-array.py

```python
# peak-index-in-a-mountain-arra is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def peakIndexInMountainArray(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        left, right = 0, len(A)
        while left < right:
            mid = left + (right-left)//2
            if A[mid] > A[mid+1]:
                right = mid
            else:
                left = mid+1
        return left
```

h-index.py

```
# DESC
# Given an array of citations (each citation is a non-negative integer) of a resea
# rcher, write a function to compute the researcher's h-index.
# Note: If there are several possible values for h, the maximum one is taken as th
# e h-index.
# Example:
# 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 paper
# s have no more than h citations each."

# NOTE
#

# EXAMPLE
# Input: citations = [3,0,6,1,5]
# Output: 3
# Explanation: [3,0,6,1,5] means the res
# earcher 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.

# 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:
                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

```
# DESC
# Values in the traversals pre and post are distinct positive integers.
# Example 1:
# Return any binary tree that matches the given preorder and postorder traversals.
# Note:

# NOTE
# pre[] and post[] are both permutations of 1, 2, ..., pre.length.
# It is guaranteed an answer exists. If there exists multiple answers, you can ret
# urn any of them.
# 1 <= pre.length == post.length <= 30

# EXAMPLE
# 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]

# 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
    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

```
# DESC
# Example 2:
# Example 1:
# Note: You may assume that n is not less than 2 and not larger than 58.
# Given a positive integer n, break it into the sum of at least two positive integ
# ers and maximize the product of those integers. Return the maximum product you c
# an get.

# NOTE
#

# EXAMPLE
# Input: 10
# Output: 36
# Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.
# Input: 2
# Output: 1
# Explanation: 2 = 1 + 1, 1 × 1 = 1.

# 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]
```

binary-number-with-alternating-bits.py

```
# DESC
# Example 4:
# Example 2:
# Example 3:
# Given a positive integer, check whether it has alternating bits: namely, if two
# adjacent bits will always have different values.
# Example 1:

# NOTE
#

# EXAMPLE
# Input: 7
# Output: False
# Explanation:
# The binary representation of 7 is: 111.
# Input: 11
# Output: False
# Explanation:
# The binary representation of 11 is: 1011.
# Input: 10
# Output: True
# Explanation:
# The binary representation of 10 is: 1010.
# Input: 5
# Output: True
# Explanation:
# The binary representation of 5 is: 101

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def hasAlternatingBits(self, n):
        """
        :type n: int
        :rtype: bool
        """
        n, curr = divmod(n, 2)
        while n > 0:
            if curr == n % 2:
                return False
            n, curr = divmod(n, 2)
        return True
```

predict-the-winner.py

```
# DESC
# Example 1:
# Constraints:
# Given an array of scores that are non-negative integers. Player 1 picks one of t
# he 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 availab
# le for the next player. This continues until all the scores have been chosen. Th
# e player with the maximum score wins.
# Example 2:
# Given an array of scores, predict whether player 1 is the winner. You can assume
#  each player plays to maximize his score.

# NOTE
# If the scores of both players are equal, then player 1 is still the winner.
# Any scores in the given array are non-negative integers and will not exceed 10,0
# 00,000.
# 1 <= length of the array <= 20.

# EXAMPLE
# Input: [1, 5, 233, 7]
# Output: True
# Explanation: Player 1 first chooses 1. Then p
# layer 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 (1
# 2), so you need to return True representing player1 can win.
# Input: [1, 5, 2]
# Output: False
# Explanation: Initially, player 1 can choose betwe
# en 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 s
# core 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.

# 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
```

toeplitz-matrix.py

```
# DESC
# M x N
# Example 2:
# Example 1:
# Note:
# Now given an M x N matrix, return True if and only if the matrix is Toeplitz.
# A matrix is Toeplitz if every diagonal from top-left to bottom-right has the sam
# e element.
# Follow up:

# NOTE
# matrix will be a 2D array of integers.
# What if the matrix is so large that you can only load up a partial row into the
# memory at once?
# matrix will have a number of rows and columns in range [1, 20].
# What if the matrix is stored on disk, and the memory is limited such that you ca
# n only load at most one row of the matrix into the memory at once?
# matrix[i][j] will be integers in range [0, 99].

# EXAMPLE
# Input:
# matrix = [
#   [1,2,3,4],
#   [5,1,2,3],
#   [9,5,1,2]
# ]
# Output: True
# Explanati
# on:
# In the above grid, the diagonals are:
# "[9]", "[5, 5]", "[1, 1, 1]", "[2, 2,
# 2]", "[3, 3]", "[4]".
# In each diagonal all elements are the same, so the answer
# is True.
# Input:
# matrix = [
#   [1,2],
#   [2,2]
# ]
# Output: False
# Explanation:
# The diagonal "[1
# , 2]" has different elements.

# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def isToeplitzMatrix(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: bool
        """
        return all(i == 0 or j == 0 or matrix[i-1][j-1] == val
                   for i, row in enumerate(matrix)
                   for j, val in enumerate(row))
```

```python
class Solution2(object):
    def isToeplitzMatrix(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: bool
        """
        for row_index, row in enumerate(matrix):
            for digit_index, digit in enumerate(row):
                if not row_index or not digit_index:
                    continue
                if matrix[row_index - 1][digit_index - 1] != digit:
                    return False
        return True
```

equal-tree-partition.py

```python
# equal-tree-partition is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def checkEqualTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        def getSumHelper(node, lookup):
            if not node:
                return 0
            total = node.val + \
                    getSumHelper(node.left, lookup) + \
                    getSumHelper(node.right, lookup)
            lookup[total] += 1
            return total

        lookup = collections.defaultdict(int)
        total = getSumHelper(root, lookup)
        if total == 0:
            return lookup[total] > 1
        return total%2 == 0 and (total/2) in lookup
```

brace-expansion.py

```
# brace-expansion is not found.
# Time:  O(p*l * log(p*l)), p is the production of all number of options
#                         , l is the length of a word
# Space: O(p*l)

import itertools


class Solution(object):
    def expand(self, S):  # nested is fine
        """
        :type S: str
        :rtype: List[str]
        """
        def form_words(options):
            words = map("".join, itertools.product(*options))
            words.sort()
            return words

        def generate_option(expr, i):
            option_set = set()
            while i[0] != len(expr) and expr[i[0]] != "}":
                i[0] += 1  # { or ,
                for option in generate_words(expr, i):
                    option_set.add(option)
            i[0] += 1  # }
            option = list(option_set)
            option.sort()
            return option

        def generate_words(expr, i):
            options = []
            while i[0] != len(expr) and expr[i[0]] not in ",}":
                tmp = []
                if expr[i[0]] not in "{,}":
                    tmp.append(expr[i[0]])
                    i[0] += 1  # a-z
                elif expr[i[0]] == "{":
                    tmp = generate_option(expr, i)
                options.append(tmp)
            return form_words(options)

        return generate_words(S, [0])


class Solution2(object):
    def expand(self, S):  # nested is fine
        """
        :type S: str
        :rtype: List[str]
        """
        def form_words(options):
            words = []
            total = 1
            for opt in options:
                total *= len(opt)
            for i in xrange(total):
                tmp = []
```

```
            for opt in reversed(options):
                i, c = divmod(i, len(opt))
                tmp.append(opt[c])
            tmp.reverse()
            words.append("".join(tmp))
        words.sort()
        return words

    def generate_option(expr, i):
        option_set = set()
        while i[0] != len(expr) and expr[i[0]] != "}":
            i[0] += 1  # { or ,
            for option in generate_words(expr, i):
                option_set.add(option)
        i[0] += 1  # }
        option = list(option_set)
        option.sort()
        return option

    def generate_words(expr, i):
        options = []
        while i[0] != len(expr) and expr[i[0]] not in ",}":
            tmp = []
            if expr[i[0]] not in "{,}":
                tmp.append(expr[i[0]])
                i[0] += 1  # a-z
            elif expr[i[0]] == "{":
                tmp = generate_option(expr, i)
            options.append(tmp)
        return form_words(options)

    return generate_words(S, [0])
```

reformat-the-string.py

```python
# reformat-the-string is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def reformat(self, s):
        """
        :type s: str
        :rtype: str
        """
        def char_gen(start, end, count):
            for c in xrange(ord(start), ord(end)+1):
                c = chr(c)
                for i in xrange(count[c]):
                    yield c
            yield ''

        count = collections.defaultdict(int)
        alpha_cnt = 0
        for c in s:
            count[c] += 1
            if c.isalpha():
                alpha_cnt += 1
        if abs(len(s)-2*alpha_cnt) > 1:
            return ""

        result = []
        it1, it2 = char_gen('a', 'z', count), char_gen('0', '9', count)
        if alpha_cnt < len(s)-alpha_cnt:
            it1, it2 = it2, it1
        while len(result) < len(s):
            result.append(next(it1))
            result.append(next(it2))
        return "".join(result)
```

remove-covered-intervals.py

```python
# DESC
# Constraints:
# After doing so, return the number of remaining intervals.
# Example 1:
# Given a list of intervals, remove all intervals that are covered by another inte
# rval in the list. Interval [a,b) is covered by interval [c,d) if and only if c <
# = a and b <= d.

# NOTE
# 1 <= intervals.length <= 1000
# intervals[i] != intervals[j] for all i != j
# 0 <= intervals[i][0] < intervals[i][1] <= 10^5

# EXAMPLE
# Input: intervals = [[1,4],[3,6],[2,8]]
# Output: 2
# Explanation: Interval [3,6] is
# covered by [2,8], therefore it is removed.

# 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
```

guess-number-higher-or-lower.py

```python
# DESC
# 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.
# You call a pre-defined API guess(int num) which returns 3 possible results (-1,
# 1, or 0):
# Every time you guess wrong, I'll tell you whether the number is higher or lower.
# Example :

# NOTE
#

# EXAMPLE
# Input: n = 10, pick = 6
# Output: 6
# -1 : My number is lower
#  1 : My number is higher
#  0 : Congrats! You got it!

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def guessNumber(self, n):
        """
        :type n: int
        :rtype: int
        """
        left, right = 1, n
        while left <= right:
            mid = left + (right - left) / 2
            if guess(mid) <= 0: # noqa
                right = mid - 1
            else:
                left = mid + 1
        return left
```

search-in-a-sorted-array-of-unknown-size.py

```python
# search-in-a-sorted-array-of-unknown-size is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def search(self, reader, target):
        """
        :type reader: ArrayReader
        :type target: int
        :rtype: int
        """
        left, right = 0, 19999
        while left <= right:
            mid = left + (right-left)//2
            response = reader.get(mid)
            if response > target:
                right = mid-1
            elif response < target:
                left = mid+1
            else:
                return mid
        return -1
```

range-sum-of-sorted-subarray-sums.py

```
# range-sum-of-sorted-subarray-sums is not found.
# Time:  O(nlog(sum(nums)))
# Space: O(n)

# binary search + sliding window solution
class Solution(object):
    def rangeSum(self, nums, n, left, right):
        """
        :type nums: List[int]
        :type n: int
        :type left: int
        :type right: int
        :rtype: int
        """
        def countUntil(nums, target):
            result, curr, left = 0, 0, 0
            for right in xrange(len(nums)):
                curr += nums[right]
                while curr > target:
                    curr -= nums[left]
                    left += 1
                result += right-left+1
            return result

        def sumUntil(nums, prefix, target):
            result, curr, total, left = 0, 0, 0, 0
            for right in xrange(len(nums)):
                curr += nums[right]
                total += nums[right]*(right-left+1)
                while curr > target:
                    curr -= nums[left]
                    total -= prefix[right+1]-prefix[(left-1)+1]
                    left += 1
                result += total
            return result

        def sumLessOrEqualTo(prefix, nums, left, right, count):
            while left <= right:
                mid = left + (right-left)//2
                if countUntil(nums, mid)-count >= 0:
                    right = mid-1
                else:
                    left = mid+1
            return sumUntil(nums, prefix, left)-left*(countUntil(nums, left)-count)

        MOD = 10**9+7
        prefix = [0]*(len(nums)+1)
        for i in xrange(len(nums)):
            prefix[i+1] = prefix[i]+nums[i]
        m, M = min(nums), sum(nums)
        return (sumLessOrEqualTo(prefix, nums, m, M, right) -
                sumLessOrEqualTo(prefix, nums, m, M, left-1))%MOD



# Time:  O(rlogr), worst: O(n^2 * logn)
# Space: O(n)
import heapq
```

```python
# heap solution
class Solution2(object):
    def rangeSum(self, nums, n, left, right):
        """
        :type nums: List[int]
        :type n: int
        :type left: int
        :type right: int
        :rtype: int
        """
        MOD = 10**9+7
        min_heap = [];
        for i, num in enumerate(nums, 1):
            heapq.heappush(min_heap, (num, i))
        result = 0
        for i in xrange(1, right+1):
            total, j = heapq.heappop(min_heap)
            if i >= left:
                result = (result+total)%MOD
            if j+1 <= n:
                heapq.heappush(min_heap, (total+nums[j], j+1))
        return result
```

valid-perfect-square.py

```
# DESC
# Example 2:
# Example 1:
# Given a positive integer num, write a function which returns True if num is a pe
# rfect square else False.
# Constraints:
# Follow up: Do not use any built-in library function such as sqrt.

# NOTE
# 1 <= num <= 2^31 - 1

# EXAMPLE
# Input: num = 14
# Output: false
# Input: num = 16
# Output: true

# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def isPerfectSquare(self, num):
        """
        :type num: int
        :rtype: bool
        """
        left, right = 1, num
        while left <= right:
            mid = left + (right - left) / 2
            if mid >= num / mid:
                right = mid - 1
            else:
                left = mid + 1
        return left == num / left and num % left == 0
```

the-k-weakest-rows-in-a-matrix.py

```
# DESC
# A row i is weaker than row j, if the number of soldiers in row i is less than th
# e number of soldiers in row j, or they have the same number of soldiers but i is
#  less than j. Soldiers are always stand in the frontier of a row, that is, alway
# s ones may appear first and then zeros.
# Constraints:
# Example 1:
# Given a m * n matrix mat of ones (representing soldiers) and zeros (representing
#  civilians), return the indexes of the k weakest rows in the matrix ordered from
#  the weakest to the strongest.
# Example 2:

# NOTE
# m == mat.length
# 1 <= k <= m
# 2 <= n, m <= 100
# n == mat[i].length
# matrix[i][j] is either 0 or 1.

# EXAMPLE
# Input: mat =
# [[1,0,0,0],
#  [1,1,1,1],
#  [1,0,0,0],
#  [1,0,0,0]],
# k = 2
# Output: [0
# ,2]
# Explanation:
# The number of soldiers for each row is:
# row 0 -> 1
# row 1 ->
# 4
# row 2 -> 1
# row 3 -> 1
# Rows ordered from the weakest to the strongest are [0
# ,2,3,1]
# Input: mat =
# [[1,1,0,0,0],
#  [1,1,1,1,0],
#  [1,0,0,0,0],
#  [1,1,0,0,0],
#  [1,1,1,1,
# 1]],
# k = 3
# Output: [2,0,3]
# Explanation:
# The number of soldiers for each row is
# :
# row 0 -> 2
# row 1 -> 4
# row 2 -> 1
# row 3 -> 2
# row 4 -> 5
# Rows ordered from
#  the weakest to the strongest are [2,0,3,1,4]

# Time:  O(m * n)
```

```
# Space: O(k)

class Solution(object):
    def kWeakestRows(self, mat, k):
        """
        :type mat: List[List[int]]
        :type k: int
        :rtype: List[int]
        """
        result, lookup = [], set()
        for j in xrange(len(mat[0])):
            for i in xrange(len(mat)):
                if mat[i][j] or i in lookup:
                    continue
                lookup.add(i)
                result.append(i)
                if len(result) == k:
                    return result
        for i in xrange(len(mat)):
            if i in lookup:
                continue
            lookup.add(i)
            result.append(i)
            if len(result) == k:
                break
        return result


# Time:  O(m * n)
# Space: O(k)
import collections


class Solution2(object):
    def kWeakestRows(self, mat, k):
        """
        :type mat: List[List[int]]
        :type k: int
        :rtype: List[int]
        """
        lookup = collections.OrderedDict()
        for j in xrange(len(mat[0])):
            for i in xrange(len(mat)):
                if mat[i][j] or i in lookup:
                    continue
                lookup[i] = True
                if len(lookup) == k:
                    return lookup.keys()
        for i in xrange(len(mat)):
            if i in lookup:
                continue
            lookup[i] = True
            if len(lookup) == k:
                break
        return lookup.keys()


# Time:  O(m * n + klogk)
# Space: O(m)
import random
```

```
class Solution3(object):
    def kWeakestRows(self, mat, k):
        """
        :type mat: List[List[int]]
        :type k: int
        :rtype: List[int]
        """
        def nth_element(nums, n, compare=lambda a, b: a < b):
            def partition_around_pivot(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:
                pivot_idx = random.randint(left, right)
                new_pivot_idx = partition_around_pivot(left, right, pivot_idx, nums, compare)
                if new_pivot_idx == n:
                    return
                elif new_pivot_idx > n:
                    right = new_pivot_idx - 1
                else:  # new_pivot_idx < n
                    left = new_pivot_idx + 1

        nums = [(sum(mat[i]), i) for i in xrange(len(mat))]
        nth_element(nums, k)
        return map(lambda x: x[1], sorted(nums[:k]))
```

missing-number-in-arithmetic-progression.py

```python
# missing-number-in-arithmetic-progression is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def missingNumber(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        def check(arr, d, x):
            return arr[x] != arr[0] + d*x

        d = (arr[-1]-arr[0])//len(arr)
        left, right = 0, len(arr)-1
        while left <= right:
            mid = left + (right-left)//2
            if check(arr, d, mid):
                right = mid-1
            else:
                left = mid+1
        return arr[0] + d*left


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def missingNumber(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        return (min(arr)+max(arr))*(len(arr)+1)//2 - sum(arr)
```

most-profit-assigning-work.py

```python
# DESC
# Every worker can be assigned at most one job, but one job can be completed multi
# ple times.
# worker[i]
# Example 1:
# We have jobs: difficulty[i] is the difficulty of the ith job, and profit[i] is t
# he profit of the ith job.
# What is the most profit we can make?
# Notes:
# Now we have some workers. worker[i] is the ability of the ith worker, which mean
# s that this worker can only complete a job with difficulty at most worker[i].
# For example, if 3 people attempt the same job that pays $1, then the total profi
# t will be $3.  If a worker cannot complete any job, his profit is $0.

# NOTE
# 1 <= worker.length <= 10000
# difficulty[i], profit[i], worker[i]  are in range [1, 10^5]
# 1 <= difficulty.length = profit.length <= 10000

# EXAMPLE
# Input: difficulty = [2,4,6,8,10], profit = [10,20,30,40,50], worker = [4,5,6,7]
#
# Output: 100
# Explanation: Workers are assigned jobs of difficulty [4,4,6,6] and
# they get profit of [20,20,30,30] seperately.

# Time:  O(mlogm + nlogn), m is the number of workers,
#                        , n is the number of jobs
# Space: O(n)

class Solution(object):
    def maxProfitAssignment(self, difficulty, profit, worker):
        """
        :type difficulty: List[int]
        :type profit: List[int]
        :type worker: List[int]
        :rtype: int
        """
        jobs = zip(difficulty, profit)
        jobs.sort()
        worker.sort()
        result, i, max_profit = 0, 0, 0
        for ability in worker:
            while i < len(jobs) and jobs[i][0] <= ability:
                max_profit = max(max_profit, jobs[i][1])
                i += 1
            result += max_profit
        return result
```

number-of-ways-of-cutting-a-pizza.py

```python
# number-of-ways-of-cutting-a-pizza is not found.
# Time:  O(m * n * k * (m + n))
# Space: O(m * n * k)

class Solution(object):
    def ways(self, pizza, k):
        """
        :type pizza: List[str]
        :type k: int
        :rtype: int
        """
        MOD = 10**9+7
        prefix = [[0]*len(pizza[0]) for _ in xrange(len(pizza))]
        for j in reversed(xrange(len(pizza[0]))):
            accu = 0
            for i in reversed(xrange(len(pizza))):
                accu += int(pizza[i][j] == 'A')
                prefix[i][j] = (prefix[i][j+1] if (j+1 < len(pizza[0])) else 0) + accu
        dp = [[[0]*k for _ in xrange(len(pizza[0]))] for _ in xrange(len(pizza))]
        for i in reversed(xrange(len(pizza))):
            for j in reversed(xrange(len(pizza[0]))):
                dp[i][j][0] = 1
                for m in xrange(1, k):
                    for n in xrange(i+1, len(pizza)):
                        if prefix[i][j] == prefix[n][j]:
                            continue
                        if prefix[n][j] == 0:
                            break
                        dp[i][j][m] = (dp[i][j][m] + dp[n][j][m-1]) % MOD
                    for n in xrange(j+1, len(pizza[0])):
                        if prefix[i][j] == prefix[i][n]:
                            continue
                        if prefix[i][n] == 0:
                            break
                        dp[i][j][m] = (dp[i][j][m] + dp[i][n][m-1]) % MOD
        return dp[0][0][k-1]
```

