Valid Binary Search Tree

```
class Node(object):
  def __init__(self, val, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right

class Solution(object):
  def _isValidBSTHelper(self, n, lows , high):
    if not n:
      return True
    val = n.val
    if ((val > low and val < high) and
        self._isValidBSTHelper(n.left, low, n.val) and
        self._isValidBSTHelper(n.right, n.val, high)):
        return True
    return False

  def isValidBST(self, n):
    return self._isValidBSTHelper(n, float('-inf'),
float('inf'))


#   5
#  / \
# 4   7
node = Node(5)
node.left = Node(4)
node.right = Node(7)
print(Solution().isValidBST(node))

#   5
#  / \
# 4   7
#    /
#   2
node = Node(5)
node.left = Node(4)
node.right = Node(7)
node.right.left = Node(2)
print(Solution().isValidBST(node))
# False
```

Ransom Note

```
from collections import defaultdict

class Solution(object):
  def canSpell(self, magazine, note):
    letters = defaultdict(int)
    for c in magazine:
      letters[c] += 1

    for c in note:
      if letters[c] <= 0:
        return False
      letters[c] -= 1

    return True

print(Solution().canSpell(['a', 'b', 'c', 'd', 'e', 'f'],
'bed'))
# True

print(Solution().canSpell(['a', 'b', 'c', 'd', 'e', 'f'],
'cat'))
# False
```

Add two numbers as a linked list

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


class Solution:
  def addTwoNumbers(self, l1, l2):
    return self.addTwoNumbersRecursive(l1, l2, 0)
    # return self.addTwoNumbersIterative(l1, l2)

  def addTwoNumbersRecursive(self, l1, l2, c):
    val = l1.val + l2.val + c
    c = val // 10
    ret = Node(val % 10)

    if l1.next != None or l2.next != None:
      if not l1.next:
        l1.next = Node(0)
      if not l2.next:
        l2.next = Node(0)
      ret.next =
self.addTwoNumbersRecursive(l1.next, l2.next, c)
    elif c:
      ret.next = Node(c)
    return ret

  def addTwoNumbersIterative(self, l1, l2):
    a = l1
    b = l2
    c = 0
    ret = current = None

    while a or b:
      val = a.val + b.val + c
      c = val // 10
      if not current:
        ret = current = Node(val % 10)
      else:
        current.next = Node(val % 10)
        current = current.next

      if a.next or b.next:
        if not a.next:
          a.next = Node(0)
        if not b.next:
          b.next = Node(0)
      elif c:
        current.next = Node(c)
      a = a.next
      b = b.next
    return ret

l1 = Node(2)
l1.next = Node(4)
l1.next.next = Node(3)

l2 = Node(5)
l2.next = Node(6)
l2.next.next = Node(4)

answer = Solution().addTwoNumbers(l1, l2)
while answer:
  print(answer.val, end=' ')
  answer = answer.next
# 7 0 8
```

Two Sum

```
class Solution(object):
 def twoSum(self, nums, target):
  for i1, a in enumerate(nums):
   for i2, b in enumerate(nums):
    if a == b:
     continue
    if a + b == target:
     return [i1, i2]
  return []

 def twoSumB(self, nums, target):
  values = {}
  for i, num in enumerate(nums):
   diff = target - num
   if diff in values:
    return [i, values[diff]]
   values[num] = i
  return []


print(Solution().twoSumB([2, 7, 11, 15], 18))
```

First and Last Indices of an Element in a Sorted Array

```
class Solution:
 def getRange(self, arr, target):
  first = self.binarySearchIterative(arr, 0, len(arr) -
1, target, True)
  last = self.binarySearchIterative(arr, 0, len(arr) -
1, target, False)
  return [first, last]

 def binarySearch(self, arr, low, high, target,
findFirst):
  if high < low:
   return -1
  mid = low + (high - low) // 2
  if findFirst:
   if (mid == 0 or target > arr[mid - 1]) and
arr[mid] == target:
    return mid
   if target > arr[mid]:
    return self.binarySearch(arr, mid + 1, high,
target, findFirst)
   else:
    return self.binarySearch(arr, low, mid - 1,
target, findFirst)
  else:
   if (mid == len(arr)-1 or target < arr[mid + 1])
and arr[mid] == target:
    return mid
   elif target < arr[mid]:
    return self.binarySearch(arr, low, mid - 1,
target, findFirst)
   else:
    return self.binarySearch(arr, mid + 1, high,
target, findFirst)

 def binarySearchIterative(self, arr, low, high,
target, findFirst):
  while True:
   if high < low:
    return -1
   mid = low + (high - low) // 2
   if findFirst:
```

