Codes

September 30, 2019

```
# Determine whether or not a given string contains no duplicate characters.


def contains_no_duplicates(string):
  letters = {} #initialize empty string (hash table)
  for letter in string: #for loop over string
    if letter in letters: #check if the letter is in the hashtable
      return False
    letters[letter] = True #assign boolean value
  return True
*****************************************************************
# Determine whether or not one string is a permutation of another.
def is_permutation(str1, str2):
  counter = Counter() #define dictionary with 0 if the key is not in the
     dict
  for letter in str1: #look at the first string
    counter[letter] += 1
  for letter in str2: #look at the second string
    if not letter in counter: #check if the letter is not in the string
      return False
    counter[letter] -= 1
    if counter[letter] == 0:
      del counter[letter] #delete the ket and value from the hashtable
  return len(counter) == 0 #return the length of hashtable
#return 0 if the key is not in the hashtable
class Counter(dict):
  def __missing__(self, key):
    return 0
****************************************************************
# Replace spaces in the middle of a string with "%20" assuming the end of
   the
# string contains twice as many spaces as are in the middle.
def escape_spaces_1(string):
# strip is used to remove all the leading and trailing spaces from a string
```

```
#returns a copy of the string where all occurrences of a substring is
   replaced with another substring
  return string.strip().replace(" ", "%20")

def escape_spaces_2(string):
  # Convert string to list to prepare to be modified
  letters = list(string)
  i = len(letters) - 1
  j = i
  while letters[i] == " ":
    i -= 1
  while j != i:
    # Replace space with %20
    if letters[i] == " ":
      letters[j-2] = "%"
      letters[j-1] = "2"
      letters[j] = "0"
      j -= 2
    # Copy the original character
    else:
      letters[j] = letters[i]
    i -= 1
    j -= 1
  return ''.join(letters) #convert list to string
****************************************************************
# Determine whether the edit distance between two strings is less than 2.
def one_away(str1, str2):
  len_diff = abs(len(str1) - len(str2))
  if len_diff > 1:
    return False
  elif len_diff == 0: #check for repalce
    difference_count = 0
    for i in xrange(len(str1)):
      if str1[i] != str2[i]:
        difference_count += 1
        if difference_count > 1:
          return False
    return True
  else:
    if len(str1) > len(str2): #check for removal/insertion
      longer, shorter = str1, str2
    else:
      longer, shorter = str2, str1
    shift = 0
    for i in xrange(len(shorter)):
```

```
      if shorter[i] != longer[i + shift]:
        if shift or (shorter[i] != longer[i + 1]):
          return False
        shift = 1
    return True
```

****

```
# Compress a string made up of letters by replacing each substring
   containing
# a single type of letter by that letter followed by the count if the result
# is shorter than the original.

def compress(string):
  if len(string) == 0:
    return string
  parts = []
  current_letter = string[0]
  current_count = 1
  for letter in string[1:]:
    if current_letter == letter:
      current_count += 1
    else:
      parts.append(current_letter + str(current_count))
      current_letter = letter
      current_count = 1
  parts.append(current_letter + str(current_count))
  compressed = "".join(parts)
  if len(compressed) < len(string):
    return compressed
  else:
    return string
```

****

```
# Compress a string made up of letters by replacing each substring
   containing
# a single type of letter by that letter followed by the count if the result
# is shorter than the original.

def compress(string):
  if len(string) == 0:
    return string
  parts = [] #initialize empty list
  current_letter = string[0]
  current_count = 1
  for letter in string[1:]: #string started from element 1
```

```
    if current_letter == letter:
      current_count += 1
    else:
      parts.append(current_letter + str(current_count))
      current_letter = letter
      current_count = 1
  parts.append(current_letter + str(current_count))
  compressed = "".join(parts)
  if len(compressed) < len(string):
    return compressed
  else:
    return string
****************************************************************
def rotate_matrix(m):
  n = len(m)
  rotm = [None] * n
  for row in xrange(n): #In Python 3, there is no xrange, but the range
     function
#behaves like xrange in Python 2.
    rotm[row] = [None] * n #initialize a list of size n
  for row in xrange(n):
    for col in xrange(n):
      rotm[n - col - 1][row] = m[row][col]
  return rotm

def rotate_matrix_in_place(m):
  n = len(m)
  for col in xrange(n/2):
    for row in xrange(col, n - col - 1):
      temp1 = m[n - col - 1][row]
      m[n - col - 1][row] = m[row][col]
      temp2 = m[n - row - 1][n - col - 1]
      m[n - row - 1][n - col - 1] = temp1
      temp1 = m[col][n - row - 1]
      m[col][n - row - 1] = temp2
      m[row][col] = temp1

def rotate_matrix(matrix):
    # assume clockwise rotation
    N, M = matrix.shape
    assert(N == M) # must be NxN matrix
    for l in range(math.floor(N / 2)):
        for i in range(l, N - 1 - l):
            # from top row to right column
            right_col_temp = matrix[i, - 1 - l] # save first number from
```

```
            right column
          matrix[i, - 1 - l] = matrix[l, i]

          # from left column to top row
          matrix[l, i] = matrix[N - 1 - i, l]

          # from bottom row to left column
          matrix[N - 1 - i, l] = matrix[-1 - l, N - 1 - i]

          # from right column to bottom row
          matrix[-1 - l, N - 1 - i] = right_col_temp
    return matrix

def rotate_matrix(matrix):
    '''rotates a matrix 90 degrees clockwise'''
    n = len(matrix)
    for layer in range(n // 2): #Division (floor)
        first, last = layer, n - layer - 1
        for i in range(first, last):
            # save top
            top = matrix[layer][i]

            # left -> top
            matrix[layer][i] = matrix[-i - 1][layer]

            # bottom -> left
            matrix[-i - 1][layer] = matrix[-layer - 1][-i - 1]

            # right -> bottom
            matrix[-layer - 1][-i - 1] = matrix[i][- layer - 1]

            # top -> right
            matrix[i][- layer - 1] = top
    return matrix
****************************************************************
# Determine whether or not a given string is a rotation of another string.

def isRotation(s1, s2):
    if len(s1) == len(s2) and len(s1) > 0:
        s1s1 = ''.join([s1, s1])
        if s2 in s1s1: #check if s2 is in s1+s1
            return True
    return False
def is_rotation(s1, s2):
  if len(s1) != len(s2):
```

```
    return False
  return is_substring(s1 + s1, s2)

def is_substring(s1, s2):
  if len(s2) > len(s1):
    return False
  for i in xrange(len(s1) - len(s2) + 1):
    is_substring_here = True
    for j in xrange(len(s2)):
      if s1[i + j] != s2[j]:
        is_substring_here = False
        break
    if is_substring_here:
      return True
  return False
****************************************************************
# Given a matrix, zero out every row and column that contains a zero.
def set_zero(matrix):
    # most optimal solution: O(RxC) time and O(1) space
    R, C = matrix.shape
    # determine if first row and column contain zeros
    first_zero_row = False
    first_zero_col = False
    for c in range(C):
        if matrix[0, c] == 0:
            first_zero_row = True
            break
    for r in range(R):
        if matrix[r, 0] == 0:
            first_zero_col = True
            break
    # check the rest of the matrix for zeros and use first row and col to
    # store this information
    for r in range(1, R):
        for c in range(1, C):
            if matrix[r, c] == 0:
                matrix[0, c] = 0
                matrix[r, 0] = 0
    # look at storage and apply zeros to appropriate rows and columns
    for r in range(1, R):
        if matrix[r, 0] == 0:
            matrix[r, :] = 0
    for c in range(1, C):
        if matrix[0, c] == 0:
            matrix[:, c] = 0
```

```
    # look at first row and first col booleans to zero out first row and col
    if first_zero_row:
        matrix[0, :] = 0
    if first_zero_col:
        matrix[:, 0] = 0
    return matrix
#using dict
def zero_out_row_col(m):
    h = len(m)
    l = len(m[0])
    ipositions = {}
    jpositions = {}

    for i in range (0, h):
        for j in range(0, l):
            if m[i][j] == 0:
                ipositions[i] = 1
                jpositions[j] = 1

    for i in ipositions.keys():
        for j in range (0, l):
            m[i][j] = 0
    for j in jpositions.keys():
        for i in range (0, h):
            m[i][j] = 0
#initialize a 2 dimensional matrix
def rotate_matrix(m):
    l = len(m)
    rotate = [None]*l
    for row in range (0,l):
        rotate[row] = [None]*l
****************************************************************
def remove_duplicates(head):
  node = head
  if node:
    values = {node.data: True}
    while node.next:
      if node.next.data in values:
        node.next = node.next.next
      else:
        values[node.next.data] = True
        node = node.next
  return head

class Node():
```

```
  def __init__(self, data, next):
    self.data = data
    self.next = next
#define a method to change a node to an string
  def __str__(self):
    string = str(self.data)
    if self.next:
      string += ',' + str(self.next)
    return string
def test_remove_duplicates(self):
    head = Node(1,Node(3,Node(3,Node(1,Node(5,None)))))
#without extra buffer
def removeDups_noDS(myList):
        if myList.head == None:
                return

        current = myList.head

        while current.next != None:
                runner = current
                while runner.next != None:
                        if runner.next.value == current.value:
                                runner.next = runner.next.next
                        else:
                                runner = runner.next
                current = current.next
*****************************************************************
# Return the k^{th} to last node in a linked list.

import unittest

def kth_to_last(head, k):
  lead, follow = head, head
  for _ in xrange(k):
    if not lead:
      return None
    lead = lead.next
  while lead:
    lead, follow = lead.next, follow.next
  return follow

class Node():
  def __init__(self, data, next=None):
    self.data, self.next = data, next
```

```
def test_kth_to_last(self):
    head = Node(1,Node(2,Node(3,Node(4,Node(5,Node(6,Node(7)))))))
def find_k_recursevely(self, k):#access the firt element of tuple
        return self.find_k_recurse(k, self.head)[0]

    def find_k_recurse(self, k, x):
        if x is None:
            return (None, 0)
        k_data, cnt = self.find_k_recurse(k, x.next)
        if cnt == k: #how to return tuple
            return (x.data, cnt+1)
        else:
            return (k_data, cnt+1)
****************************************************************
# Delete the given nonterminal node from the containing linked list.
def delete_middle(node):