make-two-arrays-equal-by-reversing-sub-arrays.py

```python
# make-two-arrays-equal-by-reversing-sub-arrays is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def canBeEqual(self, target, arr):
        """
        :type target: List[int]
        :type arr: List[int]
        :rtype: bool
        """
        return collections.Counter(target) == collections.Counter(arr)


# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def canBeEqual(self, target, arr):
        """
        :type target: List[int]
        :type arr: List[int]
        :rtype: bool
        """
        target.sort(), arr.sort()
        return target == arr
```

number-of-recent-calls.py

```python
# DESC
# Example 1:
# Note:
# Any ping with time in [t - 3000, t] will count, including the current ping.
# It has only one method: ping(int t), where t represents some time in milliseconds.
# Return the number of pings that have been made from 3000 milliseconds ago until now.
# Write a class RecentCounter to count recent requests.
# ping(int t)
# It is guaranteed that every call to ping uses a strictly larger value of t than
# before.

# NOTE
# Each test case will call ping with strictly increasing values of t.
# Each test case will have at most 10000 calls to ping.
# Each call to ping will have 1 <= t <= 10^9.

# EXAMPLE
# Input: inputs = ["RecentCounter","ping","ping","ping","ping"], inputs = [[],[1],
# [100],[3001],[3002]]
# Output: [null,1,2,3,3]

# Time:  O(1) on average
# Space: O(w), w means the size of the last milliseconds.

import collections


class RecentCounter(object):

    def __init__(self):
        self.__q = collections.deque()

    def ping(self, t):
        """
        :type t: int
        :rtype: int
        """
        self.__q.append(t)
        while self.__q[0] < t-3000:
            self.__q.popleft()
        return len(self.__q)
```

count-of-smaller-numbers-after-self.py

```python
# DESC
# You are given an integer array nums and you have to return a new counts array. T
# he counts array has the property where counts[i] is the number of smaller elemen
# ts to the right of nums[i].
# Constraints:
# Example 1:

# NOTE
# -10^4 <= nums[i] <= 10^4
# 0 <= nums.length <= 10^5

# EXAMPLE
# Input: nums = [5,2,6,1]
# Output: [2,1,1,0]
# Explanation:
# To the right of 5 there a
# re 2 smaller elements (2 and 1).
# To the right of 2 there is only 1 smaller eleme
# nt (1).
# To the right of 6 there is 1 smaller element (1).
# To the right of 1 ther
# e is 0 smaller element.

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

class Solution(object):
    def countSmaller(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        def countAndMergeSort(num_idxs, start, end, counts):
            if end - start <= 0:  # The size of range [start, end] less than 2 is always with count 0.
                return 0

            mid = start + (end - start) / 2
            countAndMergeSort(num_idxs, start, mid, counts)
            countAndMergeSort(num_idxs, mid + 1, end, counts)
            r = mid + 1
            tmp = []
            for i in xrange(start, mid + 1):
                # Merge the two sorted arrays into tmp.
                while r <= end and num_idxs[r][0] < num_idxs[i][0]:
                    tmp.append(num_idxs[r])
                    r += 1
                tmp.append(num_idxs[i])
                counts[num_idxs[i][1]] += r - (mid + 1)

            # Copy tmp back to num_idxs
            num_idxs[start:start+len(tmp)] = tmp

        num_idxs = []
        counts = [0] * len(nums)
        for i, num in enumerate(nums):
            num_idxs.append((num, i))
        countAndMergeSort(num_idxs, 0, len(num_idxs) - 1, counts)
        return counts
```

```
# Time:  O(nlogn)
# Space: O(n)
# BIT solution.
class Solution2(object):
    def countSmaller(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        def binarySearch(A, target, compare):
            start, end = 0, len(A) - 1
            while start <= end:
                mid = start + (end - start) / 2
                if compare(target, A[mid]):
                    end = mid - 1
                else:
                    start = mid + 1
            return start

        class BIT(object):
            def __init__(self, n):
                self.__bit = [0] * n

            def add(self, i, val):
                while i < len(self.__bit):
                    self.__bit[i] += val
                    i += (i & -i)

            def query(self, i):
                ret = 0
                while i > 0:
                    ret += self.__bit[i]
                    i -= (i & -i)
                return ret

        # Get the place (position in the ascending order) of each number.
        sorted_nums, places = sorted(nums), [0] * len(nums)
        for i, num in enumerate(nums):
            places[i] = binarySearch(sorted_nums, num, lambda x, y: x <= y)

        # Count the smaller elements after the number.
        ans, bit = [0] * len(nums), BIT(len(nums) + 1)
        for i in reversed(xrange(len(nums))):
            ans[i] = bit.query(places[i])
            bit.add(places[i] + 1, 1)
        return ans

# Time:  O(nlogn)
# Space: O(n)
# BST solution.
class Solution3(object):
    def countSmaller(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        res = [0] * len(nums)
        bst = self.BST()
        # Insert into BST and get left count.
```

```
        for i in reversed(xrange(len(nums))):
            bst.insertNode(nums[i])
            res[i] = bst.query(nums[i])

        return res

    class BST(object):
        class BSTreeNode(object):
            def __init__(self, val):
                self.val = val
                self.count = 0
                self.left = self.right = None

        def __init__(self):
            self.root = None

        # Insert node into BST.
        def insertNode(self, val):
            node = self.BSTreeNode(val)
            if not self.root:
                self.root = node
                return
            curr = self.root
            while curr:
                # Insert left if smaller.
                if node.val < curr.val:
                    curr.count += 1  # Increase the number of left children.
                    if curr.left:
                        curr = curr.left
                    else:
                        curr.left = node
                        break
                else:  # Insert right if larger or equal.
                    if curr.right:
                        curr = curr.right
                    else:
                        curr.right = node
                        break

        # Query the smaller count of the value.
        def query(self, val):
            count = 0
            curr = self.root
            while curr:
                # Insert left.
                if val < curr.val:
                    curr = curr.left
                elif val > curr.val:
                    count += 1 + curr.count  # Count the number of the smaller nodes.
                    curr = curr.right
                else:  # Equal.
                    return count + curr.count
            return 0
```

find-peak-element.py

```
# DESC
# Example 2:
# A peak element is an element that is greater than its neighbors.
# You may imagine that nums[-1] = nums[n] = -.
# Given an input array nums, where nums[i]  nums[i+1], find a peak element and re
# turn its index.
# Example 1:
# The array may contain multiple peaks, in that case return the index to any one o
# f the peaks is fine.
# Follow up: Your solution should be in logarithmic complexity.

# NOTE
#

# EXAMPLE
# Input: nums = [1,2,3,1]
# Output: 2
# Explanation: 3 is a peak element and your func
# tion should return the index number 2.
# Input: nums = [1,2,1,3,5,6,4]
# Output: 1 or 5
# Explanation: Your function can ret
# urn either index number 1 where the peak element is 2,
#              or index nu
# mber 5 where the peak element is 6.

# 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
```

find-lucky-integer-in-an-array.py

```python
# find-lucky-integer-in-an-arra is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findLucky(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        count = collections.Counter(arr)
        result = -1
        for k, v in count.iteritems():
            if k == v:
                result = max(result, k)
        return result
```

reorder-list.py

```
# DESC
# Given a singly linked list L: L0→L1→...→Ln-1→Ln,
#
# reorder it to: L0→Ln→L1→Ln-1→L2→
# Ln-2→...
# Example 1:
# You may not modify the values in the list's nodes, only nodes itself may be changed.
# Example 2:

# NOTE
#

# EXAMPLE
# Given 1->2->3->4, reorder it to 1->4->2->3.
# 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
```

find-permutation.py

```python
# find-permutation is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findPermutation(self, s):
        """
        :type s: str
        :rtype: List[int]
        """
        result = []
        for i in xrange(len(s)+1):
            if i == len(s) or s[i] == 'I':
                result += range(i+1, len(result), -1)
        return result
```

reducing-dishes.py

```python
# reducing-dishes is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def maxSatisfaction(self, satisfaction):
        """
        :type satisfaction: List[int]
        :rtype: int
        """
        satisfaction.sort(reverse=True)
        result, curr = 0, 0
        for x in satisfaction:
            curr += x
            if curr <= 0:
                break
            result += curr
        return result
```

number-of-steps-to-reduce-a-number-in-binary-representation-to-one.py

```python
# number-of-steps-to-reduce-a-number-in-binary-representation-to-one is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numSteps(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, carry = 0, 0
        for i in reversed(xrange(1, len(s))):
            if int(s[i]) + carry == 1:
                carry = 1  # once it was set, it would keep carrying forever
                result += 2
            else:
                result += 1
        return result+carry
```

day-of-the-year.py

```python
# day-of-the-year is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def __init__(self):
        def dayOfMonth(M):
            return (28 if (M == 2) else 31-(M-1)%7%2)

        self.__lookup = [0]*12
        for M in xrange(1, len(self.__lookup)):
            self.__lookup[M] += self.__lookup[M-1]+dayOfMonth(M)

    def dayOfYear(self, date):
        """
        :type date: str
        :rtype: int
        """
        Y, M, D = map(int, date.split("-"))
        leap = 1 if M > 2 and (((Y % 4 == 0) and (Y % 100 != 0)) or (Y % 400 == 0)) else 0
        return self.__lookup[M-1]+D+leap


# Time:  O(1)
# Space: O(1)
class Solution2(object):
    def dayOfYear(self, date):
        """
        :type date: str
        :rtype: int
        """
        def numberOfDays(Y, M):
            leap = 1 if ((Y % 4 == 0) and (Y % 100 != 0)) or (Y % 400 == 0) else 0
            return (28+leap if (M == 2) else 31-(M-1)%7%2)

        Y, M, result = map(int, date.split("-"))
        for i in xrange(1, M):
            result += numberOfDays(Y, i)
        return result
```

maximum-length-of-pair-chain.py

```
# DESC
# Given a set of pairs, find the length longest chain which can be formed. You nee
# dn't use up all the given pairs. You can select pairs in any order.
# Note:
# (c, d)
# 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.
# You are given n pairs of numbers. In every pair, the first number is always smal
# ler than the second number.
# Example 1:

# NOTE
# The number of given pairs will be in the range [1, 1000].

# EXAMPLE
# Input: [[1,2], [2,3], [3,4]]
# Output: 2
# Explanation: The longest chain is [1,2] -
# > [3,4]

# 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
```

string-matching-in-an-array.py

```python
# string-matching-in-an-arra is not found.
# Time:  O(n + m + z) = O(n), n is the total size of patterns
#                           , m is the total size of query string
#                           , z is the number of all matched strings
#                           , O(n) = O(m) = O(z) in this problem
# Space: O(t), t is the total size of ac automata trie

import collections


class AhoNode(object):
    def __init__(self):
        self.children = collections.defaultdict(AhoNode)
        self.indices = []
        self.suffix = None
        self.output = None


class AhoTrie(object):

    def step(self, letter):
        while self.__node and letter not in self.__node.children:
            self.__node = self.__node.suffix
        self.__node = self.__node.children[letter] if self.__node else self.__root
        return self.__get_ac_node_outputs(self.__node)

    def reset(self):
        self.__node = self.__root

    def __init__(self, patterns):
        self.__root = self.__create_ac_trie(patterns)
        self.__node = self.__create_ac_suffix_and_output_links(self.__root)

    def __create_ac_trie(self, patterns):  # Time:  O(n), Space: O(t)
        root = AhoNode()
        for i, pattern in enumerate(patterns):
            node = root
            for c in pattern:
                node = node.children[c]
            node.indices.append(i)
        return root

    def __create_ac_suffix_and_output_links(self, root):  # Time:  O(n), Space: O(t)
        queue = collections.deque()
        for node in root.children.itervalues():
            queue.append(node)
            node.suffix = root

        while queue:
            node = queue.popleft()
            for c, child in node.children.iteritems():
                queue.append(child)
                suffix = node.suffix
                while suffix and c not in suffix.children:
                    suffix = suffix.suffix
                child.suffix = suffix.children[c] if suffix else root
                child.output = child.suffix if child.suffix.indices else child.suffix.output
```

```
        return root

    def __get_ac_node_outputs(self, node):  # Time:  O(z)
        result = []
        for i in node.indices:
            result.append(i)
        output = node.output
        while output:
            for i in output.indices:
                result.append(i)
            output = output.output
        return result


class Solution(object):
    def stringMatching(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        trie = AhoTrie(words)
        lookup = set()
        for i in xrange(len(words)):
            trie.reset()
            for c in words[i]:
                for j in trie.step(c):
                    if j != i:
                        lookup.add(j)
        return [words[i] for i in lookup]


# Time:  O(n^2 * l), n is the number of strings
# Space: O(l)      , l is the max length of strings
class Solution2(object):
    def stringMatching(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        def getPrefix(pattern):
            prefix = [-1]*len(pattern)
            j = -1
            for i in xrange(1, len(pattern)):
                while j != -1 and pattern[j+1] != pattern[i]:
                    j = prefix[j]
                if pattern[j+1] == pattern[i]:
                    j += 1
                prefix[i] = j
            return prefix

        def kmp(text, pattern, prefix):
            if not pattern:
                return 0
            if len(text) < len(pattern):
                return -1
            j = -1
            for i in xrange(len(text)):
                while j != -1 and pattern[j+1] != text[i]:
                    j = prefix[j]
                if pattern[j+1] == text[i]:
```

```
                j += 1
            if j+1 == len(pattern):
                return i-j
        return -1

    result = []
    for i, pattern in enumerate(words):
        prefix = getPrefix(pattern)
        for j, text in enumerate(words):
            if i != j and kmp(text, pattern, prefix) != -1:
                result.append(pattern)
                break
    return result


# Time:  O(n^2 * l^2), n is the number of strings
# Space: O(1)        , l is the max length of strings
class Solution3(object):
    def stringMatching(self, words):
        """
        :type words: List[str]
        :rtype: List[str]
        """
        result = []
        for i, pattern in enumerate(words):
            for j, text in enumerate(words):
                if i != j and pattern in text:
                    result.append(pattern)
                    break
        return result
```

n-queens.py

```
# DESC
# The n-queens puzzle is the problem of placing n queens on an n×n chessboard such
#  that no two queens attack each other.
# Each solution contains a distinct board configuration of the n-queens' placement
# , where 'Q' and '.' both indicate a queen and an empty space respectively.
# Example:
# Given an integer n, return all distinct solutions to the n-queens puzzle.

# NOTE
#

# EXAMPLE
# Input: 4
# Output: [
#  [".Q..",  // Solution 1
#   "...Q",
#   "Q...",
#   "..Q."],
#
#  [".
# .Q.",  // Solution 2
#   "Q...",
#   "...Q",
#   ".Q.."]
# ]
# Explanation: There exist tw
# o distinct solutions to the 4-queens puzzle as shown above.

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

class Solution(object):
    def solveNQueens(self, n):
        """
        :type n: int
        :rtype: List[List[str]]
        """
        def dfs(curr, cols, main_diag, anti_diag, result):
            row, n = len(curr), len(cols)
            if row == n:
                result.append(map(lambda x: '.'*x + "Q" + '.'*(n-x-1), curr))
                return
            for i in xrange(n):
                if cols[i] or main_diag[row+i] or anti_diag[row-i+n]:
                    continue
                cols[i] = main_diag[row+i] = anti_diag[row-i+n] = True
                curr.append(i)
                dfs(curr, cols, main_diag, anti_diag, result)
                curr.pop()
                cols[i] = main_diag[row+i] = anti_diag[row-i+n] = False

        result = []
        cols, main_diag, anti_diag = [False]*n, [False]*(2*n), [False]*(2*n)
        dfs([], cols, main_diag, anti_diag, result)
        return result


# For any point (x,y), if we want the new point (p,q) don't share the same row, column, or diagonal.
```

```
# then there must have ```p+q != x+y``` and ```p-q!= x-y```
# the former focus on eliminate 'left bottom right top' diagonal
# the latter focus on eliminate 'left top right bottom' diagonal

# - col_per_row: the list of column index per row
# - cur_rowcurrent row we are seraching for valid column
# - xy_diffthe list of x-y
# - xy_sumthe list of x+y
class Solution2(object):
    def solveNQueens(self, n):
        """
        :type n: int
        :rtype: List[List[str]]
        """
        def dfs(col_per_row, xy_diff, xy_sum):
            cur_row = len(col_per_row)
            if cur_row == n:
                ress.append(col_per_row)
            for col in range(n):
                if col not in col_per_row and cur_row-col not in xy_diff and cur_row+col not in xy_sum:
                    dfs(col_per_row+[col], xy_diff+[cur_row-col], xy_sum+[cur_row+col])
        ress = []
        dfs([], [], [])
        return [['.'*i + 'Q' + '.'*(n-i-1) for i in res] for res in ress]
```

integer-to-roman.py

```
# DESC
# Given an integer, convert it to a roman numeral. Input is guaranteed to be withi
# n the range from 1 to 3999.
# Example 5:
# For example, two is written as II in Roman numeral, just two one's added togethe
# r. Twelve is written as, XII, which is simply X + II. The number twenty seven is
#  written as XXVII, which is XX + V + II.
# Example 3:
# Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.
# Example 2:
# Example 4:
# Roman numerals are usually written largest to smallest from left to right. Howev
# er, 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 princip
# le applies to the number nine, which is written as IX. There are six instances w
# here subtraction is used:
# Example 1:

# NOTE
# C can be placed before D (500) and M (1000) to make 400 and 900.
# 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.

# EXAMPLE
# Input: 9
# Output: "IX"
# Input: 3
# Output: "III"
# Input: 58
# Output: "LVIII"
# Explanation: L = 50, V = 5, III = 3.
# Input: 1994
# Output: "MCMXCIV"
# Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
# Symbol       Value
# I             1
# V             5
# X             10
# L
#   50
# C             100
# D             500
# M             1000
# Input: 4
# Output: "IV"

# 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)
```

non-negative-integers-without-consecutive-ones.py

```
# DESC
# Given a positive integer n, find the number of non-negative integers less than o
# r equal to n, whose binary representations do NOT contain consecutive ones.
# Example 1:
# Note:
# 1 <= n <= 109

# NOTE
#

# EXAMPLE
# Input: 5
# Output: 5
# Explanation:
# Here are the non-negative integers <= 5 with th
# eir corresponding binary representations:
# 0 : 0
# 1 : 1
# 2 : 10
# 3 : 11
# 4 : 100
# 5 :
# 101
# Among them, only integer 3 disobeys the rule (two consecutive ones) and the
# other 5 satisfy the rule.

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def findIntegers(self, num):
        """
        :type num: int
        :rtype: int
        """
        dp = [0] * 32
        dp[0], dp[1] = 1, 2
        for i in xrange(2, len(dp)):
            dp[i] = dp[i-1] + dp[i-2]
        result, prev_bit = 0, 0
        for i in reversed(xrange(31)):
            if (num & (1 << i)) != 0:
                result += dp[i]
                if prev_bit == 1:
                    result -= 1
                    break
                prev_bit = 1
            else:
                prev_bit = 0
        return result + 1
```

maximum-number-of-balloons.py

```python
# maximum-number-of-balloons is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def maxNumberOfBalloons(self, text):
        """
        :type text: str
        :rtype: int
        """
        TARGET = "balloon"
        source_count = collections.Counter(text)
        target_count = collections.Counter(TARGET)
        return min(source_count[c]//target_count[c] for c in target_count.iterkeys())
```

cherry-pickup-ii.py

```
# cherry-pickup-ii is not found.
# Time:  O(m * n^2)
# Space: O(n^2)

import itertools


class Solution(object):
    def cherryPickup(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        dp = [[[float("-inf")]*(len(grid[0])+2) for _ in xrange(len(grid[0])+2)] for _ in xrange(2)]
        dp[0][1][len(grid[0])] = grid[0][0] + grid[0][len(grid[0])-1]
        for i in xrange(1, len(grid)):
            for j in xrange(1, len(grid[0])+1):
                for k in xrange(1, len(grid[0])+1):
                    dp[i%2][j][k] = max(dp[(i-1)%2][j+d1][k+d2] for d1 in xrange(-1, 2) for d2 in xrange(-1, 2
                                    ((grid[i][j-1]+grid[i][k-1]) if j != k else grid[i][j-1])
        return max(itertools.imap(max, *dp[(len(grid)-1)%2]))


# Time:  O(m * n^2)
# Space: O(n^2)
import itertools


class Solution2(object):
    def cherryPickup(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        dp = [[[float("-inf")]*len(grid[0]) for _ in xrange(len(grid[0]))] for _ in xrange(2)]
        dp[0][0][len(grid[0])-1] = grid[0][0] + grid[0][len(grid[0])-1]
        for i in xrange(1, len(grid)):
            for j in xrange(len(grid[0])):
                for k in xrange(len(grid[0])):
                    dp[i%2][j][k] = max(dp[(i-1)%2][j+d1][k+d2] for d1 in xrange(-1, 2) for d2 in xrange(-1, 2
                                        if 0 <= j+d1 < len(grid[0]) and 0 <= k+d2 < len(grid[0])) + \
                                    ((grid[i][j]+grid[i][k]) if j != k else grid[i][j])
        return max(itertools.imap(max, *dp[(len(grid)-1)%2]))
```