```
        if (mid == 0 or target > arr[mid - 1]) and arr[mid] == target:
          return mid
        if target > arr[mid]:
          low = mid + 1
        else:
          high = mid - 1
      else:
        if (mid == len(arr)-1 or target < arr[mid + 1]) and arr[mid] == target:
          return mid
        elif target < arr[mid]:
          high = mid - 1
        else:
          low = mid + 1

arr = [1, 3, 3, 5, 7, 8, 9, 9, 9, 15]
x = 9
print(Solution().getRange(arr, 9))
# [6, 8]
```

Mark As Complete

Permutations

```
class Solution(object):
  def _permuteHelper(self, nums, start=0):
    if start == len(nums) - 1:
      return [nums[:]]

    result = []
    for i in range(start, len(nums)):
      nums[start], nums[i] = nums[i], nums[start]
      result += self._permuteHelper(nums, start + 1)
      nums[start], nums[i] = nums[i], nums[start]
    return result

  def permute(self, nums):
    return self._permuteHelper(nums)

  def permute2(self, nums, values=[]):
    if not nums:
      return [values]
    result = []
    for i in range(len(nums)):
      result += self.permute2(nums[:i] + nums[i+1:], values + [nums[i]])
    return result

  def permute2Iterative(self, nums):
    results = []
    stack = [(nums, [])]
    while len(stack):
      nums, values = stack.pop()
      if not nums:
        results += [values]
      for i in range(len(nums)):
        stack.append((nums[:i]+nums[i+1:], values+[nums[i]]))
    return results


print(Solution().permute([1, 2, 3]))
# [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1], [3, 1, 2]]

print(Solution().permute2([1, 2, 3]))
# [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1], [3, 1, 2]]
```

```
print(Solution().permute2Iterative([1, 2, 3]))
# [[3, 2, 1], [3, 1, 2], [2, 3, 1], [2, 1, 3], [1, 3, 2], [1,
2, 3]]
```

Sorting a list with 3 unique numbers

```
def sortNums(nums):
  counts = {}
  for n in nums:
    counts[n] = counts.get(n, 0) + 1
  return ([1] * counts.get(1, 0) +
          [2] * counts.get(2, 0) +
          [3] * counts.get(3, 0))
```

```
def sortNums2(nums):
  one_index = 0
  three_index = len(nums) - 1
  index = 0
  while index <= three_index:
    if nums[index] == 1:
      nums[index], nums[one_index] =
nums[one_index], nums[index]
      one_index += 1
      index += 1
    elif nums[index] == 2:
      index += 1
    elif nums[index] == 3:
      nums[index], nums[three_index] =
nums[three_index], nums[index]
      three_index -= 1
  return nums
```

```
print(sortNums2([3, 3, 2, 1, 3, 2, 1]))
# [1, 1, 2, 2, 3, 3, 3]
```

Queue Reconstruction By Height

```
class Solution:
  def reconstructQueue(self, input):
    input.sort(key=lambda x:
            (-x[0], x[1])
            )
    res = []
    for person in input:
      res.insert(person[1], person)
    return res
```

```
input = [[7, 0], [4, 4], [7, 1], [5, 0], [6, 1], [5, 2]]
print(Solution().reconstructQueue(input))
# [[5,0], [7, 0], [5, 2], [6, 1], [4, 4], [7, 1]]
```

Find the non-duplicate number

```
class Solution(object):
  def singleNumber(self, nums):
    occurrence = {}

    for n in nums:
      occurrence[n] = occurrence.get(n, 0) + 1

    for key, value in occurrence.items():
      if value == 1:
        return key
```