  next = node.next
  node.data = next.data
  node.next = next.next
****************************************************************
# Partition a linked list so that all of the nodes containing values less
   than
# a pivot value occur before all of the nodes containing values greater than
# or equal to the pivot value.

def partition(head, pivot):
  a_head, a_tail = None, None
  b_head, b_tail = None, None
  node = head
  while node:
    if node.data < pivot:
      if a_head:
        a_tail.next, a_tail = node, node
      else:
        a_head, a_tail = node, node
    else:
      if b_head:
        b_tail.next, b_tail = node, node
      else:
        b_head, b_tail = node, node
    node = node.next
  a_tail.next = b_head
  return a_head
#building the list from left to right
def partition(myList, partition):
```

```
    smallHead = smallTail = bigHead = None
    current = myList.head

    while current != None:
        nextNode = current.next
        current.next = None
        if(current.value < partition):
                if not smallHead:
                        smallHead = smallTail = current
                else:
                        current.next = smallHead
                        smallHead = current
        else:
                if bigHead:
                                current.next = bigHead
                bigHead = current

        current = nextNode

    smallTail.next = bigHead

    # Return head of new list
    return smallHead
****************************************************************
# Sum two numbers that are represented with linked lists with decimal digits
# in reverse order of magnitude.

def sum_lists(num1, num2):
  node1, node2 = num1, num2
  carry = 0
  result_head, result_node = None, None
  while node1 or node2 or carry:
    value = carry
    if node1:
      value += node1.data
      node1 = node1.next
    if node2:
      value += node2.data
      node2 = node2.next
    if result_node:
      result_node.next = Node(value % 10) #the remainder of the value
      result_node = result_node.next
    else:
      result_node = Node(value % 10)
```

```
      result_head = result_node
    carry = value / 10 #the integer devision
  return result_head
#sum lists recursively
def sum_lists_recursevely(h1, h2, carry):
    result = Node(0)
    if h1 is None and h2 is None and carry == 0:
        return None
    val = carry
    if h1 is not None:
        val += h1.data
        h1 = h1.next
    if h2 is not None:
        val += h2.data
        h2 = h2.next
    carry = val // 10
    result.data = val % 10
    result.next = sum_lists_recursevely(h1, h2, carry)
    return result
****************************************************************
#padding a list
def pad_list(ll, n):
    while n != 0:
        add_to_front(ll, 0)
def add_to_front(ll, data):
    # add_to_front_node(ll.head, data)
    ll.head = Node(data, ll.head)
#method length for the node
def length(h):
    i = 0
    while h:
        i += 1
        h = h.next
    return i
****************************************************************
def sum_lists_not_reversed_helper(h1, h2):
    result = Node(0)
    if h1.next is None and h2.next is None:
        val = h1.data + h2.data
        result.data = val % 10
        return (result, val // 10)
    result.next, carry = sum_lists_not_reversed_helper(h1.next, h2.next)
    val = carry + h1.data + h2.data
    result.data = val % 10
    return (result, val // 10)
```

```
def sum_lists_not_reversed_recursively(ll1, ll2, carry):
    h1 = ll1.head
    h2 = ll2.head
    l1 = length(h1)
    l2 = length(h2)
    if l1 > l2:
        pad_list(ll1,l1-l2)
    elif l1 < l2:
        pad_list(ll2,l2-l1)
    result, carry = sum_lists_not_reversed_helper(h1, h2)
    if carry != 0:
        return add_to_front_node(result, carry)
    else:
        return result
****************************************************************
def is_palindrome(head):
  forward, backward = head, copy_reverse(head)
  while forward:
    if forward.data != backward.data:
      return False
    forward, backward = forward.next, backward.next
  return True

def copy_reverse(head):
  prev = None
  node = copy(head)
  while node:
    next = node.next
    node.next = prev
    prev = node
    node = copy(next)
  return prev

def copy(node):
  if node:
    return Node(node.data, node.next)
  else:
    return None
****************************************************************
#implementation of stack
class Stack(object):

    def __init__(self, head=None):
        self.head = head
```

```
    def push(self, data):
        self.head = Node(data, self.head)

    def pop(self):
        if self.head is None:
            return None
        old_head = self.head
        self.head = old_head.next
        return old_head.data

    def peek(self, data):
        return self.head
```
****
```
def is_palindrome_iterative(head): #using stack
    s = Stack()
    x = head
    runner = head
    while runner.next and runner.next.next: #reach the half of stack
        s.push(x)
        x = x.next
        runner = runner.next.next
    if runner.next is not None: # even
        s.push(x)
    x = x.next
    while x:
        if s.pop().data != x.data:
            return False
        x = x.next
    return True
```
****
```
def is_palindrome_recursive(head):
    x = head
    runner = head
    while runner.next and runner.next.next:
        x = x.next
        runner = runner.next.next
    _, result = check_palindrome(head, x)
    return result

def check_palindrome(x1, x2):
    if x2.next is None:
        return (x1, True)
    x2 = x2.next
    x1, _ = check_palindrome(x1, x2)
```

```
    if x1 and x2.data == x1.data:
        return (x1.next, True)
    else:
        return (None, False)
```

```
****************************************************************
#implementation using deque
from LinkedList import LinkedList
from collections import deque

def isPalindrome(myList):
        current = runner = myList.head
        stack = deque()

        while runner != None:
                stack.append(current.value)
                current = current.next
                runner = runner.next
                if runner:
                        runner = runner.next
                else:
                        # In the case the list is even we need to pop the
                           middle element
                        stack.pop()

        while current != None:
                if current.value != stack.pop():
                        return False
                current = current.next

        return True
********************************************************************
# Return an intersecting node if two linked lists intersect.
# Note that the intersection is defined based on reference, not value.
#using hashtable
def intersection(head1, head2):
  nodes = {}
  node = head1
  while node:
    nodes[node] = True
    node = node.next
  node = head2
  while node:
    if node in nodes:
```

```
            return node
        node = node.next
    return None
#efficient algorithm
def isIntersection(listA,listB):

        # Calculate lenght A
        currentA = listA.head
        currentB = listB.head

        print(len(listA),len(listB))

        if len(listA) > len(listB):
                for _ in range(len(listA)-len(listB)):
                        currentA = currentA.next

        if len(listB) > len(listA):
                for _ in range(len(listB)-len(listA)):
                        currentB = currentB.next

        while currentA != None:
                if currentA is currentB: #new way for comparison
                        return currentA
                currentA = currentA.next
                currentB = currentB.next

        return None
****************************************************************

def get_tail_and_cnt(h):
    x = h
    i = 0
    while x.next:
        i += 1
        x = x.next
    return (x, i)

def strip_start(h, i):
    while h and i != 0:
        i -= 1
        h = h.next
    return h

def intersect(h1, h2):
    tail1, i1 = get_tail_and_cnt(h1)
```

```
    tail2, i2 = get_tail_and_cnt(h2)
    if tail1 != tail2:
        return None
    x1 = h1
    x2 = h2
    if i1 > i2:
        h1 = strip_start(h1,i1-i2)
    elif i1 < i2:
        h2 = strip_start(h2,i2-i1)
    while h1 and h2:
        if h1.next == h2.next:
            return h1.next
        h1 = h1.next
        h2 = h2.next
    return None
****************************************************************

# Detect whether or not a linked list contains a cycle.
#using hashtable
def detect_cycle(head):
  nodes = {}
  node = head
  while node:
    if node in nodes:
      return node
    nodes[node] = True
    node = node.next
#Example: creating cycle list


def loopDetection(myList):


    slow = runner = myList.head

    while runner and runner.next:

        slow = slow.next
        runner = runner.next.next

        if slow == runner:

            slow2 = head
            while slow != slow2:
                slow = slow.next
```

```
            slow2 = slow2.next

        return slow

    return None

def test_detect_cycle(self):
    head1 = Node(100,Node(200,Node(300)))
    self.assertEqual(detect_cycle(head1), None)
    node1 = Node(600)
    node2 = Node(700,Node(800,Node(900,node1)))
    node1.next = node2
    head2 = Node(500,node1)
    self.assertEqual(detect_cycle(head2), node1)
  return None
****************************************************************
#implementation of stack
class LinkedList:
        def __init__(self):
                self.head = None

        def __iter__(self):
            current = self.head
            while current:
                yield current
                current = current.next

        def __str__(self):
            values = [str(x) for x in self]
            return ' -> '.join(values)

        def __len__(self):
            result = 0
            node = self.head
            while node:
                result += 1
                node = node.next
            return result

        def addNode(self,node):
                if self.head == None:
                        self.head = node
                else:
                        current = self.head
                        while current.next != None:
```

```
                current = current.next
            current.next = node

    def addElement(self,value):

        if self.head == None:
            self.head = Node(value)
        else:
            current = self.head
            while current.next != None:
                current = current.next
            current.next = Node(value)

    def addRandom(self,min_val,max_val,total):
        for _ in range(total):
            self.addElement(randint(min_val, max_val))

    def addMultiple(self, values):
        for v in values:
            self.addElement(v)

****************************************************************
# Use a single array to implement three stacks.

class ThreeStacks():
  def __init__(self):
    self.array = [None, None, None]
    self.current = [0, 1, 2]

  def push(self, item, stack_number):
    if not stack_number in [0, 1, 2]:
      raise Exception("Bad stack number") #raise exception
    while len(self.array) <= self.current[stack_number]:
      self.array += [None] * len(self.array) #increasing the size of array
    self.array[self.current[stack_number]] = item
    self.current[stack_number] += 3

  def pop(self, stack_number):
    if not stack_number in [0, 1, 2]:
      raise Exception("Bad stack number")
    if self.current[stack_number] > 3:
      self.current[stack_number] -= 3
    item = self.array[self.current[stack_number]]
    self.array[self.current[stack_number]] = None
    return item
```

```
class ThreeStacks(object):

    def __init__(self, size_of_stack):
        self.stack_list = [ None for _ in range(3 * size_of_stack) ] #
            initialize array
        self.p = [0, 100, 200]

    def push(self, stack_index, data):
        if self.p[stack_index] == 100 * stack_index + 100:
            raise Exception("Stack is full.")
        else:
            self.stack_list[self.p[stack_index]] = data
            self.p[stack_index] += 1

    def pop(self, stack_index):
        if self.is_empty(stack_index):
            raise Exception("Stack is empty.")
        else:
            data = self.stack_list[self.p[stack_index]]
            self.stack_list[self.p[stack_index]] = None
            self.p[stack_index] -= 1
            return data

    def peek(self, stack_index):
        if self.is_empty(stack_index):
            raise Exception("Stack is empty.")