check-if-a-number-is-majority-element-in-a-sorted-array.py

```
# check-if-a-number-is-majority-element-in-a-sorted-arra is not found.
# Time:  O(logn)
# Space: O(1)

import bisect


class Solution(object):
    def isMajorityElement(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: bool
        """
        if len(nums) % 2:
            if nums[len(nums)//2] != target:
                return False
        else:
            if not (nums[len(nums)//2-1] == nums[len(nums)//2] == target):
                return False

        left = bisect.bisect_left(nums, target)
        right= bisect.bisect_right(nums, target)
        return (right-left)*2 > len(nums)
```

rotting-oranges.py

```
# DESC
# Example 3:
# Example 1:
# In a given grid, each cell can have one of three values:
# Note:
# Example 2:
# Every minute, any fresh orange that is adjacent (4-directionally) to a rotten or
# ange becomes rotten.
# Return the minimum number of minutes that must elapse until no cell has a fresh
# orange.  If this is impossible, return -1 instead.

# NOTE
# 1 <= grid[0].length <= 10
# the value 0 representing an empty cell;
# grid[i][j] is only 0, 1, or 2.
# the value 2 representing a rotten orange.
# the value 1 representing a fresh orange;
# 1 <= grid.length <= 10

# EXAMPLE
# Input: [[2,1,1],[1,1,0],[0,1,1]]
# Output: 4
# Input: [[0,2]]
# Output: 0
# Explanation:  Since there are already no fresh oranges
# at minute 0, the answer is just 0.
# Input: [[2,1,1],[0,1,1],[1,0,1]]
# Output: -1
# Explanation:  The orange in the bott
# om left corner (row 2, column 0) is never rotten, because rotting only happens 4
# -directionally.

# 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
```

sum-of-root-to-leaf-binary-numbers.py

```
# DESC
# For all leaves in the tree, consider the numbers represented by the path from th
# e root to that leaf.
# Given a binary tree, each node has value 0 or 1.  Each root-to-leaf path represe
# nts a binary number starting with the most significant bit.  For example, if the
#  path is 0 -> 1 -> 1 -> 0 -> 1, then this could represent 01101 in binary, which
#  is 13.
# Note:
# Return the sum of these numbers.
# Example 1:

# NOTE
# The answer will not exceed 2^31 - 1.
# The number of nodes in the tree is between 1 and 1000.
# node.val is 0 or 1.

# EXAMPLE
# Input: [1,0,1,0,1,0,1]
# Output: 22
# Explanation: (100) + (101) + (110) + (111) = 4
#  + 5 + 6 + 7 = 22

# 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 sumRootToLeaf(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        M = 10**9 + 7
        def sumRootToLeafHelper(root, val):
            if not root:
                return 0
            val = (val*2 + root.val) % M
            if not root.left and not root.right:
                return val
            return (sumRootToLeafHelper(root.left, val) +
                    sumRootToLeafHelper(root.right, val)) % M

        return sumRootToLeafHelper(root, 0)
```

apply-discount-every-n-orders.py

```python
# apply-discount-every-n-orders is not found.
# Time:  ctor:    O(m), m is the number of all products
#        getBill: O(p), p is the number of products to bill
# Space: O(m)

class Cashier(object):

    def __init__(self, n, discount, products, prices):
        """
        :type n: int
        :type discount: int
        :type products: List[int]
        :type prices: List[int]
        """
        self.__n = n
        self.__discount = discount
        self.__curr = 0
        self.__lookup = {p : prices[i] for i, p in enumerate(products)}

    def getBill(self, product, amount):
        """
        :type product: List[int]
        :type amount: List[int]
        :rtype: float
        """
        self.__curr = (self.__curr+1) % self.__n
        result = 0.0
        for i, p in enumerate(product):
            result += self.__lookup[p]*amount[i]
        return result * (1.0 - self.__discount/100.0 if self.__curr == 0 else 1.0)
```

check-if-it-is-a-good-array.py

```python
# check-if-it-is-a-good-arra is not found.
# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def isGoodArray(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        def gcd(a, b):
            while b:
                a, b = b, a%b
            return a

        # Bézout's identity
        result = nums[0]
        for num in nums:
            result = gcd(result, num)
            if result == 1:
                break
        return result == 1
```

snapshot-array.py

```python
# snapshot-arra is not found.
# 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
        """
        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

```
# DESC
# Example 2:
# Example 1:
# Note:
# Given a string and a string dictionary, find the longest string in the dictionar
# y that can be formed by deleting some characters of the given string. If there a
# re more than one possible results, return the longest word with the smallest lex
# icographical order. If there is no possible result, return the empty string.

# NOTE
# The size of the dictionary won't exceed 1,000.
# The length of all the strings in the input won't exceed 1,000.
# All the strings in the input will only contain lower-case letters.

# EXAMPLE
# Input:
# s = "abpcplea", d = ["ale","apple","monkey","plea"]
#
# Output:
# "apple"
# Input:
# s = "abpcplea", d = ["a","b","c"]
#
# Output:
# "a"

# Time:  O((d * l) * logd), l is the average length of words
# 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 ""
```

reverse-string-ii.py

```
# DESC
# Example:

# NOTE
# The string consists of lower English letters only.
# Length of the given string and k will in the range [1, 10000]

# EXAMPLE
# Input: s = "abcdefg", k = 2
# Output: "bacdfeg"

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def reverseStr(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: str
        """
        s = list(s)
        for i in xrange(0, len(s), 2*k):
            s[i:i+k] = reversed(s[i:i+k])
        return "".join(s)
```

maximum-subarray.py

```
# maximum-subarra is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, curr = float("-inf"), float("-inf")
        for x in nums:
            curr = max(curr+x, x)
            result = max(result, curr)
        return result
```

power-of-two.py

```python
# DESC
# Given an integer, write a function to determine if it is a power of two.
# Example 2:
# Example 3:
# Example 1:

# NOTE
#

# EXAMPLE
# Input: 16
# Output: true
# Explanation: 24 = 16
# Input: 218
# Output: false
# Input: 1
# Output: true
# Explanation: 20 = 1

# Time:  O(1)
# Space: O(1)

class Solution(object):
    # @param {integer} n
    # @return {boolean}
    def isPowerOfTwo(self, n):
        return n > 0 and (n & (n - 1)) == 0


class Solution2(object):
    # @param {integer} n
    # @return {boolean}
    def isPowerOfTwo(self, n):
        return n > 0 and (n & ~-n) == 0
```

number-of-closed-islands.py

```python
# number-of-closed-islands is not found.
# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def closedIsland(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        def fill(grid, i, j):
            if not (0 <= i < len(grid) and
                    0 <= j < len(grid[0]) and
                    grid[i][j] == 0):
                return False
            grid[i][j] = 1
            for dx, dy in directions:
                fill(grid, i+dx, j+dy)
            return True

        for j in xrange(len(grid[0])):
            fill(grid, 0, j)
            fill(grid, len(grid)-1, j)
        for i in xrange(1, len(grid)):
            fill(grid, i, 0)
            fill(grid, i, len(grid[0])-1)
        result = 0
        for i in xrange(1, len(grid)-1):
            for j in xrange(1, len(grid[0])-1):
                if fill(grid, i, j):
                    result += 1
        return result
```

jump-game-ii.py

```python
# DESC
# You can assume that you can always reach the last index.
# Your goal is to reach the last index in the minimum number of jumps.
# Note:
# Given an array of non-negative integers, you are initially positioned at the fir
# st index of the array.
# Example:
# Each element in the array represents your maximum jump length at that position.

# NOTE
#

# EXAMPLE
# Input: [2,3,1,1,4]
# Output: 2
# Explanation: The minimum number of jumps to reach t
# he last index is 2.
#     Jump 1 step from index 0 to 1, then 3 steps to the last
# index.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A, a list of integers
    # @return an integer
    def jump(self, A):
        jump_count = 0
        reachable = 0
        curr_reachable = 0
        for i, length in enumerate(A):
            if i > reachable:
                return -1
            if i > curr_reachable:
                curr_reachable = reachable
                jump_count += 1
            reachable = max(reachable, i + length)
        return jump_count
```

reconstruct-itinerary.py

```
# reconstruct-itinerar is not found.
# Time:  O(t! / (n1! * n2! * ... nk!)), t is the total number of tickets,
#                                       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
```

power-of-four.py

```
# DESC
# Given an integer (signed 32 bits), write a function to check whether it is a pow
# er of 4.
# Example 2:
# Example 1:
# Follow up: Could you solve it without loops/recursion?

# NOTE
#

# EXAMPLE
# Input: 5
# Output: false
# Input: 16
# Output: true

# Time:  O(1)
# Space: O(1)

class Solution(object):
    def isPowerOfFour(self, num):
        """
        :type num: int
        :rtype: bool
        """
        return num > 0 and (num & (num - 1)) == 0 and \
               ((num & 0b01010101010101010101010101010101) == num)


# Time:  O(1)
# Space: O(1)
class Solution2(object):
    def isPowerOfFour(self, num):
        """
        :type num: int
        :rtype: bool
        """
        while num and not (num & 0b11):
            num >>= 2
        return (num == 1)


class Solution3(object):
    def isPowerOfFour(self, num):
        """
        :type num: int
        :rtype: bool
        """
        num = bin(num)
        return True if num[2:].startswith('1') and len(num[2:]) == num.count('0') and num.count('0') % 2 and '
```

the-maze.py

```python
# the-maze is not found.
# Time:  O(max(r, c) * w)
# Space: O(w)

import collections


class Solution(object):
    def hasPath(self, maze, start, destination):
        """
        :type maze: List[List[int]]
        :type start: List[int]
        :type destination: List[int]
        :rtype: bool
        """
        def neighbors(maze, node):
            for i, j in [(-1, 0), (0, 1), (0, -1), (1, 0)]:
                x, y = node
                while 0 <= x + i < len(maze) and \
                      0 <= y + j < len(maze[0]) and \
                      not maze[x+i][y+j]:
                    x += i
                    y += j
                yield x, y

        start, destination = tuple(start), tuple(destination)
        queue = collections.deque([start])
        visited = set()
        while queue:
            node = queue.popleft()
            if node in visited: continue
            if node == destination:
                return True
            visited.add(node)
            for neighbor in neighbors(maze, node):
                queue.append(neighbor)

        return False
```

longest-line-of-consecutive-one-in-matrix.py

```python
# longest-line-of-consecutive-one-in-matrix is not found.
# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    def longestLine(self, M):
        """
        :type M: List[List[int]]
        :rtype: int
        """
        if not M: return 0
        result = 0
        dp = [[[0] * 4 for _ in xrange(len(M[0]))] for _ in xrange(2)]
        for i in xrange(len(M)):
            for j in xrange(len(M[0])):
                dp[i % 2][j][:] = [0] * 4
                if M[i][j] == 1:
                    dp[i % 2][j][0] = dp[i % 2][j - 1][0]+1 if j > 0 else 1
                    dp[i % 2][j][1] = dp[(i-1) % 2][j][1]+1 if i > 0 else 1
                    dp[i % 2][j][2] = dp[(i-1) % 2][j-1][2]+1 if (i > 0 and j > 0) else 1
                    dp[i % 2][j][3] = dp[(i-1) % 2][j+1][3]+1 if (i > 0 and j < len(M[0])-1) else 1
                    result = max(result, max(dp[i % 2][j]))
        return result
```

sqrtx.py

```
# sqrtx is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def mySqrt(self, x):
        """
        :type x: int
        :rtype: int
        """
        if x < 2:
            return x

        left, right = 1, x // 2
        while left <= right:
            mid = left + (right - left) // 2
            if mid > x / mid:
                right = mid - 1
            else:
                left = mid + 1

        return left - 1
```

find-longest-awesome-substring.py

```python
# find-longest-awesome-substring is not found.
# Time:  O(10 * n)
# Space: O(1024)

class Solution(object):
    def longestAwesome(self, s):
        """
        :type s: str
        :rtype: int
        """
        ALPHABET_SIZE = 10
        result, mask, lookup = 0, 0, [len(s)]*(2**ALPHABET_SIZE)
        lookup[0] = -1
        for i, ch in enumerate(s):
            mask ^= 2**(ord(ch)-ord('0'))
            if lookup[mask] == len(s):
                lookup[mask] = i
            result = max(result, i - lookup[mask])
            for d in xrange(ALPHABET_SIZE):
                result = max(result, i - lookup[mask^(2**d)])
        return result
```

add-two-numbers.py

```
# DESC
# Example:
# 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 numbe
# r 0 itself.

# NOTE
#

# 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
```

sum-of-square-numbers.py

```
# DESC
# Example 2:
# Example 1:
# Given a non-negative integer c, your task is to decide whether there're two inte
# gers a and b such that a2 + b2 = c.

# NOTE
#

# EXAMPLE
# Input: 5
# Output: True
# Explanation: 1 * 1 + 2 * 2 = 5
# Input: 3
# Output: False

# Time:  O(sqrt(c) * logc)
# Space: O(1)

import math


class Solution(object):
    def judgeSquareSum(self, c):
        """
        :type c: int
        :rtype: bool
        """
        for a in xrange(int(math.sqrt(c))+1):
            b = int(math.sqrt(c-a**2))
            if a**2 + b**2 == c:
                return True
        return False
```

remove-palindromic-subsequences.py

```python
# DESC
# Return the minimum number of steps to make the given string empty.
# A string is a subsequence of a given string, if it is generated by deleting some
#  characters of a given string without changing its order.
# Given a string s consisting only of letters 'a' and 'b'. In a single step you ca
# n remove one palindromic subsequence from s.
# Constraints:
# Example 2:
# Example 3:
# Example 1:
# A string is called palindrome if is one that reads the same backward as well as
# forward.
# Example 4:

# NOTE
# 0 <= s.length <= 1000
# s only consists of letters 'a' and 'b'

# EXAMPLE
# Input: s = "abb"
# Output: 2
# Explanation: "abb" -> "bb" -> "".
# Remove palindromic
#  subsequence "a" then "bb".
# Input: s = ""
# Output: 0
# Input: s = "ababa"
# Output: 1
# Explanation: String is already palindrome
# Input: s = "baabb"
# Output: 2
# Explanation: "baabb" -> "b" -> "".
# Remove palindro
# mic subsequence "baab" then "b".

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def removePalindromeSub(self, s):
        """
        :type s: str
        :rtype: int
        """
        def is_palindrome(s):
            for i in xrange(len(s)//2):
                if s[i] != s[-1-i]:
                    return False
            return True

        return 2 - is_palindrome(s) - (s == "")
```

find-the-celebrity.py

```
# find-the-celebrit is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findCelebrity(self, n):
        """
        :type n: int
        :rtype: int
        """
        candidate = 0
        # Find the candidate.
        for i in xrange(1, n):
            if knows(candidate, i):  # noqa
                candidate = i        # All candidates < i are not celebrity candidates.
        # Verify the candidate.
        for i in xrange(n):
            candidate_knows_i = knows(candidate, i) # noqa
            i_knows_candidate = knows(i, candidate) # noqa
            if i != candidate and (candidate_knows_i or
                                   not i_knows_candidate):
                return -1
        return candidate
```

find-the-difference.py

```
# DESC
# String t is generated by random shuffling string s and then add one more letter
# at a random position.
# Example:
# Given two strings s and t which consist of only lowercase letters.
# Find the letter that was added in t.

# NOTE
#

# EXAMPLE
# Input:
# s = "abcd"
# t = "abcde"
#
# Output:
# e
#
# Explanation:
# 'e' is the letter that wa
# s added.

# Time:  O(n)
# Space: O(1)

import operator
import collections
from functools import reduce


class Solution(object):
    def findTheDifference(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        return chr(reduce(operator.xor, map(ord, s), 0) ^ reduce(operator.xor, map(ord, t), 0))

    def findTheDifference2(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: str
        """
        t = list(t)
        s = list(s)
        for i in s:
            t.remove(i)
        return t[0]

    def findTheDifference3(self, s, t):
        return chr(reduce(operator.xor, map(ord, s + t)))

    def findTheDifference4(self, s, t):
        return list((collections.Counter(t) - collections.Counter(s)))[0]

    def findTheDifference5(self, s, t):
```

```
s, t = sorted(s), sorted(t)
return t[-1] if s == t[:-1] else [x[1] for x in zip(s, t) if x[0] != x[1]][0]
```

leftmost-column-with-at-least-a-one.py

```python
# leftmost-column-with-at-least-a-one is not found.
# Time:  O(m + n)
# Space: O(1)

class BinaryMatrix(object):
    def get(self, row, col):
        pass

    def dimensions(self):
        pass


class Solution(object):
    def leftMostColumnWithOne(self, binaryMatrix):
        """
        :type binaryMatrix: BinaryMatrix
        :rtype: int
        """
        m, n = binaryMatrix.dimensions()
        r, c = 0, n-1
        while r < m and c >= 0:
            if not binaryMatrix.get(r, c):
                r += 1
            else:
                c -= 1
        return c+1 if c+1 != n else -1
```

prison-cells-after-n-days.py

```
# DESC
# Note:
# Example 1:
# There are 8 prison cells in a row, and each cell is either occupied or vacant.
# Each day, whether the cell is occupied or vacant changes according to the follow
# ing rules:
# We describe the current state of the prison in the following way: cells[i] == 1
# if the i-th cell is occupied, else cells[i] == 0.
# Example 2:
# Given the initial state of the prison, return the state of the prison after N da
# ys (and N such changes described above.)
# (Note that because the prison is a row, the first and the last cells in the row
# can't have two adjacent neighbors.)

# NOTE
# cells[i] is in {0, 1}
# If a cell has two adjacent neighbors that are both occupied or both vacant, then
#  the cell becomes occupied.
# Otherwise, it becomes vacant.
# cells.length == 8
# 1 <= N <= 10^9

# EXAMPLE
# Input: cells = [0,1,0,1,1,0,0,1], N = 7
# Output: [0,0,1,1,0,0,0,0]
# Explanation:
#
# The following table summarizes the state of the prison on each day:
# Day 0: [0, 1
# , 0, 1, 1, 0, 0, 1]
# Day 1: [0, 1, 1, 0, 0, 0, 0, 0]
# Day 2: [0, 0, 0, 0, 1, 1, 1,
#  0]
# Day 3: [0, 1, 1, 0, 0, 1, 0, 0]
# Day 4: [0, 0, 0, 0, 0, 1, 0, 0]
# Day 5: [0, 1
# , 1, 1, 0, 1, 0, 0]
# Day 6: [0, 0, 1, 0, 1, 1, 0, 0]
# Day 7: [0, 0, 1, 1, 0, 0, 0,
#  0]
# Input: cells = [1,0,0,1,0,0,1,0], N = 1000000000
# Output: [0,0,1,1,1,1,1,0]

# 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
```

```python
# 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)
```

design-a-leaderboard.py

```
# design-a-leaderboard is not found.
# Time:  ctor:  O(1)
#        add:   O(1)
#        top:   O(n)
#        reset: O(1)
# Space: O(n)

import collections
import random


class Leaderboard(object):

    def __init__(self):
        self.__lookup = collections.Counter()

    def addScore(self, playerId, score):
        """
        :type playerId: int
        :type score: int
        :rtype: None
        """
        self.__lookup[playerId] += score

    def top(self, K):
        """
        :type K: int
        :rtype: int
        """
        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:
                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

        scores = self.__lookup.values()
        kthElement(scores, K, lambda a, b: a > b)
        return sum(scores[:K])

    def reset(self, playerId):
        """
```

```
        :type playerId: int
        :rtype: None
        """
        self.__lookup[playerId] = 0
```

nested-list-weight-sum.py

```python
# nested-list-weight-sum is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    def depthSum(self, nestedList):
        """
        :type nestedList: List[NestedInteger]
        :rtype: int
        """
        def depthSumHelper(nestedList, depth):
            res = 0
            for l in nestedList:
                if l.isInteger():
                    res += l.getInteger() * depth
                else:
                    res += depthSumHelper(l.getList(), depth + 1)
            return res
        return depthSumHelper(nestedList, 1)
```

subdomain-visit-count.py

```
# DESC
# We are given a list cpdomains of count-paired domains. We would like a list of c
# ount-paired domains, (in the same format as the input, and in any order), that e
# xplicitly counts the number of visits to each subdomain.
# A website domain like "discuss.leetcode.com" consists of various subdomains. At
# the top level, we have "com", at the next level, we have "leetcode.com", and at
# the lowest level, "discuss.leetcode.com". When we visit a domain like "discuss.l
# eetcode.com", we will also visit the parent domains "leetcode.com" and "com" imp
# licitly.
# Now, call a "count-paired domain" to be a count (representing the number of visi
# ts this domain received), followed by a space, followed by the address. An examp
# le of a count-paired domain might be "9001 discuss.leetcode.com".
# Notes:

# NOTE
# The length of cpdomains will not exceed 100.
# The length of each domain name will not exceed 100.
# Each address will have either 1 or 2 "." characters.
# The input count in any count-paired domain will not exceed 10000.
# The answer output can be returned in any order.