```
def singleNumber2(self, nums):
  unique = 0
  for n in nums:
    unique ^= n
  return unique

print(Solution().singleNumber2([4, 3, 2, 4, 1, 3, 2]))
```

Reverse A Linkedlist

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

  def __repr__(self):
    res = str(self.val)
    if self.next:
      res += str(self.next)
    return res

class Solution(object):
  def reverse(self, node):
    curr = node
    prev = None

    while curr != None:
      tmp = curr.next
      curr.next = prev
      prev = curr
      curr = tmp

    return prev

node = Node(1, Node(2, Node(3, Node(4, Node(5)))))

print(Solution().reverse(node))
# 54321
```

Maximum In A Stack

```
class MaxStack(object):
  def __init__(self):
    self.stack = []
    self.maxes = []

  def push(self, val):
    self.stack.append(val)
    if self.maxes and self.maxes[-1] > val:
      self.maxes.append(self.maxes[-1])
    else:
      self.maxes.append(val)

  def pop(self):
    if self.maxes:
      self.maxes.pop()
    return self.stack.pop()

  def max(self):
    return self.maxes[-1]

s = MaxStack()
s.push(1)
s.push(2)
s.push(3)
s.push(2)
print('max', s.max())
```

```
print(s.pop())
print('max', s.max())
print(s.pop())
print('max', s.max())
print(s.pop())
print('max', s.max())
print(s.pop())
```

<u>Mark As Complete</u>

<u>Course Schedule</u>

```
class Solution:

 def _hasCycle(self, graph, course, seen, cache):
  if course in cache:
   return cache[course]

  if course in seen:
   return True
  if course not in graph:
   return False

  seen.add(course)
  ret = False
  for neighbors in graph[course]:
   if self._hasCycle(graph, neighbors, seen,
cache):
    ret = True
    break
  seen.remove(course)

  cache[course] = ret
  return ret

 def canFinish(self, numCourses, prerequisites):
  graph = {}
  for prereq in prerequisites:
   if prereq[0] in graph:
    graph[prereq[0]].append(prereq[1])
   else:
    graph[prereq[0]] = [prereq[1]]

  for course in range(numCourses):
   if self._hasCycle(graph, course, set(), {}):
    return False

  return True


print(Solution().canFinish(2, [[1, 0]]))
# True

print(Solution().canFinish(2, [[1, 0], [0, 1]]))
# False
```

<u>Find Pythagorean Triplets</u>

```
def findPythagoreanTriplets(nums):
 for a in nums:
  for b in nums:
   for c in nums:
    if a*a + b*b == c*c:
     return True
 return False

def findPythagoreanTriplets2(nums):
 squares = set([n*n for n in nums])

 for a in nums:
  for b in nums:
```

```
    if a * a + b * b in squares:
      return True
  return False

print(findPythagoreanTriplets2([3, 5, 12, 5, 13]))
# True
```

Push Dominoes

```
class Solution(object):
  def pushDominoes(self, dominoes):
    forces = [0] * len(dominoes)
    max_force = len(dominoes)

    force = 0
    for i, d in enumerate(dominoes):
      if d == 'R':
        force = max_force
      if d == 'L':
        force = 0
      else:
        force = max(0, force - 1)
      forces[i] = force

    for i in range(len(dominoes) - 1, -1, -1):
      d = dominoes[i]
      if d == 'L':
        force = max_force
      if d == 'R':
        force = 0
      else:
        force = max(0, force - 1)
      forces[i] -= force

    result = ''
    for f in forces:
      if f == 0:
        result += '.'
      elif f > 0:
        result += 'R'
      else:
        result += 'L'
    return result

print(Solution().pushDominoes('..R...L..R.'))
# ..RR.LL..RR
```