        else:
            return self.stack_list[self.p[stack_index]]

    def is_empty(self, stack_index):
        return self.p[stack_index] == 100 * stack_index

****************************************************************

# Implement a stack with a function that returns the current minimum item.

class MinStack():
  def __init__(self):
    self.top, self._min = None, None

  def min(self):
    if not self._min:
      return None
```

```
    return self._min.data

  def push(self, item):
    if self._min and (self._min.data < item):
      self._min = Node(data=self._min.data, next=self._min)
    else:
      self._min = Node(data=item, next=self._min)
    self.top = Node(data=item, next=self.top)

  def pop(self):
    if not self.top:
      return None
    self._min = self._min.next
    item = self.top.data
    self.top = self.top.next
    return item

#implement with two stacks

class Stack(object):

    def __init__(self, head=None):
        self.head = head

    def push(self, data):
        self.head = Node(data, self.head)

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty.")
        else:
            data = self.head.data
            self.head = self.head.next
            return data

    def is_empty(self):
        return self.head is None

    def peek(self):
        if self.is_empty():
            raise Exception("Stack is empty.")
        else:
            return self.head.data

class StackWithMin2(object):
```

```
#define two objects of class stack
    min_stack = Stack()
    data_stack = Stack()

    def push(self, data):
        if self.min_stack.is_empty() or data <= self.min_stack.peek():
            self.min_stack.push(data)
        self.data_stack.push(data)

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty.")
        else:
            data = self.data_stack.pop()
            if self.min_stack.peek() == data:
                self.min_stack.pop()
            return data

    def min(self):
        return self.min_stack.peek()

    def peek(self):
        return self.data_stack.peek()

    def is_empty(self):
        return self.min_stack.is_empty()
*****************************************************************
from . import Stack as s


class StackMin(s.Stack): #inheritance from class Stack
    def __init__(self):
        super().__init__() #use super to reference the parent class instead
            of hard-coding it.
        self.min_stack = s.Stack()

    def push(self, value):
        super().push(value)
        if self.min_stack.is_empty() or value < self.min_stack.peek():
            self.min_stack.push(value)

    def pop(self):
        if not super().is_empty():
            value = super().pop()
            if value <= self.min_stack.peek():
```

```
            self.min_stack.pop()

    def see_min(self):
        return self.min_stack.peek()

****************************************************************
# Implement a class that acts as a single stack made out of multiple stacks
# which each have a set capacity.

class MultiStack():
  def __init__(self, capacity):
    self.capacity = capacity
    self.stacks = []

  def push(self, item):
    if len(self.stacks) and (len(self.stacks[-1]) < self.capacity):
      self.stacks[-1].append(item) #append item to the stack
    else:
      self.stacks.append([item]) #append stack into the set of stacks

  def pop(self):
    while len(self.stacks) and (len(self.stacks[-1]) == 0):
      self.stacks.pop()
    if len(self.stacks) == 0:
      return None
    item = self.stacks[-1].pop()
    if len(self.stacks[-1]) == 0:
#pop from the set of stacks
      self.stacks.pop()
    return item

  def pop_at(self, stack_number):
    if (stack_number < 0) or (len(self.stacks) <= stack_number):
      return None
    if len(self.stacks[stack_number]) == 0:
      return None
    return self.stacks[stack_number].pop()

class SetOfStacks(object):

    def __init__(self, stack_capacity):
        self.stack_capacity = stack_capacity
        self.stacks = [Stack()]

    def push(self, data):
```

```
    if len(self.stacks) == 0 or self.stacks[-1].get_size() == self.
        stack_capacity:
        self.stacks.append(Stack())
    self.stacks[-1].push(data)

def pop(self):
    if len(self.stacks) == 0:
        raise Exception("Stack is empty.")
    data = self.stacks[-1].pop()
    if self.stacks[-1].get_size() == 0:
        self.stacks.pop()
    return data

# leave stacks half-full
def popAtHalfFull(self, index):
    if len(self.stacks) == 0:
        raise Exception("Stack is empty.")
    if index > len(self.stacks) - 1:
        raise IndexError("Index out of range.")
    else:
        data = self.stacks[index].pop()
        if self.stacks[index].get_size() == 0: #size of a list
            del self.stacks[index]
        return data

# full implementation
def popAt(self, index):
    if len(self.stacks) == 0:
        raise Exception("Stack is empty.")
    if index > len(self.stacks) - 1:
        raise IndexError("Index out of range.")
    else:
        data = self.stacks[index].pop()
        if self.stacks[index].get_size() == 0:
            del self.stacks[index]
        else:
            self.reorder_stacks(index)
        return data

def reorder_stacks(self, index):
    while index < len(self.stacks) - 1:
        prev = None
        x = self.stacks[index+1].head
        if x.next:
            while x.next: #delete the node
```

```
                prev = x
                x = x.next
            prev.next = x.next
        else:
            del self.stacks[index+1]
        self.stacks[index].push(x.data)
        index += 1

    def print_stack(self):
        i = len(self.stacks)
        while i > 0:
            stack = self.stacks[i-1]
            i -= 1
            x = stack.head
            while x:
                print(x.data, "-> ", end="")
                x = x.next
        print(None)
```

```
****************************************************************
# Implement a queue using two stacks.

class QueueViaStacks():
  def __init__(self):
    self.in_stack = Stack()
    self.out_stack = Stack()

  def add(self, item):
    self.in_stack.push(item)

  def remove(self):
    if len(self.out_stack) == 0:
      while len(self.in_stack):
        self.out_stack.push(self.in_stack.pop())
    return self.out_stack.pop()
```

```
****************************************************************
# Sort a stack with the smallest on top using only a single temporary stack
def sort(stack):
    s = stack.copy()
    r = Stack()
    while not s.is_empty():
        tmp = s.pop()
        while not r.is_empty() and tmp > r.peek():
            s.push(r.pop())
```

```
        r.push(tmp)
    return r

#find the smallest in each iteratio and push it into stack
def sort_stack(stack):
    temp = s.Stack()
    smallest = None
    stack_size = stack.size
    for i in range(stack_size):
        for j in range(stack_size - i, 0, -1):
            current = stack.pop()
            if smallest is None:
                smallest = current
            else:
                if current < smallest:
                    temp.push(smallest)
                    smallest = current
                else:
                    temp.push(current)
        stack.push(smallest)
        smallest = None
        while not temp.is_empty():
            stack.push(temp.pop())
#implementation in recursive format
def sort_stack(stack):
  temp = Stack()
  previous = stack.pop()
  current = stack.pop()
  while current:
    if current < previous: #put the biggest element on top of temporary
       stack
      temp.push(current)
    else:
      temp.push(previous)
      previous = current
    current = stack.pop()
  is_sorted = True
  current = temp.pop()
  while current: #return the elements in the main stack
    if current > previous:
      is_sorted = False
      stack.push(current)
    else:
      stack.push(previous)
      previous = current
```

```
    current = temp.pop()
  stack.push(previous)
  if is_sorted:
    return stack
  return sort_stack(stack)
```
****

```
# Implement a cat and dog queue for an animal shelter.
class Animal():
  def __init__(self, name):
    self.name = name

  def __str__(self):
    return self.name
#inheritate from class Animal
class Cat(Animal): pass
class Dog(Animal): pass

class AnimalShelter():
  def __init__(self):
    self.cats, self.dogs = [], []

  def enqueue(self, animal):
    #check if the class of animal class is equal to Cat
    if animal.__class__ == Cat: self.cats.append(animal)
    else: self.dogs.append(animal)

  def dequeueAny(self):
    if len(self.cats): return self.dequeueCat()
    return self.dequeueDog()

#dequeue one element from cats
  def dequeueCat(self):
    if len(self.cats) == 0: return None
    cat = self.cats[0]
    self.cats = self.cats[1:]
    return cat
#dequeue one element from dogs
  def dequeueDog(self):
    if len(self.dogs) == 0: return None
    dog = self.dogs[0]
    self.dogs = self.dogs[1:]
    return dog
#add the date the animal is added
import datetime
```

```
class Animal:
	def __init__(self, name, categorie):
		self.name = name
		self.categorie = categorie
		self.added = str(datetime.datetime.now())

	def getAdded(self):
		return self.added
#pop the element only if the date is earlier
def dequeueAny(self):
		if not self.catList and not self.dogList:
			raise Exception('There are no more animals!')
		elif not self.catList and self.dogList:
			return self.dogList.popleft()
		elif self.catList and not self.dogList:
			return self.catList.popleft()
		else:
			if self.dogList[0].getAdded() < self.catList[0].
			   getAdded():
				return self.dogList.popleft()
			else:
				return self.catList.popleft()
#Example
shelter = AnimalShelter()
    shelter.enqueue(Cat("Hanzack"))
    shelter.enqueue(Dog("Pluto"))
    shelter.enqueue(Cat("Garfield"))
    shelter.enqueue(Cat("Tony"))
    shelter.enqueue(Dog("Clifford"))
****************************************************************
#create a node in graph/tree
class Node():
  def __init__(self, data, adjacency_list=None):
    self.data = data
    self.adjacency_list = adjacency_list or []
    self.shortest_path = None

  def add_edge_to(self, node):
    #add a node into array
    self.adjacency_list += [node]

  def __str__(self):
    return self.data

#implement a queue in a graph
```

```
class Queue():
  def __init__(self):
    self.array = []

  def add(self, item):
    self.array.append(item)

  def remove(self):
    if not len(self.array):
      return None
    item = self.array[0]
    #delete element from array
    del self.array[0]
    return item
#another implementation of queue
class Queue:

    q = list()

    def enqueue(self, a):
        self.q.insert(0, a)

    def dequeue(self):
        if len(self.q) == 0:
            return None
        return self.q.pop()

    def __len__(self):
        return len(self.q)
****************************************************************
#implementation of graph
class TetraGraphNode:
    def __init__(self, value, frontier=False, explored=False):
        self.value = value
        self.frontier = frontier
        self.explored = explored


class Graph:
    def __init__(self):
        self.nodes_list = []

    def get_children(self, node):
        return self.nodes_list[node.value - 1][1:]
```

```
    def get_node(self, value):
        return self.nodes_list[value - 1][0]
****************************************************************
# Find a route from the first node to the second node in a directed graph.

def find_route(node1, node2):
  found_path = None
  queue = Queue()
  node = node1
  node.shortest_path = [node]
  all_visited_nodes = [node]
  while node:
    for adjacent in node.adjacency_list:
      if not adjacent.shortest_path:
        adjacent.shortest_path = node.shortest_path + [adjacent]
        if adjacent == node2:
          found_path = node.shortest_path + [adjacent]
          break
        queue.add(adjacent)
        all_visited_nodes.append(adjacent)
    node = queue.remove()
  for visited in all_visited_nodes:
    visited.shortest_path = None
  return found_path

def is_there_a_route(g, v, w):
    if v == w:
        return True
    for x in g:
        x.status = Status.UNVISITED
    q = Queue()
    v.status = Status.VISITING
    q.enqueue(v)
    while len(q) > 0:
        x = q.dequeue()
        for adj_vertex in x.adj:
            if adj_vertex.status == Status.UNVISITED:
                if adj_vertex == w:
                    return True
                adj_vertex.status = Status.VISITING
                q.enqueue(adj_vertex)
        x.status = Status.VISITED
    return False

def path_exists_BFS(graph, start, end):
```

```
    if start is None or end is None:
        return False
    start.frontier = True
    frontier = [start]
    temp_frontier = []
    while len(frontier) > 0:
        for node in frontier:
            if node.value == end.value:
                return True
            for child in graph.get_children(node):
                if not child.frontier and not child.explored:
                    temp_frontier += [child]
                    child.frontier = True
            node.explored = True
        frontier = temp_frontier
        temp_frontier = []
    return False

****************************************************************
#implementation of BST
class BSTNode():
  def __init__(self, data=None, left=None, right=None):
    self.data, self.left, self.right = data, left, right

  def __str__(self):
    string = "(" + str(self.data)
    if self.left: string += str(self.left)
    else: string += "."