# EXAMPLE
# Example 1:
# Input:
# ["9001 discuss.leetcode.com"]
# Output:
# ["9001 discuss.leetcod
# e.com", "9001 leetcode.com", "9001 com"]
# Explanation:
# We only have one website
# domain: "discuss.leetcode.com". As discussed above, the subdomain "leetcode.com"
#  and "com" will also be visited. So they will all be visited 9001 times.
# Example 2:
# Input:
# ["900 google.mail.com", "50 yahoo.com", "1 intel.mail.com", "
# 5 wiki.org"]
# Output:
# ["901 mail.com","50 yahoo.com","900 google.mail.com","5 wi
# ki.org","5 org","1 intel.mail.com","951 com"]
# Explanation:
# We will visit "googl
# e.mail.com" 900 times, "yahoo.com" 50 times, "intel.mail.com" once and "wiki.org
# " 5 times. For the subdomains, we will visit "mail.com" 900 + 1 = 901 times, "co
# m" 900 + 50 + 1 = 951 times, and "org" 5 times.

# Time:  O(n), is the length of cpdomains (assuming the length of cpdomains[i] is fixed)
# Space: O(n)

import collections


class Solution(object):
    def subdomainVisits(self, cpdomains):
        """
        :type cpdomains: List[str]
        :rtype: List[str]
        """
        result = collections.defaultdict(int)
```

```
    for domain in cpdomains:
        count, domain = domain.split()
        count = int(count)
        frags = domain.split('.')
        curr = []
        for i in reversed(xrange(len(frags))):
            curr.append(frags[i])
            result[".".join(reversed(curr))] += count

    return ["{} {}".format(count, domain) \
            for domain, count in result.iteritems()]
```

sum-of-digits-in-the-minimum-number.py

```python
# sum-of-digits-in-the-minimum-number is not found.
# Time:  O(n * l)
# Space: O(l)

class Solution(object):
    def sumOfDigits(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        total = sum([int(c) for c in str(min(A))])
        return 1 if total % 2 == 0 else 0
```

license-key-formatting.py

```
# DESC
# Given a number K, we would want to reformat the strings such that each group con
# tains exactly K characters, except for the first group which could be shorter th
# an K, but still must contain at least one character. Furthermore, there must be
# a dash inserted between two groups and all lowercase letters should be converted
#  to uppercase.
# Note:
# You are given a license key represented as a string S which consists only alphan
# umeric character and dashes. The string is separated into N+1 groups by N dashes
# .
# Example 1:
# Example 2:
# Given a non-empty string S and a number K, format the string according to the ru
# les described above.

# NOTE
# String S is non-empty.
# The length of string S will not exceed 12,000, and K is a positive integer.
# String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9)
# and dashes(-).

# EXAMPLE
# Input: S = "2-5g-3-J", K = 2
#
# Output: "2-5G-3J"
#
# Explanation: The string S has b
# een split into three parts, each part has 2 characters except the first part as
# it could be shorter as mentioned above.
# Input: S = "5F3Z-2e-9-w", K = 4
#
# Output: "5F3Z-2E9W"
#
# Explanation: The string S
# has been split into two parts, each part has 4 characters.
# Note that the two ext
# ra dashes are not needed and can be removed.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def licenseKeyFormatting(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: str
        """
        result = []
        for i in reversed(xrange(len(S))):
            if S[i] == '-':
                continue
            if len(result) % (K + 1) == K:
                result += '-'
            result += S[i].upper()
        return "".join(reversed(result))
```

compare-strings-by-frequency-of-the-smallest-character.py

```python
# compare-strings-by-frequency-of-the-smallest-character is not found.
# Time:  O((m + n)logn), m is the number of queries, n is the number of words
# Space: O(n)

import bisect


class Solution(object):
    def numSmallerByFrequency(self, queries, words):
        """
        :type queries: List[str]
        :type words: List[str]
        :rtype: List[int]
        """
        words_freq = sorted(word.count(min(word)) for word in words)
        return [len(words)-bisect.bisect_right(words_freq, query.count(min(query))) \
                for query in queries]
```

path-with-maximum-gold.py

```
# path-with-maximum-gold is not found.
# 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
```

number-of-atoms.py

```
# DESC
# Example 2:
# Note:
# Example 3:
# A formula placed in parentheses, and a count (optionally added) is also a formul
# a.  For example, (H2O2) and (H2O2)3 are formulas.
# An atomic element always starts with an uppercase character, then zero or more l
# owercase letters, representing the name.
# Two formulas concatenated together produce another formula.  For example, H2O2He
# 3Mg4 is also a formula.
# 1 or more digits representing the count of that element may follow if the count
# is greater than 1.  If the count is 1, no digits will follow.  For example, H2O
# and H2O2 are possible, but H1O2 is impossible.
# Given a formula, output the count of all elements as a string in the following f
# orm: the first name (in sorted order), followed by its count (if that count is m
# ore than 1), followed by the second name (in sorted order), followed by its coun
# t (if that count is more than 1), and so on.
# Example 1:
# Given a chemical formula (given as a string), return the count of each atom.

# NOTE
# The length of formula will be in the range [1, 1000].
# formula will only consist of letters, digits, and round parentheses, and is a va
# lid formula as defined in the problem.
# All atom names consist of lowercase letters, except for the first character whic
# h is uppercase.

# EXAMPLE
# Input:
# formula = "K4(ON(SO3)2)2"
# Output: "K4N2O14S4"
# Explanation:
# The count of
#  elements are {'K': 4, 'N': 2, 'O': 14, 'S': 4}.
# Input:
# formula = "Mg(OH)2"
# Output: "H2MgO2"
# Explanation:
# The count of elements
#  are {'H': 2, 'Mg': 1, 'O': 2}.
# Input:
# formula = "H2O"
# Output: "H2O"
# Explanation:
# The count of elements are {'
# H': 2, 'O': 1}.

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

import collections
import re


class Solution(object):
    def countOfAtoms(self, formula):
        """
        :type formula: str
```

```
    :rtype: str
    """
    parse = re.findall(r"([A-Z][a-z]*)(\d*)|(\()|(\))(\d*)", formula)
    stk = [collections.Counter()]
    for name, m1, left_open, right_open, m2 in parse:
        if name:
          stk[-1][name] += int(m1 or 1)
        if left_open:
          stk.append(collections.Counter())
        if right_open:
            top = stk.pop()
            for k, v in top.iteritems():
              stk[-1][k] += v * int(m2 or 1)

    return "".join(name + (str(stk[-1][name]) if stk[-1][name] > 1 else '') \
                  for name in sorted(stk[-1]))
```

minimum-time-to-collect-all-apples-in-a-tree.py

```python
# minimum-time-to-collect-all-apples-in-a-tree is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def minTime(self, n, edges, hasApple):
        """
        :type n: int
        :type edges: List[List[int]]
        :type hasApple: List[bool]
        :rtype: int
        """
        graph = collections.defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        result = [0, 0]
        s = [(1, (-1, 0, result))]
        while s:
            step, params = s.pop()
            if step == 1:
                par, node, ret = params
                ret[:] = [0, int(hasApple[node])]
                for nei in reversed(graph[node]):
                    if nei == par:
                        continue
                    new_ret = [0, 0]
                    s.append((2, (new_ret, ret)))
                    s.append((1, (node, nei, new_ret)))
            else:
                new_ret, ret = params
                ret[0] += new_ret[0]+new_ret[1]
                ret[1] |= bool(new_ret[0]+new_ret[1])
        return 2*result[0]


# Time:  O(n)
# Space: O(n)
class Solution_Recu(object):
    def minTime(self, n, edges, hasApple):
        """
        :type n: int
        :type edges: List[List[int]]
        :type hasApple: List[bool]
        :rtype: int
        """
        def dfs(graph, par, node, hasApple):
            result, extra = 0, int(hasApple[node])
            for nei in graph[node]:
                if nei == par:
                    continue
                count, found = dfs(graph, node, nei, hasApple)
                result += count+found
                extra |= bool(count+found)
```

```
            return result, extra

        graph = collections.defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)
        return 2*dfs(graph, -1, 0, hasApple)[0]


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def minTime(self, n, edges, hasApple):
        """
        :type n: int
        :type edges: List[List[int]]
        :type hasApple: List[bool]
        :rtype: int
        """
        graph = collections.defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        result = [0]
        s = [(1, (-1, 0, result))]
        while s:
            step, params = s.pop()
            if step == 1:
                par, node, ret = params
                tmp = [int(hasApple[node])]
                s.append((3, (tmp, ret)))
                for nei in reversed(graph[node]):
                    if nei == par:
                        continue
                    new_ret = [0]
                    s.append((2, (new_ret, tmp, ret)))
                    s.append((1, (node, nei, new_ret)))
            elif step == 2:
                new_ret, tmp, ret = params
                ret[0] += new_ret[0]
                tmp[0] |= bool(new_ret[0])
            else:
                tmp, ret = params
                ret[0] += tmp[0]
        return 2*max(result[0]-1, 0)


# Time:  O(n)
# Space: O(n)
class Solution2_Recu(object):
    def minTime(self, n, edges, hasApple):
        """
        :type n: int
        :type edges: List[List[int]]
        :type hasApple: List[bool]
        :rtype: int
        """
        def dfs(graph, par, node, has_subtree):
            result, extra = 0, int(hasApple[node])
```

```
    for nei in graph[node]:
        if nei == par:
            continue
        count = dfs(graph, node, nei, hasApple)
        result += count
        extra |= bool(count)
    return result+extra

graph = collections.defaultdict(list)
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)
return 2*max(dfs(graph, -1, 0, hasApple)-1, 0)
```

unique-binary-search-trees.py

```
# DESC
# Constraints:
# Example:
# Given n, how many structurally unique BST's (binary search trees) that store val
# ues 1 ... n?

# NOTE
# 1 <= n <= 19

# 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

# 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-number-of-refueling-stops.py

```
# DESC
# Note:
# Example 1:
# When the car reaches a gas station, it may stop and refuel, transferring all the
#  gas from the station into the car.
# The car starts with an infinite tank of gas, which initially has startFuel liter
# s of fuel in it.  It uses 1 liter of gas per 1 mile that it drives.
# Along the way, there are gas stations.  Each station[i] represents a gas station
#  that is station[i][0] miles east of the starting position, and has station[i][1
# ] liters of gas.
# A car travels from a starting position to a destination which is target miles ea
# st of the starting position.
# Note that if the car reaches a gas station with 0 fuel left, the car can still r
# efuel there.  If the car reaches the destination with 0 fuel left, it is still c
# onsidered to have arrived.
# Example 2:
# station[i]
# What is the least number of refueling stops the car must make in order to reach
# its destination?  If it cannot reach the destination, return -1.
# Example 3:

# NOTE
# 1 <= target, startFuel, stations[i][1] <= 10^9
# 0 < stations[0][0] < stations[1][0] < ... < stations[stations.length-1][0] < target
# 0 <= stations.length <= 500

# EXAMPLE
# Input: target = 100, startFuel = 10, stations = [[10,60],[20,30],[30,30],[60,40]
# ]
# Output: 2
# Explanation:
# We start with 10 liters of fuel.
# We drive to position
# 10, expending 10 liters of fuel.  We refuel from 0 liters to 60 liters of gas.
# T
# hen, we drive from position 10 to position 60 (expending 50 liters of fuel),
# and
#  refuel from 10 liters to 50 liters of gas.  We then drive to and reach the targ
# et.
# We made 2 refueling stops along the way, so we return 2.
# Input: target = 1, startFuel = 1, stations = []
# Output: 0
# Explanation: We can re
# ach the target without refueling.
# Input: target = 100, startFuel = 1, stations = [[10,100]]
# Output: -1
# Explanation
# : We can't reach the target (or even the first gas station).

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

import heapq


class Solution(object):
    def minRefuelStops(self, target, startFuel, stations):
        """
```

```
        :type target: int
        :type startFuel: int
        :type stations: List[List[int]]
        :rtype: int
        """
        max_heap = []
        stations.append((target, float("inf")))

        result = prev = 0
        for location, capacity in stations:
            startFuel -= location - prev
            while max_heap and startFuel < 0:
                startFuel += -heapq.heappop(max_heap)
                result += 1
            if startFuel < 0:
                return -1
            heapq.heappush(max_heap, -capacity)
            prev = location

        return result
```

web-crawler.py

```
# web-crawler is not found.
# Time:  O(|V| + |E|)
# Space: O(|V|)

# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
class HtmlParser(object):
   def getUrls(self, url):
       """
       :type url: str
       :rtype List[str]
       """
       pass


class Solution(object):
    def crawl(self, startUrl, htmlParser):
        """
        :type startUrl: str
        :type htmlParser: HtmlParser
        :rtype: List[str]
        """
        SCHEME = "http://"
        def hostname(url):
            pos = url.find('/', len(SCHEME))
            if pos == -1:
                return url
            return url[:pos]

        result = [startUrl]
        lookup = set(result)
        for from_url in result:
            name = hostname(from_url)
            for to_url in htmlParser.getUrls(from_url):
                if to_url not in lookup and name == hostname(to_url):
                    result.append(to_url)
                    lookup.add(to_url)
        return result
```

maximum-69-number.py

```python
# maximum-69-number is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def maximum69Number (self, num):
        """
        :type num: int
        :rtype: int
        """
        curr, base, change = num, 3, 0
        while curr:
            if curr%10 == 6:
                change = base
            base *= 10
            curr //= 10
        return num+change


# Time:  O(logn)
# Space: O(logn)
class Solution2(object):
    def maximum69Number (self, num):
        """
        :type num: int
        :rtype: int
        """
        return int(str(num).replace('6', '9', 1))
```

find-eventual-safe-states.py

```
# DESC
# Which nodes are eventually safe?  Return them as an array in sorted order.
# Now, say our starting node is eventually safe if and only if we must eventually
# walk to a terminal node.  More specifically, there exists a natural number K so
# that for any choice of where to walk, we must have stopped at a terminal node in
#  less than K steps.
# Note:
# In a directed graph, we start at some node and every turn, walk along a directed
#  edge of the graph.  If we reach a node that is terminal (that is, it has no out
# going directed edges), we stop.
# The directed graph has N nodes with labels 0, 1, ..., N-1, where N is the length
#  of graph.  The graph is given in the following form: graph[i] is a list of labe
# ls j such that (i, j) is a directed edge of the graph.

# NOTE
# The number of edges in the graph will not exceed 32000.
# graph will have length at most 10000.
# Each graph[i] will be a sorted list of different integers, chosen within the ran
# ge [0, graph.length - 1].

# EXAMPLE
# Example:
# Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]
# Output: [2,4,5,6]
# Here i
# s a diagram of the above graph.

# Time:  O(|V| + |E|)
# Space: O(|V|)

import collections


class Solution(object):
    def eventualSafeNodes(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: List[int]
        """
        WHITE, GRAY, BLACK = 0, 1, 2

        def dfs(graph, node, lookup):
            if lookup[node] != WHITE:
                return lookup[node] == BLACK
            lookup[node] = GRAY
            for child in graph[node]:
                if lookup[child] == BLACK:
                    continue
                if lookup[child] == GRAY or \
                   not dfs(graph, child, lookup):
                    return False
            lookup[node] = BLACK
            return True

        lookup = collections.defaultdict(int)
        return filter(lambda node: dfs(graph, node, lookup), xrange(len(graph)))
```

shortest-distance-from-all-buildings.py

```python
# shortest-distance-from-all-buildings is not found.
# Time:  O(k * m * n), k is the number of the buildings
# Space: O(m * n)

class Solution(object):
    def shortestDistance(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        def bfs(grid, dists, cnts, x, y):
            dist, m, n = 0, len(grid), len(grid[0])
            visited = [[False for _ in xrange(n)] for _ in xrange(m)]

            pre_level = [(x, y)]
            visited[x][y] = True
            while pre_level:
                dist += 1
                cur_level = []
                for i, j in pre_level:
                    for dir in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        I, J = i+dir[0], j+dir[1]
                        if 0 <= I < m and 0 <= J < n and grid[I][J] == 0 and not visited[I][J]:
                            cnts[I][J] += 1
                            dists[I][J] += dist
                            cur_level.append((I, J))
                            visited[I][J] = True

                pre_level = cur_level


        m, n, cnt = len(grid),  len(grid[0]), 0
        dists = [[0 for _ in xrange(n)] for _ in xrange(m)]
        cnts = [[0 for _ in xrange(n)] for _ in xrange(m)]
        for i in xrange(m):
            for j in xrange(n):
                if grid[i][j] == 1:
                    cnt += 1
                    bfs(grid, dists, cnts, i, j)

        shortest = float("inf")
        for i in xrange(m):
            for j in xrange(n):
                if dists[i][j] < shortest and cnts[i][j] == cnt:
                    shortest = dists[i][j]

        return shortest if shortest != float("inf") else -1
```

flip-game-ii.py

```
# flip-game-ii is not found.
# Time:  O(n + c^2)
# Space: O(c)

import itertools
import re


# The best theory solution (DP, O(n + c^2)) could be seen here:
# https://leetcode.com/discuss/64344/theory-matters-from-backtracking-128ms-to-dp-0m
class Solution(object):
    def canWin(self, s):
        g, g_final = [0], 0
        for p in itertools.imap(len, re.split('-+', s)):
            while len(g) <= p:
                # Theorem 2: g[game] = g[subgame1]^g[subgame2]^g[subgame3]...
                # and find first missing number.
                g += min(set(xrange(p)) - {x^y for x, y in itertools.izip(g[:len(g)/2], g[-2:-len(g)/2-2:-1])}
            g_final ^= g[p]
        return g_final > 0  # Theorem 1: First player must win iff g(current_state) != 0


# Time:  O(n + c^3 * 2^c * logc), n is length of string, c is count of "++"
# Space: O(c * 2^c)
# hash solution.
# We have total O(2^c) game strings,
# and each hash key in hash table would cost O(c),
# each one has O(c) choices to the next one,
# and each one would cost O(clogc) to sort,
# so we get O((c * 2^c) * (c * clogc)) = O(c^3 * 2^c * logc) time.
# To cache the results of all combinations, thus O(c * 2^c) space.
class Solution2(object):
    def canWin(self, s):
        """
        :type s: str
        :rtype: bool
        """
        lookup = {}

        def canWinHelper(consecutives):                                     # O(2^c) time
            consecutives = tuple(sorted(c for c in consecutives if c >= 2))     # O(clogc) time
            if consecutives not in lookup:
                lookup[consecutives] = any(not canWinHelper(consecutives[:i] + (j, c-2-j) + consecutives[i+1:]
                                           for i, c in enumerate(consecutives)  # O(c) time
                                           for j in xrange(c - 1))              # O(c) time
            return lookup[consecutives]                                         # O(c) time

        # re.findall: O(n) time, canWinHelper: O(c) in depth
        return canWinHelper(map(len, re.findall(r'\+\++', s)))


# Time:  O(c * n * c!), n is length of string, c is count of "++"
# Space: O(c * n), recursion would be called at most c in depth.
#                  Besides, it costs n space for modifying string at each depth.
class Solution3(object):
    def canWin(self, s):
        """
        :type s: str
```

```
        :rtype: bool
        """
        i, n = 0, len(s) - 1
        is_win = False
        while not is_win and i < n:                                     # O(n) time
            if s[i] == '+':
                while not is_win and i < n and s[i+1] == '+':           # O(c) time
                    # t(n, c) = c * (t(n, c-1) + n) + n = ...
                    # = c! * t(n, 0) + n * c! * (c + 1) * (1/0! + 1/1! + ... 1/c!)
                    # = n * c! + n * c! * (c + 1) * O(e) = O(c * n * c!)
                    is_win = not self.canWin(s[:i] + '--' + s[i+2:])    # O(n) space
                    i += 1
            i += 1
        return is_win
```