Simple Calculator

```
class Solution(object):
  def __eval_helper(self, expression, index):
    op = '+'
    result = 0
    while index < len(expression):
      char = expression[index]
      if char in ('+', '-'):
        op = char
      else:
        value = 0
        if char.isdigit():
          value = int(char)
        elif char == '(':
          (value, index) =
self.__eval_helper(expression, index + 1)
        if op == '+':
          result += value
        if op == '-':
          result -= value
      index += 1
    return (result, index)
```

```
 def eval(self, expression):
   return self.__eval_helper(expression, 0)[0]

print(Solution().eval('(1 + (2 + (3 + (4 + 5))))'))
# 15
```

Product Of Array Except Self

```
class Solution:
 def productExceptSelf(self, nums):
   right = [1] * len(nums)
   prod = 1
   for i in range(len(nums) - 2, -1, -1):
     prod *= nums[i+1]
     right[i] = prod

   prod = 1
   for i in range(1, len(nums)):
     prod *= nums[i-1]
     right[i] *= prod

   return right


print(Solution().productExceptSelf([1, 2, 3, 4]))
# [24, 12, 8, 6]
```

Non Decreasing Array

```
class Solution(object):
 def checkPossibility(self, nums):
   invalid_index = -1
   for i in range(len(nums) - 1):
     if nums[i] > nums[i+1]:
       if invalid_index != -1:
         return False
       invalid_index = i

   if invalid_index == -1:
     return True
   if invalid_index == 0:
     return True
   if invalid_index == len(nums) - 2:
     return True
   if nums[invalid_index] <= nums[invalid_index +
2]:
     return True
   if nums[invalid_index - 1] <=
nums[invalid_index + 1]:
     return True
   return False

print(Solution().checkPossibility([4, 1, 2]))
# True

print(Solution().checkPossibility([3, 2, 4, 1]))
# False
```

Word Search

```
class Grid(object):
 def __init__(self, matrix):
   self.matrix = matrix

 def __wordSearchRight(self, index, word):
   for i in range(len(self.matrix[index])):
     if word[i] != self.matrix[index][i]:
       return False
   return True
```

```
 def __wordSearchBottom(self, index, word):
  for i in range(len(self.matrix)):
   if word[i] != self.matrix[i][index]:
    return False
  return True

 def wordSearch(self, word):
  for i in range(len(self.matrix)):
   if self.__wordSearchRight(i, word):
    return True
  for i in range(len(self.matrix[0])):
   if self.__wordSearchBottom(i, word):
    return True
  return False

matrix = [
  ['F', 'A', 'C', 'I'],
  ['O', 'B', 'Q', 'P'],
  ['A', 'N', 'O', 'B'],
  ['M', 'A', 'S', 'S']]

print(Grid(matrix).wordSearch('FOAM'))
# True
```

Top K Frequent Elements

```
import heapq
import collections

class Solution(object):
 def topKFrequent(self, nums, k):
  count = collections.defaultdict(int)
  for n in nums:
   count[n] += 1
  heap = []
  for num, c in count.items():
   heap.append((-c, num))
  heapq.heapify(heap)

  result = []
  for i in range(k):
   result.append(heapq.heappop(heap)[1])
  return result

print(Solution().topKFrequent([1, 1, 1, 2, 2, 3, ],
2))
# [1, 2]
```

Remove Kth Last Element From Linked List

```
class Node:
 def __init__(self, val, next):
  self.val = val
  self.next = next

 def __str__(self):
  n = self
  answer = ''
  while n:
   answer += str(n.val)
   n = n.next
  return answer

def remove_kth_from_linked_list(node, k):
 slow, fast = node, node
 for i in range(k):
  fast = fast.next
 if not fast:
```

```python
  return node.next

 prev = None
 while fast:
  prev = slow
  fast = fast.next
  slow = slow.next
 prev.next = slow.next
 return node

head = Node(1, Node(2, Node(3, Node(4,
Node(5, None)))))
print(head)
# 12345

head = remove_kth_from_linked_list(head, 1)
print(head)
# 1234
```