    if self.right: string += str(self.right)
    else: string += "."
    return string + ")"
****************************************************************
#creating BST with minimal height
def minimal_height_bst(sorted_array):
  if len(sorted_array) == 0:
    return None
  middle = len(sorted_array) / 2
  left = minimal_height_bst(sorted_array[:middle])
  right = minimal_height_bst(sorted_array[(middle+1):])
  return BSTNode(sorted_array[middle], left, right)
#another implementation
class Node:
    def __init__(self, left, right, val):
        self.left = left
        self.right = right
```

```
        self.val = val


def make_bst(sorted_list):
    if len(sorted_list) < 1:
        return None
    mid_idx = (len(sorted_list) - 1) // 2
    node = Node(make_bst(sorted_list[0:mid_idx]), make_bst(sorted_list[
        mid_idx + 1:]), sorted_list[mid_idx])
    return node
****************************************************************
# Return an array of linked lists containing all elements on each depth
# of a binary tree.

def list_of_depths(binary_tree):
  if not binary_tree:
    return []
  lists = []
  queue = Queue()
  current_depth = -1
  current_tail = None
  node = binary_tree
  node.depth = 0
  while node:
    if node.depth == current_depth:
      current_tail.next = ListNode(node.data)
      current_tail = current_tail.next
    else:
      current_depth = node.depth
      current_tail = ListNode(node.data)
      lists.append(current_tail)
    for child in [node.left, node.right]:
      if child:
        child.depth = node.depth + 1
        queue.add(child)
    node = queue.remove()
  return lists


#aother implementation

from collections import deque

# Definition for a binary tree node.
class TreeNode(object):
```

```
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


def listofDepths(root):

    if not root:
        return []

    lvlList = []
    lvl = 0
    queue = deque()
    queue.append((root,lvl)) #add tuple to the queue


    while queue:
        node, lvl = queue.popleft() #pop tuple from the queue

        if len(lvlList) == lvl:
            lvlList.append([])
        lvlList[lvl].append(node.val)
        lvl +=1

        if node.left:
            queue.append((node.left,lvl))
        if node.right:
            queue.append((node.right,lvl))

    return lvlList

#recursive implementation
def node_to_ll(list_ll, node, depth): # traverse tree using dfs and add
   nodes to LL
    if node is None:
        return
    last_elem = tb.LLElem(node.val, None)
    if depth <= len(list_ll) - 1:
        penultimate_elem = list_ll[depth][1]
        penultimate_elem.next_elem = last_elem
        list_ll[depth][1] = last_elem
    else:
        list_ll += [[last_elem, last_elem]]
    node_to_ll(list_ll, node.left, depth + 1)
```

```
    node_to_ll(list_ll, node.right, depth + 1)
    return


def make_ll(root): # set up recursion. note this is the *tree* root we are
   given
    first_elem = tb.LLElem(root.val, None) # first *list* element
    list_ll = [[first_elem, first_elem]] # list of linked lists data
       structure. keep head and tail for O(n) total runtime
    node_to_ll(list_ll, root.left, 1)
    node_to_ll(list_ll, root.right, 1)
    return list_ll
*****************************************************************
# Tell whether or not a binary tree is balanced.

def is_balanced(binary_tree):
  if not binary_tree:
  #return tuple to up
    return (True, 0)
  (left_balanced, left_depth) = is_balanced(binary_tree.left)
  if not left_balanced:
    return (False, None)
  (right_balanced, right_depth) = is_balanced(binary_tree.right)
  if (not right_balanced) or (abs(left_depth - right_depth) > 1):
    return (False, None)
  depth = max(left_depth, right_depth) + 1
  return (True, depth)

class Node():
  def __init__(self, left=None, right=None):
    self.left, self.right = left, right
#collect all the statements in return
def check_height(root):
    if root is None:
        return [True, 0]
    [left_balanced, left_height] = check_height(root.left)
    [right_balanced, right_height] = check_height(root.right)
    return [left_balanced and right_balanced and abs(left_height -
       right_height) <= 1, 1 + max(left_height, right_height)]


def check_balanced(root):
    return check_height(root)[0]

def dfs(root):
```

```
        if not root:
            return 0

        leftDepth = dfs(root.left)
        if leftDepth == -1: return -1
        rightDepth = dfs(root.right)
        if rightDepth == -1: return -1

        return -1 if abs(leftDepth - rightDepth) > 1 else max(leftDepth,
            rightDepth)+1
****************************************************************
# Validate that a binary tree is a binary search tree.
#define infinite numbers
def validate_tree(binary_tree):
  return validate_tree_node(binary_tree, -float('inf'), float('inf'))

def validate_tree_node(node, left_bound, right_bound):
  if not node:
    return True
  return node.data >= left_bound and node.data <= right_bound and \
        validate_tree_node(node.left, left_bound, node.data) and \
        validate_tree_node(node.right, node.data, right_bound)
#another implementation
def checkBST(root, minVal, maxVal):

            if not root:
                return True

            if minVal != None and root.val <= minVal or maxVal != None and
                root.val >= maxVal:
                return False
            else:
                return checkBST(root.left,minVal,root.val) and checkBST(root.
                    right,root.val,maxVal)

def isValidBST(root):

        if not root:
            return True

        minVal = maxVal = None
        return (checkBST(root.left,minVal,root.val) and checkBST(root.right,
            root.val,maxVal))
****************************************************************
```

```
# Return the successor of a node in a binary search tree.

def successor(node):
  if not node:
    return None
  child = node.right
  if child:
    while child.left:
      child = child.left
  if child:
    return child
  if node.parent and node.parent.data > node.data:
    return node.parent
  return None
class TreeNode(object):
	def __init__(self, x):
		self.val = x
		self.left = None
		self.right = None
		self.parent = None


def minNode(node):

	while root.left:
		root = root.left
	return root.val

def successorNode(node):

	if node.right:
		return minNode(node.right)

	parent = node.parent

	while parent and parent.right == node:
		node = parent
		parent = parent.parent

	return node

****************************************************************
# Find the first common ancestor of two nodes in a tree.
#define dict to discover multiples
def first_common_ancestor(node1, node2):
```

```
  search1, search2 = node1, node2
  ancestors1, ancestors2 = {}, {}
  while search1 or search2:
    if search1:
      if search1 in ancestors2:
        return search1
      ancestors1[search1] = True
      search1 = search1.parent
    if search2:
      if search2 in ancestors1:
        return search2
      ancestors2[search2] = True
      search2 = search2.parent
  return None

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


class Result:
        def __init__(self, x, isAncestor):
                self.node = TreeNode(x)
                self.isAncestor = isAncestor

# Time complextity: O(n)
def first_common_ancestor1(bst, p, q):
    if not in_subtree(bst.root, p) or not in_subtree(bst.root, q):
        return None
    return _first_common_ancestor1(bst.root, p, q)

def _first_common_ancestor1(x, p, q): # pass root in
    if x is None:
        return None
    if x == p or x == q:
        return x
    is_p_on_the_left = in_subtree(x.left, p) # takes 1/2 the number of calls
         each time
    is_q_on_the_left = in_subtree(x.left, q)
    if is_p_on_the_left != is_q_on_the_left:
        return x
    if is_p_on_the_left:
        return _first_common_ancestor1(x.left, p, q)
```

```
    else:
        return _first_common_ancestor1(x.right, p, q)

# Time complextity: O(n)
def in_subtree(x, p):
    if x is None:
        return False
    if x == p:
        return True
    return in_subtree(x.left, p) or in_subtree(x.right, p)
****************************************************************

# Enumerate all inserrtion sequences that could have led to the given BST.

def bst_sequences(bst):
  return bst_sequences_partial([], [bst])

def bst_sequences_partial(partial, subtrees):
  if not len(subtrees):
    return [partial]
  sequences = []
  for index, subtree in enumerate(subtrees):
    next_partial = partial + [subtree.data]
    next_subtrees = subtrees[:index] + subtrees[index+1:]
    if subtree.left:
      next_subtrees.append(subtree.left)
    if subtree.right:
      next_subtrees.append(subtree.right)
    sequences += bst_sequences_partial(next_partial, next_subtrees)
  return sequences

def sequences(x):
    result = list()
    if x is None:
        result.append(LinkedList())
        return result
    prefix = LinkedList()
    prefix.add_last(x.key)
    left = sequences(x.left)
    right = sequences(x.right)
    for l in left:
        for r in right:
            weaved = list()
            weave(l, r, prefix, weaved)
            result += weaved
```

```
    return result

def weave(l1, l2, prefix, ary):
    if l1.is_empty() or l2.is_empty():
        #clone the prefix before add to the array
        result = prefix.clone()
        result.add_all(l1)
        result.add_all(l2)
        ary.append(result)
        return None

    h1 = l1.remove_first()
    prefix.add_last(h1)
    weave(l1, l2, prefix, ary)
    prefix.remove_last()
    l1.add_first(h1)

    h2 = l2.remove_first()
    prefix.add_last(h2)
    weave(l1, l2, prefix, ary)
    prefix.remove_last()
    l2.add_first(h2)
****************************************************************
    def clone(self):
        new = LinkedList()
        x = self.head
        while x:
            new.add_last(x.data)
            x = x.next
        return new

****************************************************************
# Enumerate all inserrtion sequences that could have led to the given BST.
def sequences2(bst):
    all_seqs = list()
    build_seqs(bst.root, list(), list(), all_seqs)
    return all_seqs

def build_seqs(x, building, seq, all_seqs): # building is a queue (append
   left, remove right)
    seq.append(x.key)
    if x.left:
        building.insert(0, x.left)
    if x.right:
        building.insert(0, x.right)
```

```
    if len(building) == 0:
        all_seqs.append(seq)
    for i in range(len(building)):
        x = building.pop()
        build_seqs(x, building.copy(), seq.copy(), all_seqs)
        building.insert(0, x)