minimum-domino-rotations-for-equal-row.py

```
# DESC
# 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.
# Example 1:
# 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.)
# Note:
# Example 2:
# If it cannot be done, return -1.
# We may rotate the i-th domino, so that A[i] and B[i] swap values.

# NOTE
# 2 <= A.length == B.length <= 20000
# 1 <= A[i], B[i] <= 6

# EXAMPLE
# Input: A = [3,5,1,2,3], B = [3,6,3,3,4]
# Output: -1
# Explanation:
# In this case, i
# t is not possible to rotate the dominoes to make one row of values equal.
# Input: A = [2,1,2,4,2,2], B = [5,2,6,2,3,2]
# Output: 2
# Explanation:
# The first fi
# gure 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 ro
# w equal to 2, as indicated by the second figure.

# 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))
```

unique-email-addresses.py

```python
# DESC
# It is possible to use both of these rules at the same time.
# Besides lowercase letters, these emails may contain '.'s or '+'s.
# If you add periods ('.') between some characters in the local name part of an em
# ail address, mail sent there will be forwarded to the same address without dots
# in the local name.  For example, "alice.z@leetcode.com" and "alicez@leetcode.com
# " forward to the same email address.  (Note that this rule does not apply for do
# main names.)
# For example, in alice@leetcode.com, alice is the local name, and leetcode.com is
#  the domain name.
# Given a list of emails, we send one email to each address in the list.  How many
#  different addresses actually receive mails?
# Every email consists of a local name and a domain name, separated by the @ sign.
# Note:
# If you add a plus ('+') in the local name, everything after the first plus sign
# will be ignored. This allows certain emails to be filtered, for example m.y+name
# @email.com will be forwarded to my@email.com.  (Again, this rule does not apply
# for domain names.)
# Example 1:
# alice@leetcode.com

# NOTE
# Local names do not start with a '+' character.
# 1 <= emails[i].length <= 100
# All local and domain names are non-empty.
# 1 <= emails.length <= 100
# Each emails[i] contains exactly one '@' character.

# EXAMPLE
# Input: ["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","tes
# temail+david@lee.tcode.com"]
# Output: 2
# Explanation: "testemail@leetcode.com" and
#  "testemail@lee.tcode.com" actually receive mails

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

class Solution(object):
    def numUniqueEmails(self, emails):
        """
        :type emails: List[str]
        :rtype: int
        """
        def convert(email):
            name, domain = email.split('@')
            name = name[:name.index('+')]
            return "".join(["".join(name.split(".")), '@', domain])

        lookup = set()
        for email in emails:
            lookup.add(convert(email))
        return len(lookup)
```

reveal-cards-in-increasing-order.py

```
# DESC
# Example 1:
# The first entry in the answer is considered to be the top of the deck.
# Initially, all the cards start face down (unrevealed) in one deck.
# In a deck of cards, every card has a unique integer.  You can order the deck in
# any order you want.
# Now, you do the following steps repeatedly, until all cards are revealed:
# Note:
# Return an ordering of the deck that would reveal the cards in increasing order.

# NOTE
# Take the top card of the deck, reveal it, and take it out of the deck.
# 1 <= A[i] <= 10^6
# 1 <= A.length <= 1000
# A[i] != A[j] for all i != j
# If there are still unrevealed cards, go back to step 1.  Otherwise, stop.
# If there are still cards in the deck, put the next top card of the deck at the b
# ottom of the deck.

# EXAMPLE
# Input: [17,13,11,2,3,5,7]
# Output: [2,13,3,11,5,17,7]
# Explanation:
# We get the de
# ck 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 t
# he deck.
# We reveal 2, and move 13 to the bottom.  The deck is now [3,11,5,17,7,1
# 3].
# 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 bot
# tom.  The deck is now [17].
# We reveal 17.
# Since all the cards revealed are in in
# creasing order, the answer is correct.

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

import collections


class Solution(object):
    def deckRevealedIncreasing(self, deck):
        """
        :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)
```

subarray-product-less-than-k.py

```python
# DESC
# Note:
# Your are given an array of positive integers nums.
# Count and print the number of (contiguous) subarrays where the product of all th
# e elements in the subarray is less than k.
# Example 1:

# NOTE
# 0 <= k < 10^6.
# 0 < nums[i] < 1000.
# 0 < nums.length <= 50000.

# EXAMPLE
# Input: nums = [10, 5, 2, 6], k = 100
# Output: 8
# Explanation: The 8 subarrays that
#  have product less than 100 are: [10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [
# 5, 2, 6].
# Note that [10, 5, 2] is not included as the product of 100 is not stri
# ctly less than k.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numSubarrayProductLessThanK(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        if k <= 1: return 0
        result, start, prod = 0, 0, 1
        for i, num in enumerate(nums):
            prod *= num
            while prod >= k:
                prod /= nums[start]
                start += 1
            result += i-start+1
        return result
```

lonely-pixel-ii.py

```
# lonely-pixel-ii is not found.
# Time:  O(m * n)
# Space: O(m * n)

import collections


class Solution(object):
    def findBlackPixel(self, picture, N):
        """
        :type picture: List[List[str]]
        :type N: int
        :rtype: int
        """
        rows, cols = [0] * len(picture),  [0] * len(picture[0])
        lookup = collections.defaultdict(int)
        for i in xrange(len(picture)):
            for j in xrange(len(picture[0])):
                if picture[i][j] == 'B':
                    rows[i] += 1
                    cols[j] += 1
            lookup[tuple(picture[i])] += 1

        result = 0
        for i in xrange(len(picture)):
            if rows[i] == N and lookup[tuple(picture[i])] == N:
                for j in xrange(len(picture[0])):
                     result += picture[i][j] == 'B' and cols[j] == N
        return result


class Solution2(object):
    def findBlackPixel(self, picture, N):
        """
        :type picture: List[List[str]]
        :type N: int
        :rtype: int
        """
        lookup = collections.Counter(map(tuple, picture))
        cols = [col.count('B') for col in zip(*picture)]
        return sum(N * zip(row, cols).count(('B', N)) \
                   for row, cnt in lookup.iteritems() \
                   if cnt == N == row.count('B'))
```

construct-the-rectangle.py

```python
# DESC
# Note:
# For a web developer, it is very important to know how to design a web page's siz
# e. So, given a specific rectangular web page’s area, your job by now is to desig
# n a rectangular web page, whose length L and width W satisfy the following requi
# rements:
# Example:

# NOTE
# The given area won't exceed 10,000,000 and is a positive integer
# The web page's width and length you designed must be positive integers.

# EXAMPLE
# 1. The area of the rectangular web page you designed must equal to the given tar
# get area.
#
# 2. The width W should not be larger than the length L, which means L
# >= W.
#
# 3. The difference between length L and width W should be as small as poss
# ible.
# Input: 4
# Output: [2, 2]
# Explanation: The target area is 4, and all the possible
# ways to construct it are [1,4], [2,2], [4,1].
# But according to requirement 2, [
# 1,4] is illegal; according to requirement 3,  [4,1] is not optimal compared to [
# 2,2]. So the length L is 2, and the width W is 2.

# Time:  O(1)
# Space: O(1)

import math


class Solution(object):
    def constructRectangle(self, area):
        """
        :type area: int
        :rtype: List[int]
        """
        w = int(math.sqrt(area))
        while area % w:
            w -= 1
        return [area // w, w]
```

fruit-into-baskets.py

```
# DESC
# 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 o
# n until you stop.
# What is the total amount of fruit you can collect with this procedure?
# You have two baskets, and each basket can carry any quantity of fruit, but you w
# ant each basket to only carry one type of fruit each.
# Example 2:
# Note:
# Example 4:
# You start at any tree of your choice, then repeatedly perform the following steps:
# In a row of trees, the i-th tree produces fruit with type tree[i].
# Example 1:
# Example 3:

# NOTE
# 0 <= tree[i] < tree.length
# 1 <= tree.length <= 40000
# 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.

# EXAMPLE
# Input: [1,2,3,2,2]
# Output: 4
# Explanation: We can collect [2,3,2,2].
# If we starte
# d at the first tree, we would only collect [1, 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].
# Input: [1,2,1]
# Output: 3
# Explanation: We can collect [1,2,1].
# 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 fr
# uits.

# 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
```

closest-binary-search-tree-value-ii.py

```
# closest-binary-search-tree-value-ii is not found.
# Time:  O(h + k)
# Space: O(h)

class Solution(object):
    def closestKValues(self, root, target, k):
        """
        :type root: TreeNode
        :type target: float
        :type k: int
        :rtype: List[int]
        """
        # Helper to make a stack to the next node.
        def nextNode(stack, child1, child2):
            if stack:
                if child2(stack):
                    stack.append(child2(stack))
                    while child1(stack):
                        stack.append(child1(stack))
                else:
                    child = stack.pop()
                    while stack and child is child2(stack):
                        child = stack.pop()

        # The forward or backward iterator.
        backward = lambda stack: stack[-1].left
        forward = lambda stack: stack[-1].right

        # Build the stack to the closest node.
        stack = []
        while root:
            stack.append(root)
            root = root.left if target < root.val else root.right
        dist = lambda node: abs(node.val - target)
        forward_stack = stack[:stack.index(min(stack, key=dist))+1]

        # Get the stack to the next smaller node.
        backward_stack = list(forward_stack)
        nextNode(backward_stack, backward, forward)

        # Get the closest k values by advancing the iterators of the stacks.
        result = []
        for _ in xrange(k):
            if forward_stack and \
                (not backward_stack or dist(forward_stack[-1]) < dist(backward_stack[-1])):
                result.append(forward_stack[-1].val)
                nextNode(forward_stack, forward, backward)
            elif backward_stack and \
                (not forward_stack or dist(backward_stack[-1]) <= dist(forward_stack[-1])):
                result.append(backward_stack[-1].val)
                nextNode(backward_stack, backward, forward)
        return result


class Solution2(object):
    def closestKValues(self, root, target, k):
        """
        :type root: TreeNode
```

```
        :type target: float
        :type k: int
        :rtype: List[int]
        """
        # Helper class to make a stack to the next node.
        class BSTIterator:
            # @param root, a binary search tree's root node
            def __init__(self, stack, child1, child2):
                self.stack = list(stack)
                self.cur = self.stack.pop()
                self.child1 = child1
                self.child2 = child2

            # @return an integer, the next node
            def next(self):
                node = None
                if self.cur and self.child1(self.cur):
                    self.stack.append(self.cur)
                    node = self.child1(self.cur)
                    while self.child2(node):
                        self.stack.append(node)
                        node = self.child2(node)
                elif self.stack:
                    prev = self.cur
                    node = self.stack.pop()
                    while node:
                        if self.child2(node) is prev:
                            break
                        else:
                            prev = node
                            node = self.stack.pop() if self.stack else None
                self.cur = node
                return node

        # Build the stack to the closet node.
        stack = []
        while root:
            stack.append(root)
            root = root.left if target < root.val else root.right
        dist = lambda node: abs(node.val - target) if node else float("inf")
        stack = stack[:stack.index(min(stack, key=dist))+1]

        # The forward or backward iterator.
        backward = lambda node: node.left
        forward = lambda node: node.right
        smaller_it, larger_it = BSTIterator(stack, backward, forward), BSTIterator(stack, forward, backward)
        smaller_node, larger_node = smaller_it.next(), larger_it.next()

        # Get the closest k values by advancing the iterators of the stacks.
        result = [stack[-1].val]
        for _ in xrange(k - 1):
            if dist(smaller_node) < dist(larger_node):
                result.append(smaller_node.val)
                smaller_node = smaller_it.next()
            else:
                result.append(larger_node.val)
                larger_node = larger_it.next()
        return result
```

max-chunks-to-make-sorted-ii.py

```
# DESC
# Note:
# Given an array arr of integers (not necessarily distinct), we split the array in
# to some number of "chunks" (partitions), and individually sort each chunk.  Afte
# r concatenating them, the result equals the sorted array.
# This question is the same as "Max Chunks to Make Sorted" except the integers of
# the given array are not necessarily distinct, the input array could be up to len
# gth 2000, and the elements could be up to 10**8.
# Example 2:
# What is the most number of chunks we could have made?
# Example 1:

# NOTE
# arr will have length in range [1, 2000].
# arr[i] will be an integer in range [0, 10**8].

# EXAMPLE
# Input: arr = [5,4,3,2,1]
# Output: 1
# Explanation:
# Splitting into two or more chunk
# s will not return the required result.
# For example, splitting into [5, 4], [3, 2
# , 1] will result in [4, 5, 1, 2, 3], which isn't sorted.
# Input: arr = [2,1,3,4,4]
# Output: 4
# Explanation:
# We can split into two chunks, su
# ch as [2, 1], [3, 4, 4].
# However, splitting into [2, 1], [3], [4], [4] is the hi
# ghest number of chunks possible.

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

class Solution(object):
    def maxChunksToSorted(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        def compare(i1, i2):
            return arr[i1]-arr[i2] if arr[i1] != arr[i2] else i1-i2

        idxs = [i for i in xrange(len(arr))]
        result, max_i = 0, 0
        for i, v in enumerate(sorted(idxs, cmp=compare)):
            max_i = max(max_i, v)
            if max_i == i:
                result += 1
        return result
```

optimize-water-distribution-in-a-village.py

```python
# optimize-water-distribution-in-a-village is not found.
# Time:  O(nlogn)
# Space: O(n)

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:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        self.count -= 1
        return True


class Solution(object):
    def minCostToSupplyWater(self, n, wells, pipes):
        """
        :type n: int
        :type wells: List[int]
        :type pipes: List[List[int]]
        :rtype: int
        """
        w = [[c, 0, i] for i, c in enumerate(wells, 1)]
        p = [[c, i, j] for i, j, c in pipes]
        result = 0
        union_find = UnionFind(n+1)
        for c, x, y in sorted(w+p):
            if not union_find.union_set(x, y):
                continue
            result += c
            if union_find.count == 1:
                break
        return result
```

valid-anagram.py

```
# DESC
# Follow up:
#
# What if the inputs contain unicode characters? How would you adapt y
# our solution to such case?
# Note:
#
# You may assume the string contains only lowercase alphabets.
# Given two strings s and t , write a function to determine if t is an anagram of s.
# Example 1:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: s = "rat", t = "car"
# Output: false
# Input: s = "anagram", t = "nagaram"
# Output: true

# Time:  O(n)
# Space: O(1)

import collections
import string


class Solution(object):
    # @param {string} s
    # @param {string} t
    # @return {boolean}
    def isAnagram(self, s, t):
        if len(s) != len(t):
            return False
        count = collections.defaultdict(int)
        for c in s:
            count[c] += 1
        for c in t:
            count[c] -= 1
            if count[c] < 0:
                return False
        return True


# Time:  O(nlogn)
# Space: O(n)
class Solution2(object):
    # @param {string} s
    # @param {string} t
    # @return {boolean}
    def isAnagram(self, s, t):
        return sorted(s) == sorted(t)


# Time:  O(n)
# Space: O(n)
class Solution3(object):
```

```python
# @param {string} s
# @param {string} t
# @return {boolean}
def isAnagram(self, s, t):
    return collections.Counter(s) == collections.Counter(t)
```

binary-tree-preorder-traversal.py

```
# binary-tree-preorder-traversal is not found.
# 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):
        """
        :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
```

climbing-stairs.py

```
# DESC
# You are climbing a stair case. It takes n steps to reach to the top.
# Example 1:
# Example 2:
# Each time you can either climb 1 or 2 steps. In how many distinct ways can you c
# limb to the top?
# Constraints:

# NOTE
# 1 <= n <= 45

# EXAMPLE
# Input: 3
# Output: 3
# Explanation: There are three ways to climb to the top.
# 1. 1 s
# tep + 1 step + 1 step
# 2. 1 step + 2 steps
# 3. 2 steps + 1 step
# Input: 2
# Output: 2
# Explanation: There are two ways to climb to the top.
# 1. 1 ste
# p + 1 step
# 2. 2 steps

# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def climbStairs(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)) \
                     for col in ZB] for row in A]

        T = [[1, 1],
             [1, 0]]
        return matrix_mult([[1,  0]], matrix_expo(T, n))[0][0]  # [a0, a(-1)] * T^n
```

```python
# Time:  O(n)
# Space: O(1)
class Solution2(object):
    """
    :type n: int
    :rtype: int
    """
    def climbStairs(self, n):
        prev, current = 0, 1
        for i in xrange(n):
            prev, current = current, prev + current,
        return current
```

encode-number.py

```python
# encode-number is not found.
# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def encode(self, num):
        """
        :type num: int
        :rtype: str
        """
        result = []
        while num:
            result.append('0' if num%2 else '1')
            num = (num-1)//2
        return "".join(reversed(result))
```

print-immutable-linked-list-in-reverse.py

```python
# print-immutable-linked-list-in-reverse is not found.
# Time:  O(n)
# Space: O(sqrt(n))

import math


class Solution(object):
    def printLinkedListInReverse(self, head):
        """
        :type head: ImmutableListNode
        :rtype: None
        """
        def print_nodes(head, count):
            nodes = []
            while head and len(nodes) != count:
                nodes.append(head)
                head = head.getNext()
            for node in reversed(nodes):
                node.printValue()

        count = 0
        curr = head
        while curr:
            curr = curr.getNext()
            count += 1
        bucket_count = int(math.ceil(count**0.5))

        buckets = []
        count = 0
        curr = head
        while curr:
            if count % bucket_count == 0:
                buckets.append(curr)
            curr = curr.getNext()
            count += 1
        for node in reversed(buckets):
            print_nodes(node, bucket_count)


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def printLinkedListInReverse(self, head):
        """
        :type head: ImmutableListNode
        :rtype: None
        """
        nodes = []
        while head:
            nodes.append(head)
            head = head.getNext()
        for node in reversed(nodes):
            node.printValue()


# Time:  O(n^2)
# Space: O(1)
```

```
class Solution3(object):
    def printLinkedListInReverse(self, head):
        """
        :type head: ImmutableListNode
        :rtype: None
        """
        tail = None
        while head != tail:
            curr = head
            while curr.getNext() != tail:
                curr = curr.getNext()
            curr.printValue()
            tail = curr
```

reverse-words-in-a-string.py

```
# DESC
# Example 1:
# Example 2:
# For C programmers, try to solve it in-place in O(1) extra space.
# Given an input string, reverse the string word by word.
# Follow up:
# Example 3:
# Note:

# NOTE
# A word is defined as a sequence of non-space characters.
# Input string may contain leading or trailing spaces. However, your reversed stri
# ng should not contain leading or trailing spaces.
# You need to reduce multiple spaces between two words to a single space in the re
# versed string.

# EXAMPLE
# Input: "  hello world!  "
# Output: "world! hello"
# Explanation: Your reversed stri
# ng should not contain leading or trailing spaces.
# Input: "the sky is blue"
# Output: "blue is sky the"
# Input: "a good   example"
# Output: "example good a"
# Explanation: You need to redu
# ce multiple spaces between two words to a single space in the reversed string.

# 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()))
```

maximum-score-after-splitting-a-string.py

```python
# maximum-score-after-splitting-a-string is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxScore(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, zeros, ones = 0, 0, 0
        for i in xrange(1, len(s)-1):
            if s[i] == '0':
                zeros += 1
            else:
                ones += 1
            result = max(result, zeros-ones)
        return result + ones + (s[0] == '0') + (s[-1] == '1')
```

swap-nodes-in-pairs.py

```
# DESC
# You may not modify the values in the list's nodes, only nodes itself may be changed.
# Given a linked list, swap every two adjacent nodes and return its head.
# Example:

# NOTE
#

# 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
```

last-substring-in-lexicographical-order.py

```
# DESC
# Example 1:
# Example 2:
# Given a string s, return the last substring of s in lexicographical order.
# Note:

# NOTE
# s contains only lowercase English letters.
# 1 <= s.length <= 4 * 10^5

# EXAMPLE
# Input: "abab"
# Output: "bab"
# Explanation: The substrings are ["a", "ab", "aba", "
# abab", "b", "ba", "bab"]. The lexicographically maximum substring is "bab".
# Input: "leetcode"
# Output: "tcode"

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

import collections


class Solution(object):
    def lastSubstring(self, s):
        """
        :type s: str
        :rtype: str
        """
        count = collections.defaultdict(list)
        for i in xrange(len(s)):
            count[s[i]].append(i)

        max_c = max(count.iterkeys())
        starts = {}
        for i in count[max_c]:
            starts[i] = i+1
        while len(starts)-1 > 0:
            lookup = set()
            next_count = collections.defaultdict(list)
            for start, end in starts.iteritems():
                if end == len(s):  # finished
                    lookup.add(start)
                    continue
                next_count[s[end]].append(start)
                if end in starts:  # overlapped
                    lookup.add(end)
            next_starts = {}
            max_c = max(next_count.iterkeys())
            for start in next_count[max_c]:
                if start not in lookup:
                    next_starts[start] = starts[start]+1
            starts = next_starts
        return s[next(starts.iterkeys()):]
```

convert-sorted-list-to-binary-search-tree.py

```
# DESC
# Example:
# For this problem, a height-balanced binary tree is defined as a binary tree in w
# hich the depth of the two subtrees of every node never differ by more than 1.
# Given a singly linked list where elements are sorted in ascending order, convert
#  it to a height balanced BST.

# NOTE
#

# EXAMPLE
# Given the sorted linked list: [-10,-3,0,5,9],
#
# One possible answer is: [0,-3,9,-
# 10,null,5], which represents the following height balanced BST:
#
#       0
#      /
# \
#    -3   9
#    /   /
#  -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):
        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