****************************************************************
# Determine whether one binary tree is a subtree of another.

def is_subtree(bt1, bt2):
  for node in tree_generator(bt1):
    if equivalent_trees(node, bt2):
      return True
  return False

def equivalent_trees(bt1, bt2):
  if not bt1:
    return not bt2
  if not bt2:
    return False
  if bt1.data != bt2.data:
    return False
  return equivalent_trees(bt1.left, bt2.left) and \
        equivalent_trees(bt1.right, bt2.right)

class Node():
  def __init__(self, data=None, left=None, right=None):
    self.data, self.left, self.right = data, left, right
 #yield func retains enough state to enable function to resume where it is
    left off
def tree_generator(node):
  if not node: return
  yield node
  for child in tree_generator(node.left): yield child
  for child in tree_generator(node.right): yield child

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


def isSameTree(t1,t2):
        if t1 and t2:
```

```
            return t1.val == t2.val and isSameTree(t1.left,t2.left) and
                isSameTree(t1.right,t1.right)
        return t1 is t2

def checkSubtree(t1,t2):

        if not t1:
                return False
        elif t1.val == t2.val and isSameTree(t1,t2):
                return True

        return checkSubtree(t1.left,t2) or checkSubtree(t1.right,t2)
****************************************************************

# Return all downward paths through a tree whose nodes sum to a target value
    .

def paths_with_sum(binary_tree, target_sum):
  partial_paths = ListDict({target_sum: [[]]})
  return paths_with_partial_sum(binary_tree, target_sum, partial_paths)

def paths_with_partial_sum(node, target_sum, partial_paths):
  if not node:
    return []
  next_partial_paths = ListDict({target_sum: [[]]})
  for path_sum, paths in partial_paths.items():
    for path in paths:
      next_partial_paths[path_sum - node.value] += [path + [node.name]]
  paths = next_partial_paths[0]
  for child in [node.left, node.right]:
    paths += paths_with_partial_sum(child, target_sum, next_partial_paths)
  return paths

class Node():
  def __init__(self, name, value, left=None, right=None):
    self.name, self.value, self.left, self.right = name, value, left, right

class ListDict(dict):
  def __missing__(self, key):
    return []

def incrementCount(pathCount, currentSum, increment):
        newCount = pathCount.get(currentSum,0) + increment
        if newCount == 0:
                del pathCount[currentSum] # Remove the key
```

```
        else:
                pathCount[currentSum] = newCount


def countPathHelp(root, targetSum, currentSum, pathCount):

        if not root:
                return 0

        currentSum += root.val
        totalPath = pathCount.get(currentSum - targetSum,0)

        if currentSum == targetSum:
                totalPath +=1

        incrementCount(pathCount,currentSum,1)
        totalPath += countPathHelp(root.left, targetSum, currentSum,
           pathCount)
        totalPath += countPathHelp(root.right, targetSum, currentSum,
           pathCount)
        incrementCount(pathCount,currentSum,-1) # Remove runningSum

        return totalPath


def countPath(root,targetSum):
        return countPathHelp(root, targetSum, 0, dict())
****************************************************************
#another way to check if the key is inside the hash table
try:
        sums_table[running_sum] += 1
    except KeyError:
        sums_table[running_sum] = 1
****************************************************************
# Time: O(n*log(n)) (if tree is balanced, otherwise worst case time
   complexity is O(n^2))
# Space: O(log(n)) (if tree is balanced, otherwise worst case space
   complexity is O(n))
def paths_with_sum_bf(bt, val):
    if bt is None:
        return None
    if bt.root is None:
        return 0
    return count_paths_with_sum_bf(bt.root, val)
```

```
def count_paths_with_sum_bf(x, target_sum):
    if x is None:
        return 0
    sum_root = count_paths_with_sum_from_root_bf(x, 0, target_sum)
    sum_left = count_paths_with_sum_bf(x.left, target_sum)
    sum_right = count_paths_with_sum_bf(x.right, target_sum)
    return sum_left + sum_root + sum_right

def count_paths_with_sum_from_root_bf(x, current_sum, target_sum):
    if x is None:
        return 0
    current_sum += x.data
    if current_sum == target_sum:
        total_paths = 1
    else:
        total_paths = 0
    total_paths += count_paths_with_sum_from_root_bf(x.left, current_sum,
        target_sum)
    total_paths += count_paths_with_sum_from_root_bf(x.right, current_sum,
        target_sum)
    return total_paths
****************************************************************

# Give the number of ways to climb n steps 1, 2, or 3 steps at a time.

def triple_step(n):
  counts = [1, 1, 2]
  if n < 3:
    return counts[n]
  i = 2
  while i < n:
    i += 1
    counts[i % 3] = sum(counts)
  return counts[i % 3]


def countWaysRec(n, memo):
        if n < 0:
                return 0
        elif n == 0:
                return 1
        elif memo[n] != -1:
                return memo[n]
        else:
                memo[n] = countWaysRec(n-1,memo) + countWaysRec(n-2,memo) +
```

```
                  countWaysRec(n-3,memo)
              return memo[n]

def countWays(n):
       memo = [ -1 for _ in range(n+1)]
       return countWaysRec(n, memo)

def triple_step(n, step=0, cnt=0):
    if step >= n:
        return 1 + cnt
    if step + 1 <= n:
        cnt = triple_step(n, step+1, cnt)
    if step + 2 <= n:
        cnt = triple_step(n, step+2, cnt)
    if step + 3 <= n:
        cnt = triple_step(n, step+3, cnt)
    return cnt
***************************************************************
# Guide a robot with "right" and "down" steps from the upper left corner of
    a grid
# to the lower right corner.

def path_through_grid(grid):
  if len(grid) == 0:
    return []
  search = []
  for r, row in enumerate(grid):
    search.append([])
    for c, blocked in enumerate(row):
      if r == 0 and c == 0:
        search[r].append("start")
      elif blocked:
        search[r].append(None)
      elif r > 0 and search[r-1][c]:
        search[r].append("down")
      elif c > 0 and search[r][c-1]:
        search[r].append("right")
      else:
        search[r].append(None)
  path = ["end"]
  r, c = len(grid) - 1, len(grid[0]) - 1
  if not search[r][c]:
    return None
  while c > 0 or r > 0:
    path.append(search[r][c])
```

```
    if search[r][c] == "down":
      r -= 1
    else:
      c -= 1
  path.append("start")
  path.reverse()
  return path

from collections import deque


def getPath(maze):

        maxR, maxC = len(maze), len(maze[0])

        path = dict()
        queue = deque()

        start = (0,0)
        path[start] = None

        queue.append(start)

        while queue:

                r,c = queue.popleft()

                if maze[r][c] == 'E':
                        print('Found')
                        break

                if r-1 > 0 and maze[r-1][c] != '|' and (r-1,c) not in path:
                        queue.append((r-1,c))
                        path[(r-1,c)] = (r,c)
                elif r+1 < maxR and maze[r+1][c] != '|' and (r+1,c) not in path:
                        queue.append((r+1,c))
                        path[(r+1,c)] = (r,c)
                if c-1 > 0 and maze[r][c-1] != '|' and (r,c-1) not in path:
                        queue.append((r,c-1))
                        path[(r,c-1)] = (r,c)
                elif c+1 < maxC and maze[r][c+1] != '|' and (r,c+1) not in path:
                        queue.append((r,c+1))
                        path[(r,c+1)] = (r,c)
```

```
        return path
*****************************************************************
# Time complexity: O(2^(r+c))
# Space complexity: O(r+c)
def find_path(grid):
    if grid is None or len(grid) == 0:
        return None
    path = list()
    if _find_path(grid, len(grid)-1, len(grid[0])-1, path):
        return path
    return None

def _find_path(grid, r, c, path):
    if r < 0 or c < 0 or not grid[r][c]:
        return False
    is_at_origin = (r == 0) and (c == 0)
    if is_at_origin or _find_path(grid, r-1, c, path) or _find_path(grid, r,
         c-1, path):
        path.append((r,c))
        return True
    return False

*****************************************************************

# Find a magic index in a sorted array.

def magic_index_distinct(array):
  if len(array) == 0 or array[0] > 0 or array[-1] < len(array) - 1:
    return None
  return magic_index_distinct_bounds(array, 0, len(array))

def magic_index_distinct_bounds(array, lower, upper):
  if lower == upper:
    return None
  middle = (lower + upper) / 2
  if array[middle] == middle:
    return middle
  elif array[middle] > middle:
    return magic_index_distinct_bounds(array, lower, middle)
  else:
    return magic_index_distinct_bounds(array, middle+1, upper)

def findMagicNumber(A):
```

```
        left, right = 0, len(A)-1

        while left <= right:

                mid = (left + right)//2

                if A[mid] == mid: return mid

                if mid > A[mid]: left = mid+1
                else: right = mid-1

        return -1
*****************************************************************
# FOLLOW UP
# What if the values are not distinct?
def followUp(A):

        def BSearch(left,right):

                if left > right:
                        return -1

                mid = (left+right)//2
                if A[mid] == mid:
                        return mid

                # Check left
                rightIndex = min(mid-1,A[mid])
                left = BSearch(left,rightIndex)
                if left != -1:
                        return left

                # Check right
                leftIndex = max(mid+1,A[mid])
                right = BSearch(leftIndex,right)

                return right

        left, right = 0, len(F)-1
        return BSearch(left, right)
*****************************************************************
# Compute the power set of a set.

def power_set(set0):
  ps = {frozenset()}
```

```
  for element in set0:
    additions = set()
    for subset in ps:
      additions.add(subset.union(element))
    ps = ps.union(additions)
  return ps

# Using combinatorics

def power_set_bin(s):
    all_subsets = list()
    for i in range(1 << len(s)):
        all_subsets.append(convert_int_to_set(s, i))
    return all_subsets

def convert_int_to_set(s, i):
    subset = list()
    j = len(s)-1
    while i != 0:
        if (i & 1) == 1:
            subset.append(s[j])
        j -= 1
        i >>= 1
    return subset

def powerSet(S):

        sets = [[]]

        for element in S:

                total = len(sets)
                for i in range(total):
                        sets.append(sets[i] + [element])

        return sets
****************************************************************
#recursive
def power_set(set):
    if len(set) == 0:
        return [[]]
    subproblem_subsets = power_set(set[0:-1])
    new_subsets = []
    for subset in subproblem_subsets:
        new_subsets += [subset + [set[-1]]]
```

```
    return subproblem_subsets + new_subsets

def power_set(s):
    return _power_set(s, 0)

def _power_set(s, i):
    all_subsets = list()
    if i == len(s):
        all_subsets.append(list())
    else:
        all_subsets = _power_set(s, i+1)
        el = s[i]
        more_subsets = list()
        for subset in all_subsets:
            new_subset = list()
            new_subset += subset
            new_subset.append(el)
            more_subsets.append(new_subset)
        all_subsets += more_subsets
    return all_subsets
****************************************************************
# Multiply two positive integers without *.
def recursive_multiply(a, b):
    if a >= b:
        return _recursive_multiply(a, b, 0)
    else:
        return _recursive_multiply(b, a, 0)

def _recursive_multiply(a, b, i):
    if b == 0:
        return 0
    total = _recursive_multiply(a, b >> 1, i + 1)
    if (b & 1) == 1:
        total += (a << i)
    return total

def recursive_multiply(a, b):
    A = max(a, b)
    B = min(a, b)
    return helper_recursive_multiply(A, B)


def helper_recursive_multiply(a, b):
    if b == 1:
        return a
```

```
    if b == 0:
        return 0
    half_b = int(b >> 1)
    if b > half_b + half_b: # odd b. shifting with truncation eliminates a 1
        return a + helper_recursive_multiply(a << 1, half_b) # odd case
    else:
        return helper_recursive_multiply(a << 1, half_b) # even case
****************************************************************
# Move the blocks on tower1 to tower3.

def towers_of_hanoi(tower1, tower2, tower3, n=None):
  if n is None:
    n = len(tower1.discs)
  if n == 0:
    return
  towers_of_hanoi(tower1, tower3, tower2, n - 1)
  disc = tower1.discs.pop()
  #print("Moving disc {} from {} to {}.".format(disc, tower1, tower3))
  tower3.discs.append(disc)
  towers_of_hanoi(tower2, tower1, tower3, n - 1)

class Tower(object):
  def __init__(self, name, discs=None):
    self.name = name
    if discs:
      self.discs = discs
    else:
      self.discs = []

  def __str__(self):
    return self.name
class Tower(object):

    def __init__(self, index):
        self.stack = list() # add: append(), remove: pop()
        self.index = index

    def add(self, n):
        if len(self.stack) != 0 and self.stack[-1] <= n:
            raise Exception("Cannot place disk in this tower.")
        else:
            self.stack.append(n)

    def move_top_to(self, t):
        t.add(self.stack.pop())
```

```
    def move_disks(self, n, dest, buff):
        if n > 0:
            self.move_disks(n-1, buff, dest)
            self.move_top_to(dest)
            buff.move_disks(n-1, dest, self)

def towers_of_hanoi(n):
    towers = [ Tower(i) for i in range(3) ]
    for i in range(n-1, -1, -1):
        towers[0].add(i)
    towers[0].move_disks(n, towers[2], towers[1])
    for t in towers:
        print(t.stack)
*****************************************************************
# List all permutations of a string that contains no duplicate letters.
def permutations(string):
  return partial_permutations("", string)

def partial_permutations(partial, letters_left):
  if len(letters_left) == 0:
    return [partial]
  permutations = []
  for i, letter in enumerate(letters_left):
    next_letters_left = letters_left[:i] + letters_left[(i+1):]
    permutations += partial_permutations(partial + letter, next_letters_left
        )
  return permutations

def permutations_without_dups2(s):
    if s is None:
        return None
    s = list(s)
    permutations = _permutations_without_dups2(s)
    result = list()
    for p in permutations:
        result.append(''.join(p))
    return result

def _permutations_without_dups2(s):
    if len(s) == 1:
        return [ s ]
    permutations = list()
    for c in s:
        s_cp = s.copy()
```

```
        s_cp.remove(c)
        for i, permutation in enumerate(_permutations_without_dups2(s_cp)):
            permutation_cp = permutation.copy()
            permutation_cp.insert(i, c)
            permutations.append(permutation_cp)

    return permutations
****************************************************************
#iterative
def allPermutations(S):

        permutations = deque()
        permutations.append([])

        for char in S:

                total = len(permutations)
                for k in range(total):

                        element = permutations.popleft()
                        for i in range(len(element),-1,-1):

                                tmp = element[:]
                                tmp.insert(i,char)
                                permutations.append(tmp)

        return permutations
****************************************************************
# List all permutation of the letters in the given string.

def permutations(string):
  return partial_permutations("", sorted(string))

def partial_permutations(partial, letters):
  if len(letters) == 0:
    return [partial]
  permutations = []
  previous_letter = None
  for i, letter in enumerate(letters):
    if letter == previous_letter:
      continue
    next_partial = partial + letter
    next_letters = letters[:i] + letters[(i+1):]
    permutations += partial_permutations(next_partial, next_letters)
    previous_letter = letter
```

```
  return permutations

def perm_dup(s):
    if s is None:
        return None
    if len(s) < 2:
        return s
    s = list(s)
    permutations = list()
    char_freq = char_counts(s)
    results = list()
    _perm_dup(list(), len(s), char_freq, permutations)
    for perm in permutations:
        results.append(''.join(perm))
    return results

def char_counts(s):
    counts = dict()
    for c in s:
        try:
            counts[c] += 1
        except KeyError:
            counts[c] = 1
    return counts

def _perm_dup(s, remaining, char_freq, permutations):
    if remaining == 0:
        permutations.append(s)
        return None
    for c in char_freq.keys():
        if char_freq[c] > 0:
            s_cp = s.copy()
            s_cp.append(c)
            char_freq[c] -= 1
            _perm_dup(s_cp, remaining-1, char_freq, permutations)
            char_freq[c] += 1
#recursive

def permutations_with_dups(string):
    hash_table = {}
    permutations = []
    for character in string:
        if character in hash_table:
            hash_table[character] += 1
        else:
```

```
            hash_table[character] = 1
    helper('', hash_table, permutations)
    return permutations


def helper(string, hash_table, permutations):
    if sum(hash_table.values()) <= 0:
        permutations.append(string)
    else:
        for character in hash_table:
            local_hash_table = hash_table.copy()
            if local_hash_table[character] <= 1:
                local_hash_table.pop(character, None)
            else:
                local_hash_table[character] -= 1
            helper(string + character, local_hash_table, permutations)
****************************************************************

# Fill in the region containing the point with the given color.

def paint_fill(image, x, y, color):
  if x < 0 or y < 0 or len(image) <= y or len(image[y]) <= x:
    return
  old_color = image[y][x]
  if old_color == color:
    return
  paint_fill_color(image, x, y, color, old_color)

def paint_fill_color(image, x, y, new_color, old_color):
  if image[y][x] != old_color:
    return
  image[y][x] = new_color
  if y > 0:
    paint_fill_color(image, x, y - 1, new_color, old_color)
  if y < len(image) - 1:
    paint_fill_color(image, x, y + 1, new_color, old_color)
  if x > 0:
    paint_fill_color(image, x - 1, y, new_color, old_color)
  if x < len(image[y]) - 1:
    paint_fill_color(image, x + 1, y, new_color, old_color)

def paint_fill(screen, p, new_color):
    if screen[p[0]][p[1]] == new_color:
        return None
    _paint_fill(screen, p, screen[p[0]][p[1]], new_color)
```

```
def _paint_fill(screen, p, old_color, new_color):
    if p[0] < 0 or p[0] >= len(screen) or \
       p[1] < 0 or p[1] >= len(screen[0]) or \
       screen[p[0]][p[1]] != old_color:
        return None
    screen[p[0]][p[1]] = new_color
    # surrounding = [[p[0]-1, p[1]-1], [p[0]-1, p[1]], [p[0]-1, p[1]+1],
    # [p[0] , p[1]-1], [p[0] , p[1]], [p[0] , p[1]+1],
    # [p[0]+1, p[1]-1], [p[0]+1, p[1]], [p[0]+1, p[1]+1]]
    surrounding = [(p[0]-1, p[1]), (p[0], p[1]-1), (p[0], p[1]+1), (p[0]+1,
        p[1])]
    for next_p in surrounding:
        _paint_fill(screen, next_p, old_color, new_color)
****************************************************************

# List all valid strings containing n opening and n closing parenthesis.


def parens1(n):
  parens_of_length = [[""]]
  if n == 0:
    return parens_of_length[0]
  for length in xrange(1, n + 1):
    parens_of_length.append([])
    for i in xrange(length):
      for inside in parens_of_length[i]:
        for outside in parens_of_length[length - i - 1]:
          parens_of_length[length].append("(" + inside + ")" + outside)
  return parens_of_length[n]


def parens(n):
    if n <= 0:
        return ['']
    else:
        combinations = []
        helper('', n, n, combinations)
        return combinations


def helper(string, left, right, combinations):
    if left <= 0 and right <= 0:
        combinations.append(string)
    else:
```

```
        if left > 0:
            helper(string + '(', left - 1, right, combinations)
        if right > left and right > 0:
            helper(string + ')', left, right - 1, combinations)

# Using list of chars
# Time: O(n)
def parens(n):
    combinations = list()
    _parens(n, 0, 0, list(), combinations)
    return build_strings(combinations)

def _parens(n, l, r, s, combinations):
    if len(s) == 2*n:
        combinations.append(s)
        return None
    if l < n:
        s_cp = s.copy()
        s_cp.append('(')
        _parens(n, l+1, r, s_cp, combinations)
    if r < l:
        s_cp = s.copy()
        s_cp.append(')')
        _parens(n, l, r+1, s_cp, combinations)

def build_strings(chars_lists):
    str_list = list()
    for chars in chars_lists:
        str_list.append(''.join(chars))
    return str_list
****************************************************************
# Count the number of ways to make change for a given number of cents.

def coins1(cents):
  count = 0
  for c in xrange(cents, -1, -25):
    count += coins1_pnd(c)
  return count

def coins1_pnd(cents):
  count = 0
  for c in xrange(cents, -1, -10):
    count += (c / 5) + 1
  return count
```

```
def make_change_improved(n):
    coins = [ 25, 10, 5, 1 ]
    ways_cache = [ [ None for _ in range(len(coins)) ] for _ in range(n+1) ]
    return _make_change_improved(n, coins, 0, ways_cache)

def _make_change_improved(n, coins, i, ways_cache):
    if ways_cache[n][i] is not None:
        return ways_cache[n][i]
    if i >= len(coins) - 1:
        return 1
    ways = 0
    possibilities = n // coins[i]
    for j in range(possibilities + 1):
        ways += _make_change_improved(n - j * coins[i], coins, i + 1,
            ways_cache)
    ways_cache[n][i] = ways
    return ways
# unoptimized and no memoization
def coin_representations(n, denom_index=0, denoms=(1, 5, 10, 25)):
    if n == 0: # base case: we've found a combination of coins that divides
       evenly into amount of change
        return 1
    if denom_index >= len(denoms): # check for case where we don't ever pick
        any coins
        return 0
    coin = denoms[denom_index] # current coin we pick. we only pick one type
        of coin at each recursion level
    branches = n // coin # max number of times we can pick this coin.