```
# DESC
# A clear path from top-left to bottom-right has length k if and only if it is com
# posed of cells C_1, C_2, ..., C_k such that:
# Return the length of the shortest such clear path from top-left to bottom-right.
#   If such a path does not exist, return -1.
# Example 1:
# In an N by N square grid, each cell is either empty (0) or blocked (1).
# Note:
# Example 2:

# NOTE
# C_1 is at location (0, 0) (ie. has value grid[0][0])
# 1 <= grid.length == grid[0].length <= 100
# Adjacent cells C_i and C_{i+1} are connected 8-directionally (ie., they are diff
# erent and share an edge or corner)
# If C_i is located at (r, c), then grid[r][c] is empty (ie. grid[r][c] == 0).
# grid[r][c] is 0 or 1
# C_k is at location (N-1, N-1) (ie. has value grid[N-1][N-1])

# EXAMPLE
# Input: [[0,0,0],[1,1,0],[1,1,0]]
#
#
# Output: 4
# Input: [[0,1],[1,0]]
#
#
# Output: 2

# 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
```

palindrome-permutation.py

```python
# palindrome-permutation is not found.
# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def canPermutePalindrome(self, s):
        """
        :type s: str
        :rtype: bool
        """
        return sum(v % 2 for v in collections.Counter(s).values()) < 2
```

sum-of-even-numbers-after-queries.py

```python
# DESC
# We have an array A of integers, and an array queries of queries.
# (Here, the given index = queries[i][1] is a 0-based index, and each query perman
# ently modifies the array A.)
# Return the answer to all queries.  Your answer array should have answer[i] as th
# e answer to the i-th query.
# Example 1:
# For the i-th query val = queries[i][0], index = queries[i][1], we add val to A[i
# ndex].  Then, the answer to the i-th query is the sum of the even values of A.
# Note:

# NOTE
# 0 <= queries[i][1] < A.length
# -10000 <= A[i] <= 10000
# 1 <= queries.length <= 10000
# 1 <= A.length <= 10000
# -10000 <= queries[i][0] <= 10000

# EXAMPLE
# Input: A = [1,2,3,4], queries = [[1,0],[-3,1],[-4,0],[2,3]]
# Output: [8,6,2,4]
# Ex
# planation:
# At the beginning, the array is [1,2,3,4].
# After adding 1 to A[0], th
# e array is [2,2,3,4], and the sum of even values is 2 + 2 + 4 = 8.
# After adding
# -3 to A[1], the array is [2,-1,3,4], and the sum of even values is 2 + 4 = 6.
# Af
# ter adding -4 to A[0], the array is [-2,-1,3,4], and the sum of even values is -
# 2 + 4 = 2.
# After adding 2 to A[3], the array is [-2,-1,3,6], and the sum of even
#  values is -2 + 6 = 4.

# Time:  O(n + q)
# Space: O(1)

class Solution(object):
    def sumEvenAfterQueries(self, A, queries):
        """
        :type A: List[int]
        :type queries: List[List[int]]
        :rtype: List[int]
        """
        total = sum(v for v in A if v % 2 == 0)

        result = []
        for v, i in queries:
            if A[i] % 2 == 0:
                total -= A[i]
            A[i] += v
            if A[i] % 2 == 0:
                total += A[i]
            result.append(total)
        return result
```

cousins-in-binary-tree.py

```python
# DESC
# Return true if and only if the nodes corresponding to the values x and y are cousins.
# We are given the root of a binary tree with unique values, and the values x and
# y of two different nodes in the tree.
# Example 2:
# Example 3:
# In a binary tree, the root node is at depth 0, and children of each depth k node
#  are at depth k+1.
# Example 1:
# Constraints:
# Two nodes of a binary tree are cousins if they have the same depth, but have dif
# ferent parents.

# NOTE
# Each node has a unique integer value from 1 to 100.
# The number of nodes in the tree will be between 2 and 100.

# EXAMPLE
# Input: root = [1,2,3,null,4], x = 2, y = 3
# Output: false
# Input: root = [1,2,3,null,4,null,5], x = 5, y = 4
# Output: true
# Input: root = [1,2,3,4], x = 4, y = 3
# Output: false

# 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 isCousins(self, root, x, y):
        """
        :type root: TreeNode
        :type x: int
        :type y: int
        :rtype: bool
        """
        def dfs(root, x, depth, parent):
            if not root:
                return False
            if root.val == x:
                return True
            depth[0] += 1
            prev_parent, parent[0] = parent[0], root
            if dfs(root.left, x, depth, parent):
                return True
            parent[0] = root
            if dfs(root.right, x, depth, parent):
                return True
            parent[0] = prev_parent
            depth[0] -= 1
```

```
    return False

depth_x, depth_y = [0], [0]
parent_x, parent_y = [None], [None]
return dfs(root, x, depth_x, parent_x) and \
       dfs(root, y, depth_y, parent_y) and \
       depth_x[0] == depth_y[0] and \
       parent_x[0] != parent_y[0]
```

palindrome-partitioning-ii.py

```
# DESC
# Example:
# Given a string s, partition s such that every substring of the partition is a pa
# lindrome.
# Return the minimum cuts needed for a palindrome partitioning of s.

# NOTE
#

# EXAMPLE
# Input: "aab"
# Output: 1
# Explanation: The palindrome partitioning ["aa","b"] could
#  be produced using 1 cut.

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

class Solution(object):
    # @param s, a string
    # @return an integer
    def minCut(self, s):
        lookup = [[False for j in xrange(len(s))] for i in xrange(len(s))]
        mincut = [len(s) - 1 - i for i in xrange(len(s) + 1)]

        for i in reversed(xrange(len(s))):
            for j in xrange(i, len(s)):
                if s[i] == s[j]  and (j - i < 2 or lookup[i + 1][j - 1]):
                    lookup[i][j] = True
                    mincut[i] = min(mincut[i], mincut[j + 1] + 1)

        return mincut[0]
```

find-smallest-letter-greater-than-target.py

```
# DESC
# target = 'z'
# Note:
# Examples:
# Given a list of sorted characters letters containing only lowercase letters, and
#  given a target letter target, find the smallest element in the list that is lar
# ger than the given target.
# Letters also wrap around.  For example, if the target is target = 'z' and letter
# s = ['a', 'b'], the answer is 'a'.

# NOTE
# letters consists of lowercase letters, and contains at least 2 unique letters.
# target is a lowercase letter.
# letters has a length in range [2, 10000].

# EXAMPLE
# Input:
# letters = ["c", "f", "j"]
# target = "a"
# Output: "c"
#
# Input:
# letters = ["c"
# , "f", "j"]
# target = "c"
# Output: "f"
#
# Input:
# letters = ["c", "f", "j"]
# target =
# "d"
# Output: "f"
#
# Input:
# letters = ["c", "f", "j"]
# target = "g"
# Output: "j"
#
# Inpu
# t:
# letters = ["c", "f", "j"]
# target = "j"
# Output: "c"
#
# Input:
# letters = ["c", "f
# ", "j"]
# target = "k"
# Output: "c"

# Time:  O(logn)
# Space: O(1)

import bisect


class Solution(object):
    def nextGreatestLetter(self, letters, target):
```

```
        """
        :type letters: List[str]
        :type target: str
        :rtype: str
        """
        i = bisect.bisect_right(letters, target)
        return letters[0] if i == len(letters) else letters[i]
```

number-of-enclaves.py

```python
# DESC
# Given a 2D array A, each cell is 0 (representing sea) or 1 (representing land)
# Note:
# Example 2:
# Return the number of land squares in the grid for which we cannot walk off the b
# oundary of the grid in any number of moves.
# Example 1:
# A move consists of walking from one land square 4-directionally to another land
# square, or off the boundary of the grid.

# NOTE
# 1 <= A[i].length <= 500
# 0 <= A[i][j] <= 1
# All rows have the same size.
# 1 <= A.length <= 500

# EXAMPLE
# 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.
# Input: [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
# Output: 3
# Explanation:
# There a
# re three 1s that are enclosed by 0s, and one 1 that isn't enclosed because its o
# n the boundary.

# 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

```
# DESC
# Example:
# Given a 2D board and a word, find if the word exists in the grid.
# Constraints:
# The word can be constructed from letters of sequentially adjacent cell, where "a
# djacent" cells are those horizontally or vertically neighboring. The same letter
#  cell may not be used more than once.

# NOTE
# 1 <= board[i].length <= 200
# 1 <= board.length <= 200
# board and word consists only of lowercase and uppercase English letters.
# 1 <= word.length <= 10^3

# 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 = "AB
# CB", return false.

# 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
```

form-largest-integer-with-digits-that-add-up-to-target.py

```python
# form-largest-integer-with-digits-that-add-up-to-target is not found.
# Time:  O(t)
# Space: O(t)

class Solution(object):
    def largestNumber(self, cost, target):
        """
        :type cost: List[int]
        :type target: int
        :rtype: str
        """
        dp = [0]
        for t in xrange(1, target+1):
            dp.append(-1)
            for i, c in enumerate(cost):
                if t-c < 0 or dp[t-c] < 0:
                    continue
                dp[t] = max(dp[t], dp[t-c]+1)
        if dp[target] < 0:
            return "0"
        result = []
        for i in reversed(xrange(9)):
            while target >= cost[i] and dp[target] == dp[target-cost[i]]+1:
                target -= cost[i]
                result.append(i+1)
        return "".join(map(str, result))


# Time:  O(t)
# Space: O(t)
class Solution2(object):
    def largestNumber(self, cost, target):
        """
        :type cost: List[int]
        :type target: int
        :rtype: str
        """
        def key(bag):
            return sum(bag), bag

        dp = [[0]*9]
        for t in xrange(1, target+1):
            dp.append([])
            for d, c in enumerate(cost):
                if t < c or not dp[t-c]:
                    continue
                curr = dp[t-c][:]
                curr[~d] += 1
                if key(curr) > key(dp[t]):
                    dp[-1] = curr
        if not dp[-1]:
            return "0"
        return "".join(str(9-i)*c for i, c in enumerate(dp[-1]))


# Time:  O(t^2)
# Space: O(t^2)
class Solution3(object):
```

```
def largestNumber(self, cost, target):
    """
    :type cost: List[int]
    :type target: int
    :rtype: str
    """
    dp = [0]
    for t in xrange(1, target+1):
        dp.append(-1)
        for i, c in enumerate(cost):
            if t-c < 0:
                continue
            dp[t] = max(dp[t], dp[t-c]*10 + i+1)
    return str(max(dp[t], 0))
```

count-submatrices-with-all-ones.py

```python
# count-submatrices-with-all-ones is not found.
# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    def numSubmat(self, mat):
        """
        :type mat: List[List[int]]
        :rtype: int
        """
        def count(heights):
            dp, stk = [0]*len(heights), []
            for i in xrange(len(heights)):
                while stk and heights[stk[-1]] >= heights[i]:
                    stk.pop()
                dp[i] = dp[stk[-1]] + heights[i]*(i-stk[-1]) if stk else heights[i]*(i-(-1))
                stk.append(i)
            return sum(dp)

        result = 0
        heights = [0]*len(mat[0])
        for i in xrange(len(mat)):
            for j in xrange(len(mat[0])):
                heights[j] = heights[j]+1 if mat[i][j] == 1 else 0
            result += count(heights)
        return result
```

string-transforms-into-another-string.py

```python
# string-transforms-into-another-string is not found.
# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def canConvert(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: bool
        """
        if str1 == str2:
            return True
        lookup = {}
        for i, j in itertools.izip(str1, str2):
            if lookup.setdefault(i, j) != j:
                return False
        return len(set(str2)) < 26
```

bricks-falling-when-hit.py

```
# DESC
# Return an array representing the number of bricks that will drop after each eras
# ure in sequence.
# We have a grid of 1s and 0s; the 1s in a cell represent bricks.  A brick will no
# t drop if and only if it is directly connected to the top of the grid, or at lea
# st one of its (4-way) adjacent bricks will not drop.
# We will do some erasures sequentially. Each time we want to do the erasure at th
# e location (i, j), the brick (if it exists) on that location will disappear, and
#  then some other bricks may drop because of that erasure.
# Note:

# NOTE
# The number of erasures will not exceed the area of the grid.
# An erasure may refer to a location with no brick - if it does, no bricks drop.
# The number of rows and columns in the grid will be in the range [1, 200].
# It is guaranteed that each erasure will be different from any other erasure, and
#  located inside the grid.

# EXAMPLE
# Example 1:
# Input:
# grid = [[1,0,0,0],[1,1,1,0]]
# hits = [[1,0]]
# Output: [2]
# Expla
# nation:
# If we erase the brick at (1, 0), the brick at (1, 1) and (1, 2) will dr
# op. So we should return 2.
# Example 2:
# Input:
# grid = [[1,0,0,0],[1,1,0,0]]
# hits = [[1,1],[1,0]]
# Output: [0,
# 0]
# Explanation:
# When we erase the brick at (1, 0), the brick at (1, 1) has alre
# ady disappeared due to the last move. So each erasure will cause no bricks dropp
# ing.  Note that the erased brick (1, 0) will not be counted as a dropped brick.

# Time:  O(r * c)
# Space: O(r * c)

class UnionFind(object):
    def __init__(self, n):
        self.set = range(n+1)
        self.size = [1]*(n+1)
        self.size[-1] = 0

    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)
```

```
        self.size[max(x_root, y_root)] += self.size[min(x_root, y_root)]
        return True

    def top(self):
        return self.size[self.find_set(len(self.size)-1)]


class Solution(object):
    def hitBricks(self, grid, hits):
        """
        :type grid: List[List[int]]
        :type hits: List[List[int]]
        :rtype: List[int]
        """
        def index(C, r, c):
            return r*C+c

        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
        R, C = len(grid), len(grid[0])

        hit_grid = [row[:] for row in grid]
        for i, j in hits:
            hit_grid[i][j] = 0

        union_find = UnionFind(R*C)
        for r, row in enumerate(hit_grid):
            for c, val in enumerate(row):
                if not val:
                    continue
                if r == 0:
                    union_find.union_set(index(C, r, c), R*C)
                if r and hit_grid[r-1][c]:
                    union_find.union_set(index(C, r, c), index(C, r-1, c))
                if c and hit_grid[r][c-1]:
                    union_find.union_set(index(C, r, c), index(C, r, c-1))

        result = []
        for r, c in reversed(hits):
            prev_roof = union_find.top()
            if grid[r][c] == 0:
                result.append(0)
                continue
            for d in directions:
                nr, nc = (r+d[0], c+d[1])
                if 0 <= nr < R and 0 <= nc < C and hit_grid[nr][nc]:
                    union_find.union_set(index(C, r, c), index(C, nr, nc))
            if r == 0:
                union_find.union_set(index(C, r, c), R*C)
            hit_grid[r][c] = 1
            result.append(max(0, union_find.top()-prev_roof-1))
        return result[::-1]
```

iterator-for-combination.py

```
# DESC
# Example:
# Constraints:
# Design an Iterator class, which has:

# NOTE
# It's guaranteed that all calls of the function next are valid.
# 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 combin
# ation.
# There will be at most 10^4 function calls per test.
# A constructor that takes a string characters of sorted distinct lowercase Englis
# h letters and a number combinationLength as arguments.
# 1 <= combinationLength <= characters.length <= 15

# 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.n
# ext(); // returns "bc"
# iterator.hasNext(); // returns false

# 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):
        """
        :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
        """
        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()
```

unique-letter-string.py

```python
# unique-letter-string is not found.
# Time:  O(n)
# Space: O(1)

import string


class Solution(object):
    def uniqueLetterString(self, S):
        """
        :type S: str
        :rtype: int
        """
        M = 10**9 + 7
        index = {c: [-1, -1] for c in string.ascii_uppercase}
        result = 0
        for i, c in enumerate(S):
            k, j = index[c]
            result += (i-j) * (j-k)
            index[c] = [j, i]
        for c in index:
            k, j = index[c]
            result += (len(S)-j) * (j-k)
        return result % M
```

complete-binary-tree-inserter.py

```
# DESC
# A complete binary tree is a binary tree in which every level, except possibly th
# e last, is completely filled, and all nodes are as far left as possible.
# Example 2:
# Write a data structure CBTInserter that is initialized with a complete binary tr
# ee and supports the following operations:
# Note:
# CBTInserter
# Example 1:

# NOTE
# CBTInserter.insert is called at most 10000 times per test case.
# CBTInserter.insert(int v) will insert a TreeNode into the tree with value node.v
# al = v so that the tree remains complete, and returns the value of the parent of
#  the inserted TreeNode;
# CBTInserter(TreeNode root) initializes the data structure on a given tree with h
# ead node root;
# The initial given tree is complete and contains between 1 and 1000 nodes.
# CBTInserter.get_root() will return the head node of the tree.
# Every value of a given or inserted node is between 0 and 5000.

# EXAMPLE
# Input: inputs = ["CBTInserter","insert","get_root"], inputs = [[[1]],[2],[]]
# Out
# put: [null,1,[1,2]]
# 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]]

# 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]
```

count-univalue-subtrees.py

```
# count-univalue-subtrees is not found.
# Time:  O(n)
# Space: O(h)

class Solution(object):
    # @param {TreeNode} root
    # @return {integer}
    def countUnivalSubtrees(self, root):
        [is_uni, count] = self.isUnivalSubtrees(root, 0)
        return count

    def isUnivalSubtrees(self, root, count):
        if not root:
            return [True, count]

        [left, count] = self.isUnivalSubtrees(root.left, count)
        [right, count] = self.isUnivalSubtrees(root.right, count)
        if self.isSame(root, root.left, left) and \
           self.isSame(root, root.right, right):
                count += 1
                return [True, count]

        return [False, count]

    def isSame(self, root, child, is_uni):
        return not child or (is_uni and root.val == child.val)
```

count-binary-substrings.py

```
# DESC
# Give a string s, count the number of non-empty (contiguous) substrings that have
#  the same number of 0's and 1's, and all the 0's and all the 1's in these substr
# ings are grouped consecutively.
# Note:
# Substrings that occur multiple times are counted the number of times they occur.
# Example 1:
# Example 2:

# NOTE
# s.length will be between 1 and 50,000.
# s will only consist of "0" or "1" characters.

# EXAMPLE
# Input: "00110011"
# Output: 6
# Explanation: There are 6 substrings that have equal
# number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".
#
#
# Notice that some of these substrings repeat and are counted the number of time
# s they occur.
#
# Also, "00110011" is not a valid substring because all the 0's (an
# d 1's) are not grouped together.
# Input: "10101"
# Output: 4
# Explanation: There are 4 substrings: "10", "01", "10",
# "01" that have equal number of consecutive 1's and 0's.

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countBinarySubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, prev, curr = 0, 0, 1
        for i in xrange(1, len(s)):
            if s[i-1] != s[i]:
                result += min(prev, curr)
                prev, curr = curr, 1
            else:
                curr += 1
        result += min(prev, curr)
        return result
```

print-foobar-alternately.py

```
# print-foobar-alternatel is not found.
# 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()
```