    representations = 0
    while branches >= 0: # subtract coin value from n for each recursive
       call. there is one case where we pick 0 coins!
        representations += coin_representations(n - branches*coin,
            denom_index + 1, denoms)
        branches -= 1
    return representations

****************************************************************
# Find all arrangements of eight queens on a chess board that cannot attack
# each other.
def n_queens(n):
    ways = list()
    _n_queens(n, 0, list(), ways)
    return ways

def _n_queens(n, c, queens, ways):
```

```
    if c == n:
        ways.append(queens)
        return None
    for r in range(n):
        position = [ r, c ]
        if meet_requirements(position, queens):
            queens_cp = queens.copy()
            queens_cp.append(position)
            _n_queens(n, c+1, queens_cp, ways)

def meet_requirements(position, queens):
    for queen in queens:
        if queen[0] == position[0]:
            return False
        if queen[1] == position[1]:
            return False
        if (abs(queen[0] - position[0]) == abs(queen[1] - position[1])):
            return False
    return True
****************************************************************
# Stack boxes as high as possible.
from random import random
from math import floor
from pprint import pprint

def get_max_height(boxes):
    boxes = sorted(boxes, key=lambda x: x['height'], reverse=True)
    max_height = 0
    max_heights = [ None for _ in range(len(boxes)) ]
    for i in range(len(boxes)):
        height = _get_max_height(boxes, i, max_heights)
        max_height = max(height, max_height)
    return max_height

def _get_max_height(boxes, i, max_heights):
    if max_heights[i] is not None:
        return max_heights[i]
    bottom = boxes[i]
    max_height = 0
    for j in range(i+1, len(boxes)):
        if meet_requirements(bottom, boxes[j]):
            height = _get_max_height(boxes, j, max_heights)
            max_height = max(height, max_height)
    max_height += bottom['height']
    max_heights[i] = max_height
```

```
    return max_height

def get_max_height2(boxes):
    boxes = sorted(boxes, key=lambda x: x['height'], reverse=True)
    max_heights = [ 0 for _ in range(len(boxes)) ]
    return _get_max_height2(boxes, None, 0, max_heights)

def _get_max_height2(boxes, bottom, offset, max_heights):
    if offset >= len(boxes):
        return 0
    new_bottom = boxes[offset]
    height_with_bottom = 0
    if bottom is None or meet_requirements(bottom, new_bottom):
        if max_heights[offset] == 0:
            max_heights[offset] = _get_max_height2(boxes, new_bottom, offset
                +1, max_heights)
            max_heights[offset] += new_bottom['height']
        height_with_bottom = max_heights[offset]
    height_without_bottom = _get_max_height2(boxes, bottom, offset+1,
        max_heights)
    return max(height_with_bottom, height_without_bottom)

def meet_requirements(bottom, top):
    return bottom['height'] > top['height'] and \
           bottom['width'] > top['width'] and \
           bottom['depth'] > top['depth']

def create_box(h,w,d):
    return { 'height': h, 'width': w, 'depth': d }

if __name__ == "__main__":
    boxes = [ create_box(1+floor(200*random()),
                         1+floor(200*random()),
                         1+floor(200*random())
                        ) for _ in range(0,10) ]
****************************************************************
# Return the number of ways that an expression can be parenthesized and
# achieve a given truth value

def count_eval(expr, value, memo=None):
  if len(expr) % 2 == 0:
    return Exception("Malformed expression.")
  if len(expr) == 1:
    return int((expr == "0") ^ value)
  if memo is None:
```

```
    memo = {}
  elif expr in memo:
    counts = memo[expr]
    return counts[int(value)]
  true_count = 0
  for opix in xrange(1, len(expr) - 1, 2):
    left, op, right = expr[:opix], expr[opix], expr[(opix+1):]
    if op == '&':
      true_count += count_eval(left,True, memo) * count_eval(right,True,
          memo)
    elif op == '|':
      true_count += count_eval(left,True, memo) * count_eval(right,True,
          memo)
      true_count += count_eval(left,False,memo) * count_eval(right,True,
          memo)
      true_count += count_eval(left,True, memo) * count_eval(right,False,
          memo)
    elif op == '^':
      true_count += count_eval(left,True, memo) * count_eval(right,False,
          memo)
      true_count += count_eval(left,False,memo) * count_eval(right,True,
          memo)
    else:
      return Exception('Unknown operation.')
  total_count = catalan_number((len(expr) - 1) / 2)
  false_count = total_count - true_count
  counts = (false_count, true_count)
  memo[expr] = counts
  return counts[int(value)]

def catalan_number(n):
  number = 1
  for i in xrange(n + 1, 2*n + 1):
    number *= i
  for i in xrange(1, n + 2):
    number /= i
  return number
****************************************************************
# Merge array b into array a given that array a contains len(b) extra space
   at
# the end.

def sorted_merge(a, b):
  bix = len(b) - 1
  aix = len(a) - len(b) - 1
```

```
  while aix >= 0 and bix >= 0:
    if a[aix] > b[bix]:
      a[aix + bix + 1] = a[aix]
      aix -= 1
    else:
      a[aix + bix + 1] = b[bix]
      bix -= 1
  while bix >= 0:
    a[bix] = b[bix]
    bix -= 1
```
****

```
# Group string anagrams together.

def group_anagrams(strings):
  pairs = [(s, sorted(s)) for s in strings]
  pairs.sort(key=lambda p: p[1])
  return [p[0] for p in pairs]

# Time: O(l * n*log(n) + l), l = length of array, n = chars in array
def group_anagrams(a):
    map_anagrams = dict()
    for s in a:
        anagram = ''.join(sorted(s))
        if anagram in map_anagrams.keys():
            map_anagrams[anagram].append(s)
        else:
            map_anagrams[anagram] = [s]
    result = list()
    for v in map_anagrams.values():
        result += v
    return result

if __name__ == "__main__":
    a = ['aaa', 'bbb', 'ccc', 'abc', 'bca', 'cba']
    print(group_anagrams(a))
```
****

```
# Search for an item in a rotated array.
def search(a, el):
    return _search(a, el, 0, len(a)-1)

def _search(a, el, lo, hi):
    if lo > hi:
        return None
    if el == a[mid]:
        return mid
```

```
    mid = (lo + hi) // 2
    if a[lo] < a[mid]: # left side is normally ordered
        if el < a[mid] and el > a[lo]: # go left
            return _search(a, el, lo, mid - 1)
        else: # go right
            return _search(a, el, mid + 1, hi)
    elif a[mid] < a[lo]: # right side is normally ordered
        if el > a[mid] and el < a[hi]: # go right
            return _search(a, el, mid + 1, hi)
        else: # go left
            return _search(a, el, lo, mid - 1)
    elif a[mid] == a[lo]: # all elements on the left are repeated
        if a[mid] != a[hi]: # go right
            return _search(a, el, mid + 1, hi)
        else: # must check both sides
            result = _search(a, el, lo, mid - 1) # try left
            if result is None:
                return _search(a, el, mid + 1, hi) # try right
            else:
                return result
    return None
****************************************************************
# Find an item in a sorted list-like object without knowing its length.

def search_listy(listy, item, leftix=0, rightix=None):
  if rightix is None:
    rightix = 4
    right = listy[rightix]
    while right < item and right != -1:
      rightix *= 2
      right = listy[rightix]
    if right == item:
      return rightix
  if leftix == rightix:
    return None
  middleix = (leftix + rightix) / 2
  middle = listy[middleix]
  if middle == item:
    return middleix
  if middle == -1 or middle > item:
    return search_listy(listy, item, leftix, middleix)
  else:
    return search_listy(listy, item, middleix+1, rightix)

class Listy(object):
```

```
  def __init__(self, array):
    self.array = array

  def __getitem__(self, ix):
    if ix < len(self.array):
      return self.array[ix]
    else:
      return -1

# Use a list, but don't use the len() method.
# Time: O(log^2(n))
def binary_search(listy, x):
    if listy[0] == -1:
        return None
    lo = 0
    hi = find_size(listy)
    while lo <= hi:
        mid = (lo + hi) // 2
        if x < listy[mid]:
            hi = mid-1
        elif x > listy[mid]:
            lo = mid+1
        else:
            return mid
    return None

def find_size(listy, i=0, delta=1):
    if listy[i+delta] == -1 and listy[i+delta-1] != -1:
        return i+delta-1
    elif listy[i+delta] != -1:
        return find_size(listy, i, 2*delta)
    else:
        return find_size(listy, i+delta//2, 1)
****************************************************************
# Search a sorted sparse array of positive integers.

def find_string(a, x):
    if a is None or x is None or x == '':
        return None
    return _find_string(a, x, 0, len(a)-1)

def get_mid(a, lo, hi):
    mid = lo + (hi - lo) // 2
    if a[mid] != '':
        return mid
```

```
    mid_l = mid-1
    mid_r = mid+1
    while a[mid_l] == '' and a[mid_r] == '':
        if mid_l < lo or mid_r > hi:
            return None
        mid_l -= 1
        mid_r += 1
    if a[mid_l] == '':
        return mid_r
    else:
        return mid_l

def _find_string(a, x, lo, hi):
    if lo <= hi:
        mid = get_mid(a, lo, hi)
        if mid is None:
            return None
        if x < a[mid]:
            return _find_string(a, x, lo, mid-1)
        elif x > a[mid]:
            return _find_string(a, x, mid+1, hi)
        else:
            return mid
    return None
*****************************************************************
# Given an input file with four billion non-negative integers, provide an
   algorithm to generate
# an integer that is not contained in the file. Assume you have 1 GB of
   memory available for this task.


# 2^31 different non-negative integers
# 1 GB = 2^33 bits
# create list of integers, each integer correspond to 64 numbers
# the list needs 2^(31 - 6) = 2^25 bytes to hold the information

def gen_integer(fname):
    bit_vector = [0] * (1 << 25)
    mark_bits(fname, bit_vector)
    return find_first_zero(bit_vector)

def find_first_zero(bit_vector):
    for i, integer in enumerate(bit_vector):
        for j in range(64):
            if ((integer >> j) & 1 == 0): # integer & (1 << i) == 0
```

```
            return 64 * i + j

def mark_bits(fname, bit_vector):
    with open(fname, 'r') as numbers:
        for n in numbers:
            try:
                mark_bit(int(n), bit_vector)
            except ValueError:
                pass

def mark_bit(number, bit_vector):
    list_position = number // 64
    bit_offset = number % 64
    bit_vector[list_position] |= (1 << bit_offset)
****************************************************************
# FOLLOW UP
# What if you have only 10 MB of memory? Assume that all values are distinct
    an we now have no more than
# one billion non-negative integers.

# FOLLOW UP

# (1)
# scan dataset once counting numbers within a range of size range_size
# store counts in range_count
# full ranges will have a count == range_size

# (2)
# scan dataset again looking for numbers, whose count != range_size and map
   them in a bit_vector

# (3)
# 10 MB of memory = 2^23 bits
# 2^30 distinct integers
# range_size < 2^23
# range_count < 2^23
# we can pick a range_size of 2^10 and a range_count of 2^13 elements

def gen_integer(fname):
    range_size = (1 << 10)
    range_counts = [0] * (1 << 13)
    count_ranges(fname, range_size, range_counts)
    start_of_range = find_range(range_counts, range_size)
    return find_integer(fname, start_of_range)
```

```
def find_integer(fname, start_of_range, range_size):
    bit_vector = mark_bits(fname, start_of_range, range_size)
    first_zero = find_first_zero(bit_vector)
    return start_of_range + first_zero

def find_first_zero(bit_vector):
    for i, integer in enumerate(bit_vector):
        for j in range(64):
            if (integer & (1 << j) == 0):
                return 64 * i + j

def mark_bits(fname, start_of_range, range_size):
    int_size = 64 # len(bin(sys.maxsize + 1)) - 2
    bit_vector = [0] * (range_size//int_size)
    with open(fname, 'r') as numbers:
        for n in numbers:
            if n >= start_of_range and n < start_of_range + range_size:
                mark_bit(n - start_of_range, bit_vector)
    return bit_vector

def mark_bit(number, bit_vector):
    bit_vector[number//64] |= (1 << (number % 64))

def find_range(range_counts, range_size):
    for i, count in enumerate(range_counts):
        if count != range_size:
            return i * range_size

def count_ranges(fname, range_size, range_counts):
    with open(fname, 'r') as numbers:
        for n in numbers:
            range_counts[n//range_size] += 1
****************************************************************
# Find the duplicates in an array that contains numbers between 0 and 32000.
# 32000 <= 2^15
# 4kb = 2^15 bits = 32768 bits

def find_duplicates(a):
    bit_vector = [0] * 500 # 500 numbers of 64 bits each (32000 bits)
    for n in a:
        vector_position = (n-1) // 64
        offset = (n-1) % 64
        if (bit_vector[vector_position] & (1 << offset)) == 0:
            bit_vector[vector_position] |= (1 << offset)
        else:
```

```
            print(n)



class BitSet(object):

    def __init__(self, size):
        self.bit_vector = [0] * (size // 64 + 1)

    def get(self, n):
        vector_position = n // 64
        offset = n % 64
        return self.bit_vector[vector_position] & (1 << offset) != 0

    def set(self, n):
        vector_position = n // 64
        offset = n % 64
        self.bit_vector[vector_position] |= (1 << offset)

def print_duplicates(a):
    bs = BitSet(32000)
    for n in a:
        if bs.get(n-1):
            print(n)
        else:
            bs.set(n-1)


if __name__ == "__main__":
    a = list()
    for i in range(32000):
        a.append(i)
    a[23322] = 22222
    a[222] = 7777
    print_duplicates(a)
****************************************************************
# Time: O(M*log(N))
def find_element(m, el):
    for r, row in enumerate(m):
        c = binary_search(row, el, 0, len(row)-1)
        if c is not None:
            return r, c

def binary_search(a, el, lo, hi):
```

```
    if lo <= hi:
        mid = (lo + hi) // 2
        if el < a[mid]:
            return binary_search(a, el, lo, mid-1)
        elif el > a[mid]:
            return binary_search(a, el, mid+1, hi)
        else:
            return mid

# Time: O(M+N)
def find_element(m, el):
    r = 0
    c = len(m) - 1
    while r < len(m) and c > 0:
        if el < m[r][c]:
            c -= 1
        elif el > m[r][c]:
            r += 1
        else:
            return r, c

# Time: O(log(N+M))
def find_element(m, el):
    return _find_element(m, [0,0], [len(m)-1, len(m[0])-1], el)

def _find_element(m, origin, dest, x):
    if not inbounds(m, origin) or not inbounds(m, dest):
        return None
    if m[origin[0]][origin[1]] == x:
        return tuple(origin)
    elif not is_before(origin, dest):
        return None
    start = origin.copy()
    diagonal_distance = min(dest[0] - origin[0], dest[1] - origin[1])
    end = [start[0] + diagonal_distance, start[1] + diagonal_distance]
    p = [0, 0]
    while is_before(start, end):
        p = set_to_average(start, end)
        if x > m[p[0]][p[1]]:
            start[0] = p[0] + 1
            start[1] = p[1] + 1
        else:
            end[0] = p[0] - 1
            end[1] = p[1] - 1
    return partition_and_search(m, origin, dest, start, x)
```

```
def partition_and_search(m, origin, dest, pivot, x):
    lower_left_origin = [pivot[0], origin[1]]
    lower_left_dest = [dest[0], pivot[1]-1]
    upper_right_origin = [origin[0], pivot[1]]
    upper_right_dest = [pivot[0]-1, dest[1]]
    lower_left = _find_element(m, lower_left_origin, lower_left_dest, x)
    if lower_left is None:
        return _find_element(m, upper_right_origin, upper_right_dest, x)
    return lower_left

def set_to_average(origin, dest):
    return [(origin[0] + dest[0]) // 2, (origin[1] + dest[1]) // 2]

def is_before(origin, dest):
    return origin[0] <= dest[0] and origin[1] <= dest[1]

def inbounds(m, coordinate):
    r, c = coordinate
    return not (r < 0 or c < 0 or r >= len(m) or c >= len(m[0]))

*************************************************************
# Record the number of elements lower than any given elements in a stream.
# Time complexity:
    # Track: O(log(n))
    # Get Rank: O(n)
# Space complexity: O(n)
class Node_v1(object):

    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BST_v1(object):

    def __init__(self, root=None):
        self.root = root

    def track(self, data):
        if self.root is None:
            self.root = Node_v1(data)
        else:
            self.root = self._track(self.root, data)
```

```
    def _track(self, x, data):
        if x is None:
            return Node_v1(data)
        if data <= x.data:
            x.left = self._track(x.left, data)
        else:
            x.right = self._track(x.right, data)
        return x

    def get_rank_of_number(self, data):
        if self.root is None:
            return None
        return self._get_rank_of_number(self.root, data)

    def _get_rank_of_number(self, x, data, found=False):
        if x is None:
            return 0
        if data <= x.data and not found:
            if data == x.data:
                found = True
            return self._get_rank_of_number(x.left, data, found)
        elif data <= x.data:
            return 1 + self._get_rank_of_number(x.left, data, found)
        else:
            return 1 + self._get_rank_of_number(x.left, data, found) \
                    + self._get_rank_of_number(x.right, data, found)
# Time complexity:
    # Track: O(log(n))
    # Get Rank: O(log(n))
# Space complexity: O(n)

class Node_v2(object):

    def __init__(self, data, left_count=0):
        self.data = data
        self.left_count = left_count
        self.left = None
        self.right = None

class BST_v2(object):

    def __init__(self, root=None):
        self.root = root

    def track(self, data):
```

```
        if self.root is None:
            self.root = Node_v2(data)
        else:
            self.root = self._track(self.root, data)

    def _track(self, x, data):
        if x is None:
            return Node_v2(data)
        if data <= x.data:
            x.left = self._track(x.left, data)
            x.left_count += 1
        else:
            x.right = self._track(x.right, data)
        return x

    def get_rank_of_number(self, data):
        if self.root is None:
            return None
        return self._get_rank_of_number(self.root, data)

    def _get_rank_of_number(self, x, data):
        if x is None:
            return None
        if data == x.data:
            return x.left_count
        elif data < x.data:
            return self._get_rank_of_number(x.left, data)
        else:
            right_count = self._get_rank_of_number(x.right, data)
            if right_count is None:
                return None
            return x.left_count + 1 + right_count


****************************************************************
# Reorder an array into peaks and valleys.
# Time complexity: O(n*log(n))
# Space complexity: O(1)
def peaks_and_valleys(a):
    a.sort()
    for i in range(1, len(a), 2):
        a[i-1], a[i] = a[i], a[i-1]

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

```
def peaks_and_valleys(a):
    for i in range(0, len(a), 2):
        _max = max_index(a, i-1, i, i+1)
        if _max != i:
            a[i], a[_max] = a[_max], a[i]

def max_index(_list, a, b, c):
    a_val = within_boundary(_list, a)
    b_val = within_boundary(_list, b)
    c_val = within_boundary(_list, c)
    _max = max(a_val, max(b_val, c_val))
    if _max == a_val:
        return a
    elif _max == b_val:
        return b
    else:
        return c

def within_boundary(_list, a):
    if a >= 0 and a < len(_list):
        return _list[a]
    else:
        return -sys.maxsize-1
****************************************************************
****************************************************************
****************************************************************
****************************************************************
****************************************************************
****************************************************************
****************************************************************
```