triples-with-bitwise-and-equal-to-zero.py

```
# DESC
# Given an array of integers A, find the number of triples of indices (i, j, k) su
# ch that:
# Note:
# Example 1:

# NOTE
# 0 <= k < A.length
# 1 <= A.length <= 1000
# 0 <= i < A.length
# 0 <= A[i] < 2^16
# A[i] & A[j] & A[k] == 0, where & represents the bitwise-AND operator.
# 0 <= j < A.length

# EXAMPLE
# Input: [2,1,3]
# Output: 12
# Explanation: We could choose the following i, j, k tri
# ples:
# (i=0, j=0, k=1) : 2 & 2 & 1
# (i=0, j=1, k=0) : 2 & 1 & 2
# (i=0, j=1, k=1) :
# 2 & 1 & 1
# (i=0, j=1, k=2) : 2 & 1 & 3
# (i=0, j=2, k=1) : 2 & 3 & 1
# (i=1, j=0, k=0
# ) : 1 & 2 & 2
# (i=1, j=0, k=1) : 1 & 2 & 1
# (i=1, j=0, k=2) : 1 & 2 & 3
# (i=1, j=1,
#  k=0) : 1 & 1 & 2
# (i=1, j=2, k=0) : 1 & 3 & 2
# (i=2, j=0, k=1) : 3 & 2 & 1
# (i=2,
# j=1, k=0) : 3 & 1 & 2

# Time:  O(nlogn), n is the max of A
# Space: O(n)

import collections


# Reference: https://blog.csdn.net/john123741/article/details/76576925
# FWT solution
class Solution(object):
    def countTriplets(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        def FWT(A, v):
            B = A[:]
            d = 1
            while d < len(B):
                for i in xrange(0, len(B), d << 1):
                    for j in xrange(d):
                        B[i+j] += B[i+j+d] * v
                d <<= 1
```

```
            return B

        k = 3
        n, max_A = 1, max(A)
        while n <= max_A:
            n *= 2
        count = collections.Counter(A)
        B = [count[i] for i in xrange(n)]
        C = FWT(map(lambda x : x**k, FWT(B, 1)), -1)
        return C[0]


# Time:  O(n^3), n is the length of A
# Space: O(n^2)
import collections


class Solution2(object):
    def countTriplets(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        count = collections.defaultdict(int)
        for i in xrange(len(A)):
            for j in xrange(len(A)):
                count[A[i]&A[j]] += 1
        result = 0
        for k in xrange(len(A)):
            for v in count:
                if A[k]&v == 0:
                    result += count[v]
        return result
```

minimum-remove-to-make-valid-parentheses.py

```
# DESC
# Given a string s of '(' , ')' and lowercase English characters.
# Example 2:
# Example 1:
# Formally, a parentheses string is valid if and only if:
# Your task is to remove the minimum number of parentheses ( '(' or ')', in any po
# sitions ) so that the resulting parentheses string is valid and return any valid
#  string.
# Example 3:
# Example 4:
# Constraints:

# NOTE
# It is the empty string, contains only lowercase characters, or
# 1 <= s.length <= 10^5
# It can be written as (A), where A is a valid string.
# s[i] is one of  '(' , ')' and lowercase English letters.
# It can be written as AB (A concatenated with B), where A and B are valid strings, or

# EXAMPLE
# Input: s = "))(("
# Output: ""
# Explanation: An empty string is also valid.
# 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.
# Input: s = "a)b(c)d"
# Output: "ab(c)d"
# Input: s = "(a(b(c)d)"
# Output: "a(b(c)d)"

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

class Solution(object):
    def minRemoveToMakeValid(self, s):
        """
        :type s: str
        :rtype: str
        """
        result = list(s)
        count = 0
        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)
```

n-ary-tree-postorder-traversal.py

```
# DESC
# Example 1:
# Follow up:
# Constraints:
# Example 2:
# Nary-Tree input serialization is represented in their level order traversal, eac
# h group of children is separated by the null value (See examples).
# Given an n-ary tree, return the postorder traversal of its nodes' values.
# Recursive solution is trivial, could you do it iteratively?

# NOTE
# The height of the n-ary tree is less than or equal to 1000
# The total number of nodes is between [0, 10^4]

# EXAMPLE
# Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,1
# 2,null,13,null,null,14]
# Output: [2,6,14,11,7,3,12,8,4,13,9,10,5,1]
# Input: root = [1,null,3,2,4,null,5,6]
# Output: [5,6,3,2,4,1]

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

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


class Solution(object):
    def postorder(self, root):
        """
        :type root: Node
        :rtype: List[int]
        """
        if not root:
            return []
        result, stack = [], [root]
        while stack:
            node = stack.pop()
            result.append(node.val)
            for child in node.children:
                if child:
                    stack.append(child)
        return result[::-1]


class Solution2(object):
    def postorder(self, root):
        """
        :type root: Node
        :rtype: List[int]
        """
        def dfs(root, result):
            for child in root.children:
                if child:
                    dfs(child, result)
```

```
    result.append(root.val)

result = []
if root:
    dfs(root, result)
return result
```

stone-game.py

```
# DESC
# Note:
# Assuming Alex and Lee play optimally, return True if and only if Alex wins the game.
# Alex and Lee take turns, with Alex starting first.  Each turn, a player takes th
# e entire pile of stones from either the beginning or the end of the row.  This c
# ontinues until there are no more piles left, at which point the person with the
# most stones wins.
# Example 1:
# The objective of the game is to end with the most stones.  The total number of s
# tones is odd, so there are no ties.
# Alex and Lee play a game with piles of stones.  There are an even number of pile
# s arranged in a row, and each pile has a positive integer number of stones piles
# [i].

# NOTE
# 1 <= piles[i] <= 500
# 2 <= piles.length <= 500
# piles.length is even.
# sum(piles) is odd.

# EXAMPLE
# Input: [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 t
# o win with 9 points.
# This demonstrated that taking the first 5 was a winning mov
# e for Alex, so we return true.

# 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
```

can-place-flowers.py

```
# DESC
# Example 2:
# Note:
# Given a flowerbed (represented as an array containing 0 and 1, where 0 means emp
# ty and 1 means not empty), and a number n, return if n new flowers can be plante
# d in it without violating the no-adjacent-flowers rule.
# Example 1:
# Suppose you have a long flowerbed in which some of the plots are planted and som
# e are not. However, flowers cannot be planted in adjacent plots - they would com
# pete for water and both would die.

# NOTE
# The input array won't violate no-adjacent-flowers rule.
# The input array size is in the range of [1, 20000].
# n is a non-negative integer which won't exceed the input array size.

# EXAMPLE
# Input: flowerbed = [1,0,0,0,1], n = 2
# Output: False
# Input: flowerbed = [1,0,0,0,1], n = 1
# Output: True

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def canPlaceFlowers(self, flowerbed, n):
        """
        :type flowerbed: List[int]
        :type n: int
        :rtype: bool
        """
        for i in xrange(len(flowerbed)):
            if flowerbed[i] == 0 and (i == 0 or flowerbed[i-1] == 0) and \
                (i == len(flowerbed)-1 or flowerbed[i+1] == 0):
                flowerbed[i] = 1
                n -= 1
            if n <= 0:
                return True
        return False
```

factor-combinations.py

```python
# factor-combinations is not found.
# Time:  O(nlogn)
# Space: O(logn)

class Solution(object):
    # @param {integer} n
    # @return {integer[][]}
    def getFactors(self, n):
        result = []
        factors = []
        self.getResult(n, result, factors)
        return result

    def getResult(self, n, result, factors):
        i = 2 if not factors else factors[-1]
        while i <= n / i:
            if n % i == 0:
                factors.append(i)
                factors.append(n / i)
                result.append(list(factors))
                factors.pop()
                self.getResult(n / i, result, factors)
                factors.pop()
            i += 1
```

longest-substring-without-repeating-characters.py

```python
# DESC
# Example 1:
# Given a string, find the length of the longest substring without repeating chara
# cters.
# Example 3:
# Example 2:

# NOTE
#

# EXAMPLE
# 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 subsequenc
# e and not a substring.
# Input: "abcabcbb"
# Output: 3
# Explanation: The answer is "abc", with the length of 3.
# Input: "bbbbb"
# Output: 1
# Explanation: The answer is "b", with the length of 1.

# 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
```

find-the-minimum-number-of-fibonacci-numbers-whose-sum-is-k.py

```python
# find-the-minimum-number-of-fibonacci-numbers-whose-sum-is-k is not found.
# Time:  O(logk)
# Space: O(1)

class Solution(object):
    def findMinFibonacciNumbers(self, k):
        """
        :type k: int
        :rtype: int
        """
        result, a, b = 0, 1, 1
        while b <= k:
            b, a = a+b, b
        while k:
            if a <= k:
                k -= a
                result += 1
            a, b = b-a, a
        return result
```

encode-n-ary-tree-to-binary-tree.py

```
# encode-n-ary-tree-to-binary-tree is not found.
# Time:  O(n)
# Space: O(h)

class Node(object):
    def __init__(self, val, children):
        self.val = val
        self.children = children


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


class Codec(object):

    def encode(self, root):
        """Encodes an n-ary tree to a binary tree.

        :type root: Node
        :rtype: TreeNode
        """
        def encodeHelper(root, parent, index):
            if not root:
                return None
            node = TreeNode(root.val)
            if index+1 < len(parent.children):
                node.left = encodeHelper(parent.children[index+1], parent, index+1)
            if root.children:
                node.right = encodeHelper(root.children[0], root, 0)
            return node

        if not root:
            return None
        node = TreeNode(root.val)
        if root.children:
            node.right = encodeHelper(root.children[0], root, 0)
        return node

    def decode(self, data):
        """Decodes your binary tree to an n-ary tree.

        :type data: TreeNode
        :rtype: Node
        """
        def decodeHelper(root, parent):
            if not root:
                return
            children = []
            node = Node(root.val, children)
            decodeHelper(root.right, node)
            parent.children.append(node)
            decodeHelper(root.left, parent)
```

```
if not data:
    return None
children = []
node = Node(data.val, children)
decodeHelper(data.right, node)
return node
```

design-search-autocomplete-system.py

```
# design-search-autocomplete-system is not found.
# Time:  O(p^2), p is the length of the prefix
# Space: O(p * t + s), t is the number of nodes of trie
#                    , s is the size of the sentences

import collections


class TrieNode(object):

    def __init__(self):
        self.__TOP_COUNT = 3
        self.infos = []
        self.leaves = {}


    def insert(self, s, times):
        cur = self
        cur.add_info(s, times)
        for c in s:
            if c not in cur.leaves:
                cur.leaves[c] = TrieNode()
            cur = cur.leaves[c]
            cur.add_info(s, times)


    def add_info(self, s, times):
        for p in self.infos:
            if p[1] == s:
                p[0] = -times
                break
        else:
            self.infos.append([-times, s])
        self.infos.sort()
        if len(self.infos) > self.__TOP_COUNT:
            self.infos.pop()


class AutocompleteSystem(object):

    def __init__(self, sentences, times):
        """
        :type sentences: List[str]
        :type times: List[int]
        """
        self.__trie = TrieNode()
        self.__cur_node = self.__trie
        self.__search = []
        self.__sentence_to_count = collections.defaultdict(int)
        for sentence, count in zip(sentences, times):
            self.__sentence_to_count[sentence] = count
            self.__trie.insert(sentence, count)


    def input(self, c):
        """
        :type c: str
        :rtype: List[str]
```

```
        """
        result = []
        if c == '#':
            self.__sentence_to_count["".join(self.__search)] += 1
            self.__trie.insert("".join(self.__search), self.__sentence_to_count["".join(self.__search)])
            self.__cur_node = self.__trie
            self.__search = []
        else:
            self.__search.append(c)
            if self.__cur_node:
                if c not in self.__cur_node.leaves:
                    self.__cur_node = None
                    return []
                self.__cur_node = self.__cur_node.leaves[c]
                result = [p[1] for p in self.__cur_node.infos]
        return result
```

number-of-good-pairs.py

```python
# number-of-good-pairs is not found.
# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def numIdenticalPairs(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return sum(c*(c-1)//2 for c in collections.Counter(nums).itervalues())
```

longest-common-prefix.py

```
# DESC
# Write a function to find the longest common prefix string amongst an array of st
# rings.
# All given inputs are in lowercase letters a-z.
# Example 1:
# If there is no common prefix, return an empty string "".
# Note:
# Example 2:

# NOTE
#

# EXAMPLE
# Input: ["dog","racecar","car"]
# Output: ""
# Explanation: There is no common prefix
#  among the input strings.
# Input: ["flower","flow","flight"]
# Output: "fl"

# Time:  O(n * k), k is the length of the common prefix
# Space: O(1)

class Solution(object):
    def longestCommonPrefix(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        if not strs:
            return ""

        for i in xrange(len(strs[0])):
            for string in strs[1:]:
                if i >= len(string) or string[i] != strs[0][i]:
                    return strs[0][:i]
        return strs[0]


# Time:  O(n * k), k is the length of the common prefix
# Space: O(k)
class Solution2(object):
    def longestCommonPrefix(self, strs):
        """
        :type strs: List[str]
        :rtype: str
        """
        prefix = ""

        for chars in zip(*strs):
            if all(c == chars[0] for c in chars):
                prefix += chars[0]
            else:
                return prefix

        return prefix
```

combination-sum-ii.py

```
# DESC
# Note:
# Example 2:
# Given a collection of candidate numbers (candidates) and a target number (target
# ), find all unique combinations in candidates where the candidate numbers sums t
# o target.
# Each number in candidates may only be used once in the combination.
# candidates
# Example 1:

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

# EXAMPLE
# 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]
# ]
# 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
```

permutation-sequence.py

```
# DESC
# Note:
# By listing and labeling all of the permutations in order, we get the following s
# equence for n = 3:
# Example 1:
# The set [1,2,3,...,n] contains a total of n! unique permutations.
# Given n and k, return the kth permutation sequence.
# Example 2:

# NOTE
# "123"
# "132"
# Given k will be between 1 and n! inclusive.
# Given n will be between 1 and 9 inclusive.
# "312"
# "213"
# "231"
# "321"

# EXAMPLE
# Input: n = 4, k = 9
# Output: "2314"
# Input: n = 3, k = 3
# Output: "213"

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

import math

# Cantor ordering solution
class Solution(object):
    def getPermutation(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: str
        """
        seq, k, fact = "", k - 1, math.factorial(n - 1)
        perm = [i for i in xrange(1, n + 1)]
        for i in reversed(xrange(n)):
            curr = perm[k / fact]
            seq += str(curr)
            perm.remove(curr)
            if i > 0:
                k %= fact
                fact /= i
        return seq
```

relative-ranks.py

```
# DESC
# Example 1:
# Note:
# Given scores of N athletes, find their relative ranks and the people with the to
# p three highest scores, who will be awarded medals: "Gold Medal", "Silver Medal"
#  and "Bronze Medal".

# NOTE
# All the scores of athletes are guaranteed to be unique.
# N is a positive integer and won't exceed 10,000.

# EXAMPLE
# Input: [5, 4, 3, 2, 1]
# Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4
# ", "5"]
# Explanation: The first three athletes got the top three highest scores,
# so they got "Gold Medal", "Silver Medal" and "Bronze Medal".
# For the left two a
# thletes, you just need to output their relative ranks according to their scores.

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

class Solution(object):
    def findRelativeRanks(self, nums):
        """
        :type nums: List[int]
        :rtype: List[str]
        """
        sorted_nums = sorted(nums)[::-1]
        ranks = ["Gold Medal", "Silver Medal", "Bronze Medal"] + map(str, range(4, len(nums) + 1))
        return map(dict(zip(sorted_nums, ranks)).get, nums)
```

minimum-absolute-difference.py

```
# minimum-absolute-difference is not found.
# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def minimumAbsDifference(self, arr):
        """
        :type arr: List[int]
        :rtype: List[List[int]]
        """
        result = []
        min_diff = float("inf")
        arr.sort()
        for i in xrange(len(arr)-1):
            diff = arr[i+1]-arr[i]
            if diff < min_diff:
                min_diff = diff
                result = [[arr[i], arr[i+1]]]
            elif diff == min_diff:
                result.append([arr[i], arr[i+1]])
        return result
```

hand-of-straights.py

```
# DESC
# Example 1:
# Now she wants to rearrange the cards into groups so that each group is size W, a
# nd consists of W consecutive cards.
# Constraints:
# Example 2:
# Alice has a hand of cards, given as an array of integers.
# Return true if and only if she can.

# NOTE
# 1 <= hand.length <= 10000
# 1 <= W <= hand.length
# 0 <= hand[i] <= 10^9

# EXAMPLE
# Input: hand = [1,2,3,4,5], W = 4
# Output: false
# Explanation: Alice's hand can't b
# e rearranged into groups of 4.
# Input: hand = [1,2,3,6,2,3,4,7,8], W = 3
# Output: true
# Explanation: Alice's hand
# can be rearranged as [1,2,3],[2,3,4],[6,7,8].

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

from collections import Counter
from heapq import heapify, heappop


class Solution(object):
    def isNStraightHand(self, hand, W):
        """
        :type hand: List[int]
        :type W: int
        :rtype: bool
        """
        if len(hand) % W:
            return False

        counts = Counter(hand)
        min_heap = list(hand)
        heapify(min_heap)
        for _ in xrange(len(min_heap)//W):
            while counts[min_heap[0]] == 0:
                heappop(min_heap)
            start = heappop(min_heap)
            for _ in xrange(W):
                counts[start] -= 1
                if counts[start] < 0:
                    return False
                start += 1
        return True
```

task-scheduler.py

```
# DESC
# You are given a char array representing tasks CPU need to do. It contains capita
# l letters A to Z where each letter represents a different task. Tasks could be d
# one without the original order of the array. Each task is done in one unit of ti
# me. For each unit of time, the CPU could complete either one task or just be idl
# e.
# Example 3:
# Example 1:
# You need to return the least number of units of times that the CPU will take to
# finish all the given tasks.
# Example 2:
# However, there is a non-negative integer n that represents the cooldown period b
# etween 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.
# Constraints:

# NOTE
# The number of tasks is in the range [1, 10000].
# The integer n is in the range [0, 100].

# EXAMPLE
# 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
# 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.
# Input: tasks = ["A","A","A","B","B","B"], n = 0
# Output: 6
# Explanation: On this c
# ase 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.

# Time:  O(n)
# Space: O(26) = O(1)

from collections import Counter


class Solution(object):
    def leastInterval(self, tasks, n):
        """
        :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))
```

number-of-ways-to-wear-different-hats-to-each-other.py

```python
# number-of-ways-to-wear-different-hats-to-each-other is not found.
# Time:  O(h * 2^n)
# Space: O(2^n)

class Solution(object):
    def numberWays(self, hats):
        """
        :type hats: List[List[int]]
        :rtype: int
        """
        MOD = 10**9 + 7
        HAT_SIZE = 40
        hat_to_people = [[] for _ in xrange(HAT_SIZE)]
        for i in xrange(len(hats)):
            for h in hats[i]:
                hat_to_people[h-1].append(i)
        dp = [0]*(1<<len(hats))
        dp[0] = 1
        for people in hat_to_people:
            for mask in reversed(xrange(len(dp))):
                for p in people:
                    if mask & (1<<p):
                        continue
                    dp[mask | (1<<p)] += dp[mask]
                    dp[mask | (1<<p)] %= MOD
        return dp[-1]
```

maximize-sum-of-array-after-k-negations.py

```
# DESC
# Example 3:
# Example 2:
# Return the largest possible sum of the array after modifying it in this way.
# Example 1:
# Note:
# Given an array A of integers, we must modify the array in the following way: we
# choose an i and replace A[i] with -A[i], and we repeat this process K times in t
# otal.  (We may choose the same index i multiple times.)

# NOTE
# 1 <= K <= 10000
# 1 <= A.length <= 10000
# -100 <= A[i] <= 100

# EXAMPLE
# Input: A = [3,-1,0,2], K = 3
# Output: 6
# Explanation: Choose indices (1, 2, 2) and
#  A becomes [3,1,0,2].
# Input: A = [2,-3,-1,5,-4], K = 2
# Output: 13
# Explanation: Choose indices (1, 4) a
# nd A becomes [2,3,-1,5,4].
# Input: A = [4,2,3], K = 1
# Output: 5
# Explanation: Choose indices (1,) and A becom
# es [4,-2,3].

# Time:  O(n) ~ O(n^2), O(n) on average.
# Space: O(1)

import random


# quick select solution
class Solution(object):
    def largestSumAfterKNegations(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        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:
                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

        kthElement(A, K, lambda a, b: a < b)
        remain = K
        for i in xrange(K):
            if A[i] < 0:
                A[i] = -A[i]
                remain -= 1
        return sum(A) - ((remain)%2)*min(A)*2


# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def largestSumAfterKNegations(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        A.sort()
        remain = K
        for i in xrange(K):
            if A[i] >= 0:
                break
            A[i] = -A[i]
            remain -= 1
        return sum(A) - (remain%2)*min(A)*2
```

check-if-a-word-occurs-as-a-prefix-of-any-word-in-a-sentence.py

```python
# check-if-a-word-occurs-as-a-prefix-of-any-word-in-a-sentence is not found.
# Time:  O(n)
# Space: O(n)

class Solution(object):
    def isPrefixOfWord(self, sentence, searchWord):
        """
        :type sentence: str
        :type searchWord: str
        :rtype: int
        """
        def KMP(text, pattern):
            def getPrefix(pattern):
                prefix = [-1] * len(pattern)
                j = -1
                for i in xrange(1, len(pattern)):
                    while j > -1 and pattern[j + 1] != pattern[i]:
                        j = prefix[j]
                    if pattern[j + 1] == pattern[i]:
                        j += 1
                    prefix[i] = j
                return prefix

            prefix = getPrefix(pattern)
            j = -1
            for i in xrange(len(text)):
                while j != -1 and pattern[j+1] != text[i]:
                    j = prefix[j]
                if pattern[j+1] == text[i]:
                    j += 1
                if j+1 == len(pattern):
                    return i-j
            return -1

        if sentence.startswith(searchWord):
            return 1
        p = KMP(sentence, ' ' + searchWord)
        if p == -1:
            return -1
        return 1+sum(sentence[i] == ' ' for i in xrange(p+1))
```

binary-prefix-divisible-by-5.py

```
# DESC
# Given an array A of 0s and 1s, consider N_i: the i-th subarray from A[0] to A[i]
#  interpreted as a binary number (from most-significant-bit to least-significant-
# bit.)
# Example 3:
# answer
# Note:
# Example 4:
# Example 1:
# Example 2:
# Return a list of booleans answer, where answer[i] is true if and only if N_i is
# divisible by 5.

# NOTE
# 1 <= A.length <= 30000
# A[i] is 0 or 1

# EXAMPLE
# Input: [1,1,1,0,1]
# Output: [false,false,false,false,false]
# Input: [0,1,1]
# Output: [true,false,false]
# Explanation:
# The input numbers in bin
# ary are 0, 01, 011; which are 0, 1, and 3 in base-10.  Only the first number is
# divisible by 5, so answer[0] is true.
# Input: [0,1,1,1,1,1]
# Output: [true,false,false,false,true,false]
# Input: [1,1,1]
# Output: [false,false,false]

# Time:  O(n)
# Space: O(1)

class Solution(object):
    def prefixesDivBy5(self, A):
        """
        :type A: List[int]
        :rtype: List[bool]
        """
        for i in xrange(1, len(A)):
            A[i] += A[i-1] * 2 % 5
        return [x % 5 == 0 for x in A]
```

confusing-number-ii.py

```
# confusing-number-ii is not found.
# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def confusingNumberII(self, N):
        """
        :type N: int
        :rtype: int
        """
        lookup = {"0":"0", "1":"1", "6":"9", "8":"8", "9":"6"}
        centers = {"0":"0", "1":"1", "8":"8"}

        def totalCount(N):
            s = str(N)
            total = 0
            p = len(lookup)**(len(s)-1)
            for i in xrange(len(s)):
                if i+1 == len(s):
                    for c in lookup.iterkeys():
                        total += int(c <= s[i])
                    continue
                smaller = 0
                for c in lookup.iterkeys():
                    smaller += int(c < s[i])
                total += smaller * p
                if s[i] not in lookup:
                    break
                p //= len(lookup)
            return total

        def validCountInLessLength(N):
            s = str(N)
            valid = 0
            total = len(centers)
            for i in xrange(1, len(s), 2):
                if i == 1:
                    valid += total
                else:
                    valid += total * (len(lookup)-1)
                    total *= len(lookup)
            total = 1
            for i in xrange(2, len(s), 2):
                valid += total * (len(lookup)-1)
                total *= len(lookup)
            return valid

        def validCountInFullLength(N):
            s = str(N)
            half_s = s[:(len(s)+1)//2]
            total = 0
            p =  len(lookup)**(len(half_s)-2) * len(centers) if (len(s) % 2) else len(lookup)**(len(half_s)-1)
            choices = centers if (len(s) % 2) else lookup
            for i in xrange(len(half_s)):
                if i+1 == len(half_s):
                    for c in choices.iterkeys():
                        if c == '0' and i == 0:
                            continue
```

```
                total += int(c < half_s[i])
            if half_s[i] not in choices:
                break
            tmp = list(half_s)
            for i in reversed(xrange(len(half_s)-(len(s) % 2))):
                tmp.append(lookup[half_s[i]])
            if int("".join(tmp)) <= N:
                total += 1
            continue

        smaller = 0
        for c in lookup.iterkeys():
            if c == '0' and i == 0:
                continue
            smaller += int(c < half_s[i])
        total += smaller * p
        if half_s[i] not in lookup:
            break
        p //= len(lookup)
    return total

return totalCount(N) - validCountInLessLength(N) - validCountInFullLength(N)
```

fair-candy-swap.py

```python
# fair-candy-swa is not found.
# Time:  O(m + n)
# Space: O(m + n)

class Solution(object):
    def fairCandySwap(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: List[int]
        """
        diff = (sum(A)-sum(B))//2
        setA = set(A)
        for b in set(B):
            if diff+b in setA:
                return [diff+b, b]
        return []
```

find-minimum-in-rotated-sorted-array.py

```python
# find-minimum-in-rotated-sorted-arra is not found.
# 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]
```

next-closest-time.py

```python
# next-closest-time is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def nextClosestTime(self, time):
        """
        :type time: str
        :rtype: str
        """
        h, m = time.split(":")
        curr = int(h) * 60 + int(m)
        result = None
        for i in xrange(curr+1, curr+1441):
            t = i % 1440
            h, m = t // 60, t % 60
            result = "%02d:%02d" % (h, m)
            if set(result) <= set(time):
                break
        return result
```

arranging-coins.py

```
# DESC
# You have a total of n coins that you want to form in a staircase shape, where ev
# ery k-th row must have exactly k coins.
# Given n, find the total number of full staircase rows that can be formed.
# Example 2:
# Example 1:
# n is a non-negative integer and fits within the range of a 32-bit signed integer.

# NOTE
#

# EXAMPLE
# n = 5
#
# The coins can form the following rows:
# ¤
# ¤ ¤
# ¤ ¤
#
# Because the 3rd row is
# incomplete, we return 2.
# n = 8
#
# The coins can form the following rows:
# ¤
# ¤ ¤
# ¤ ¤ ¤
# ¤ ¤
#
# Because the 4th r
# ow is incomplete, we return 3.

# Time:  O(logn)
# Space: O(1)

import math


class Solution(object):
    def arrangeCoins(self, n):
        """
        :type n: int
        :rtype: int
        """
        return int((math.sqrt(8*n+1)-1) / 2)  # sqrt is O(logn) time.


# Time:  O(logn)
# Space: O(1)
class Solution2(object):
    def arrangeCoins(self, n):
        """
        :type n: int
        :rtype: int
        """
        def check(mid, n):
            return mid*(mid+1) <= 2*n
```

```
left, right = 1, n
while left <= right:
    mid = left + (right-left)//2
    if not check(mid, n):
        right = mid-1
    else:
        left = mid+1
return right
```

maximum-width-of-binary-tree.py

```
# DESC
# Example 4:
# 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.
# Example 1:
# Example 3:
# Example 2:
# Constraints:
# The width of one level is defined as the length between the end-nodes (the leftm
# ost 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.

# NOTE
# The given binary tree will have between 1 and 3000 nodes.

# EXAMPLE
# Input:
#
#           1
#          /
#         3
#        / \
#       5   3
#
#
# Output: 2
# Explanation: The maximum width existing in the third level with th
# e length 2 (5,3).
# Input:
#
#           1
#          / \
#         3   2
#        /     \
#       5       9
#
#      /         \
#     6           7
# Output: 8
# Explanation:The maximum width exi
# sting in the fourth level with the length 8 (6,null,null,null,null,null,null,7).
# Input:
#
#           1
#          / \
#         3   2
#        /
#       5
#
#
# Output: 2
# Explanation: The maximum width existing in the second level with the
# length 2 (3,2).
# 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).

# 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)
```

maximum-vacation-days.py

```python
# maximum-vacation-days is not found.
# Time:  O(n^2 * k)
# Space: O(k)

class Solution(object):
    def maxVacationDays(self, flights, days):
        """
        :type flights: List[List[int]]
        :type days: List[List[int]]
        :rtype: int
        """
        if not days or not flights:
            return 0
        dp = [[0] * len(days) for _ in xrange(2)]
        for week in reversed(xrange(len(days[0]))):
            for cur_city in xrange(len(days)):
                dp[week % 2][cur_city] = days[cur_city][week] + dp[(week+1) % 2][cur_city]
                for dest_city in xrange(len(days)):
                    if flights[cur_city][dest_city] == 1:
                        dp[week % 2][cur_city] = max(dp[week % 2][cur_city], \
                                                     days[dest_city][week] + dp[(week+1) % 2][dest_city])
        return dp[0][0]
```

design-skiplist.py

```
# design-skiplist is not found.
# Time:  O(logn) on average for each operation
# Space: O(n)

# see proof in references:
# 1. https://kunigami.blog/2012/09/25/skip-lists-in-python/
# 2. https://opendatastructures.org/ods-cpp/4_4_Analysis_Skiplists.html
# 3. https://brilliant.org/wiki/skip-lists/
import random


class SkipNode(object):
    def __init__(self, level=0, num=None):
        self.num = num
        self.nexts = [None]*level


class Skiplist(object):
    P_NUMERATOR, P_DENOMINATOR = 1, 2  # P = 1/4 in redis implementation
    MAX_LEVEL = 32  # enough for 2^32 elements

    def __init__(self):
        self.__head = SkipNode()
        self.__len = 0

    def search(self, target):
        """
        :type target: int
        :rtype: bool
        """
        return True if self.__find(target, self.__find_prev_nodes(target)) else False

    def add(self, num):
        """
        :type num: int
        :rtype: None
        """
        node = SkipNode(self.__random_level(), num)
        if len(self.__head.nexts) < len(node.nexts):
            self.__head.nexts.extend([None]*(len(node.nexts)-len(self.__head.nexts)))
        prevs = self.__find_prev_nodes(num)
        for i in xrange(len(node.nexts)):
            node.nexts[i] = prevs[i].nexts[i]
            prevs[i].nexts[i] = node
        self.__len += 1

    def erase(self, num):
        """
        :type num: int
        :rtype: bool
        """
        prevs = self.__find_prev_nodes(num)
        curr = self.__find(num, prevs)
        if not curr:
            return False
        self.__len -= 1
        for i in reversed(xrange(len(curr.nexts))):
            prevs[i].nexts[i] = curr.nexts[i]
```

```
            if not self.__head.nexts[i]:
                self.__head.nexts.pop()
        return True

    def __find(self, num, prevs):
        if prevs:
            candidate = prevs[0].nexts[0]
            if candidate and candidate.num == num:
                return candidate
        return None

    def __find_prev_nodes(self, num):
        prevs = [None]*len(self.__head.nexts)
        curr = self.__head
        for i in reversed(xrange(len(self.__head.nexts))):
            while curr.nexts[i] and curr.nexts[i].num < num:
                curr = curr.nexts[i]
            prevs[i] = curr
        return prevs

    def __random_level(self):
        level = 1
        while random.randint(1, Skiplist.P_DENOMINATOR) <= Skiplist.P_NUMERATOR and \
              level < Skiplist.MAX_LEVEL:
            level += 1
        return level

    def __len__(self):
        return self.__len

    def __str__(self):
        result = []
        for i in reversed(xrange(len(self.__head.nexts))):
            result.append([])
            curr = self.__head.nexts[i]
            while curr:
                result[-1].append(str(curr.num))
                curr = curr.nexts[i]
        return "\n".join(map(lambda x: "->".join(x), result))
```

number-of-boomerangs.py

```python
# DESC
# Find the number of boomerangs. You may assume that n will be at most 500 and coo
# rdinates of points are all in the range [-10000, 10000] (inclusive).
# Example:
# Given n points in the plane that are all pairwise distinct, a "boomerang" is a t
# uple of points (i, j, k) such that the distance between i and j equals the dista
# nce between i and k (the order of the tuple matters).

# NOTE
#

# EXAMPLE
# Input:
# [[0,0],[1,0],[2,0]]
#
# Output:
# 2
#
# Explanation:
# The two boomerangs are [[1,0
# ],[0,0],[2,0]] and [[1,0],[2,0],[0,0]]

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

import collections


class Solution(object):
    def numberOfBoomerangs(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        result = 0

        for i in xrange(len(points)):
            group = collections.defaultdict(int)
            for j in xrange(len(points)):
                if j == i:
                    continue
                dx, dy =  points[i][0] - points[j][0], points[i][1] - points[j][1]
                group[dx**2 + dy**2] += 1

            for _, v in group.iteritems():
                if v > 1:
                    result += v * (v-1)

        return result

    def numberOfBoomerangs2(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        cnt = 0
        for a, i in enumerate(points):
            dis_list = []
```

```
    for b, k in enumerate(points[:a] + points[a + 1:]):
        dis_list.append((k[0] - i[0]) ** 2 + (k[1] - i[1]) ** 2)
    for z in collections.Counter(dis_list).values():
        if z > 1:
            cnt += z * (z - 1)
return cnt
```

shortest-completing-word.py

```
# DESC
# Example 2:
# Find the minimum length word from a given dictionary words, which has all the le
# tters from the string licensePlate.  Such a word is said to complete the given s
# tring licensePlate
# The license plate might have the same letter occurring multiple times.  For exam
# ple, given a licensePlate of "PP", the word "pair" does not complete the license
# Plate, but the word "supper" does.
# Example 1:
# It is guaranteed an answer exists.  If there are multiple answers, return the on
# e that occurs first in the array.
# Here, for letters we ignore case.  For example, "P" on the licensePlate still ma
# tches "p" on the word.
# Note:

# NOTE
# licensePlate will be a string with length in range [1, 7].
# words will have a length in the range [10, 1000].
# Every words[i] will consist of lowercase letters, and have length in range [1, 15].
# licensePlate will contain digits, spaces, or letters (uppercase or lowercase).

# EXAMPLE
# Input: licensePlate = "1s3 456", words = ["looks", "pest", "stew", "show"]
# Outpu
# t: "pest"
# Explanation: There are 3 smallest length words that contains the lette
# rs "s".
# We return the one that occurred first.
# Input: licensePlate = "1s3 PSt", words = ["step", "steps", "stripe", "stepple"]
#
# Output: "steps"
# Explanation: The smallest length word that contains the letters
# "S", "P", "S", and "T".
# Note that the answer is not "step", because the letter "
# s" must occur in the word twice.
# Also note that we ignored case for the purposes
#  of comparing whether a letter exists in the word.

# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def shortestCompletingWord(self, licensePlate, words):
        """
        :type licensePlate: str
        :type words: List[str]
        :rtype: str
        """
        def contains(counter1, w2):
            c2 = collections.Counter(w2.lower())
            c2.subtract(counter1)
            return all(map(lambda x: x >= 0, c2.values()))

        result = None
        counter = collections.Counter(c.lower() for c in licensePlate if c.isalpha())
```

```
for word in words:
    if (result is None or (len(word) < len(result))) and \
       contains(counter, word):
        result = word
return result
```

linked-list-components.py

```python
# DESC
# Return the number of connected components in G, where two values are connected i
# f they appear consecutively in the linked list.
# Example 2:
# Note:
# Example 1:
# We are also given the list G, a subset of the values in the linked list.
# We are given head, the head node of a linked list containing unique integer values.

# NOTE
# If N is the length of the linked list given by head, 1 <= N <= 10000.
# The value of each node in the linked list will be in the range [0, N - 1].
# G is a subset of all values in the linked list.
# 1 <= G.length <= 10000.

# EXAMPLE
# Input:
# head: 0->1->2->3
# G = [0, 1, 3]
# Output: 2
# Explanation:
# 0 and 1 are conne
# cted, so [0, 1] and [3] are the two connected components.
# Input:
# head: 0->1->2->3->4
# G = [0, 3, 1, 4]
# Output: 2
# Explanation:
# 0 and 1 are
#  connected, 3 and 4 are connected, so [0, 1] and [3, 4] are the two connected co
# mponents.

# Time:  O(m + n), m is the number of G, n is the number of nodes
# Space: O(m)

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


class Solution(object):
    def numComponents(self, head, G):
        """
        :type head: ListNode
        :type G: List[int]
        :rtype: int
        """
        lookup = set(G)
        dummy = ListNode(-1)
        dummy.next = head
        curr = dummy
        result = 0
        while curr and curr.next:
            if curr.val not in lookup and curr.next.val in lookup:
                result += 1
            curr = curr.next
        return result
```

word-search-ii.py

```
# DESC
# Given a 2D board and a list of words from the dictionary, find all words in the
# board.
# Each word must be constructed from letters of sequentially adjacent cell, where
# "adjacent" cells are those horizontally or vertically neighboring. The same lett
# er cell may not be used more than once in a word.
# Note:
# Example:

# NOTE
# The values of words are distinct.
# All inputs are consist of lowercase letters a-z.

# EXAMPLE
# Input:
# board = [
#   ['o','a','a','n'],
#   ['e','t','a','e'],
#   ['i','h','k','r'],
#
#   ['i','f','l','v']
# ]
# words = ["oath","pea","eat","rain"]
#
# Output: ["eat","oath
# "]

# Time:  O(m * n * l)
# Space: O(l)

class TrieNode(object):
    # Initialize your data structure here.
    def __init__(self):
        self.is_string = False
        self.leaves = {}

    # Inserts a word into the trie.
    def insert(self, word):
        cur = self
        for c in word:
            if not c in cur.leaves:
                cur.leaves[c] = TrieNode()
            cur = cur.leaves[c]
        cur.is_string = True


class Solution(object):
    def findWords(self, board, words):
        """
        :type board: List[List[str]]
        :type words: List[str]
        :rtype: List[str]
        """
        visited = [[False for j in xrange(len(board[0]))] for i in xrange(len(board))]
        result = {}
        trie = TrieNode()
        for word in words:
            trie.insert(word)
```

```
    for i in xrange(len(board)):
        for j in xrange(len(board[0])):
            self.findWordsRecu(board, trie, 0, i, j, visited, [], result)

    return result.keys()

def findWordsRecu(self, board, trie, cur, i, j, visited, cur_word, result):
    if not trie or i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or visited[i][j]:
        return

    if board[i][j] not in trie.leaves:
        return

    cur_word.append(board[i][j])
    next_node = trie.leaves[board[i][j]]
    if next_node.is_string:
        result["".join(cur_word)] = True

    visited[i][j] = True
    self.findWordsRecu(board, next_node, cur + 1, i + 1, j, visited, cur_word, result)
    self.findWordsRecu(board, next_node, cur + 1, i - 1, j, visited, cur_word, result)
    self.findWordsRecu(board, next_node, cur + 1, i, j + 1, visited, cur_word, result)
    self.findWordsRecu(board, next_node, cur + 1, i, j - 1, visited, cur_word, result)
    visited[i][j] = False
    cur_word.pop()
```

rectangle-overlap.py

```
# rectangle-overla is not found.
# Time:  O(1)
# Space: O(1)

class Solution(object):
    def isRectangleOverlap(self, rec1, rec2):
        """
        :type rec1: List[int]
        :type rec2: List[int]
        :rtype: bool
        """
        def intersect(p_left, p_right, q_left, q_right):
            return max(p_left, q_left) < min(p_right, q_right)

        return (intersect(rec1[0], rec1[2], rec2[0], rec2[2]) and
                intersect(rec1[1], rec1[3], rec2[1], rec2[3]))
```

fraction-addition-and-subtraction.py

```
# DESC
# Example 2:
# Example 4:
# Example 1:
# Given a string representing an expression of fraction addition and subtraction,
# you need to return the calculation result in string format. The final result sho
# uld 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 3:
# Note:

# NOTE
# Each fraction (input and output) has format ±numerator/denominator. If the first
#  input fraction or the output is positive, then '+' will be omitted.
# The numerator and denominator of the final result are guaranteed to be valid and
#  in the range of 32-bit int.
# The input string only contains '0' to '9', '/', '+' and '-'. So does the output.
# The input only contains valid irreducible fractions, where the numerator and den
# ominator 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 define
# d above.
# The number of given fractions will be in the range [1,10].

# EXAMPLE
# Input:"5/3+1/3"
# Output: "2/1"
# Input:"-1/2+1/2"
# Output: "0/1"
# Input:"-1/2+1/2+1/3"
# Output: "1/3"
# Input:"1/3-1/2"
# Output: "-1/6"

# 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)
```

minimum-swaps-to-make-strings-equal.py

```python
# minimum-swaps-to-make-strings-equal is not found.
# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minimumSwap(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: int
        """
        x1, y1 = 0, 0
        for i in xrange(len(s1)):
            if s1[i] == s2[i]:
                continue
            x1 += int(s1[i] == 'x')
            y1 += int(s1[i] == 'y')
        if x1%2 !=  y1%2:  # impossible
            return -1
        # case1: per xx or yy needs one swap, (x1//2 + y1//2)
        # case2: per xy or yx needs two swaps, (x1%2 + y1%2)
        return (x1//2 + y1//2) + (x1%2 + y1%2)
```