Mark As Complete

Valid Parentheses

```python
class Solution(object):
 def isValid(self, s):

  parens = {
   '[' : ']',
   '{' : '}',
   '(' : ')',
  }
  inv_parens = {v:k for k,v in parens.items()}

  stack = []
  for c in s:
   if c in parens:
    stack.append(c)
   elif c in inv_parens:
    if len(stack) == 0 or stack[-1] !=
inv_parens[c]:
     return False
    else:
     stack.pop()
  return len(stack) == 0

print(Solution().isValid('(){([])}'))
# True

print(Solution().isValid('(){(['))
# False
```

Find the Kth Largest Element in a List

```python
import heapq
import random


def findKthLargest(nums, k):
 return sorted(nums)[len(nums) - k]


def findKthLargest2(nums, k):
 return heapq.nlargest(k, nums)[-1]


def findKthLargest3(nums, k):
 def select(list, l, r, index):
  if l == r:
   return list[l]
```

```
    pivot_index = random.randint(l, r)
    # move pivot to the beginning of list
    list[l], list[pivot_index] = list[pivot_index], list[l]
    # partition
    i = l
    for j in range(l + 1, r + 1):
      if list[j] < list[l]:
        i += 1
        list[i], list[j] = list[j], list[i]
    # move pivot to the correct location
    list[i], list[l] = list[l], list[i]
    # recursively partition one side
    if index == i:
      return list[i]
    elif index < i:
      return select(list, l, i - 1, index)
    else:
      return select(list, i + 1, r, index)
  return select(nums, 0, len(nums) - 1, len(nums)
- k)
```

```
print(findKthLargest3([3, 5, 2, 4, 6, 8], 3))
# 5
```

3 Sum

class Solution:

```
  def threeSumBruteForce(self, nums):
    result = []
    for i1 in range(0, len(nums)):
      for i2 in range(i1+1, len(nums)):
        for i3 in range(i2+1, len(nums)):
          a, b, c = nums[i1], nums[i2], nums[i3]
          if a + b + c == 0:
            result.append([a, b, c])
    return result
```

```
  def threeSumHashmap(self, nums):
    nums.sort()
    result = []
    for i in range(len(nums)):
      self.twoSumHashmap(nums, i, result)
    return result
```

```
  def twoSumHashmap(self, nums, start, result):
    values = {}
    target = -nums[start]
    for i in range(start+1, len(nums)):
      n = nums[i]
      diff = target - n
      if diff in values:
        result.append([n, diff, nums[start]])
      values[n] = 1
```

```
  def threeSumIndices(self, nums):
    nums.sort()
    result = []
    for i in range(len(nums)):
      self.twoSumIndices(nums, i, result)
    return result
```

```
  def twoSumIndices(self, nums, start, result):
    low = start + 1
    high = len(nums) - 1
    while low < high:
      sum = nums[start] + nums[low] + nums[high]
      if sum == 0:
        result.append([nums[start], nums[low],
nums[high]])
```

```
      low += 1
      high -= 1
    elif sum < 0:
      low += 1
    else:
      high -= 1
```

```
print(Solution().threeSumBruteForce([-1, 0, 1, 2,
-4, -3]))
# [[-1, 0, 1], [1, 2, -3]]

print(Solution().threeSumHashmap([-1, 0, 1, 2,
-4, -3]))
# [[2, 1, -3], [1, 0, -1]]

print(Solution().threeSumIndices([-1, 0, 1, 2, -4,
-3]))
# [[-3, 1, 2], [-1, 0, 1]]
```

<u>Spiral Traversal</u>

```
RIGHT = 0
UP = 1
LEFT = 2
DOWN = 3


class Grid(object):
  def __init__(self, matrix):
    self.matrix = matrix

  def __next_position(self, position, direction):
    if direction == RIGHT:
      return (position[0], position[1] + 1)
    elif direction == DOWN:
      return (position[0] + 1, position[1])
    elif direction == LEFT:
      return (position[0], position[1] - 1)
    elif direction == UP:
      return (position[0] - 1, position[1])

  def __next_direction(self, direction):
    return {
        RIGHT: DOWN,
        DOWN: LEFT,
        LEFT: UP,
        UP: RIGHT
    }[direction]

  def __is_valid_position(self, pos):
    return (0 <= pos[0] < len(self.matrix) and
            0 <= pos[1] < len(self.matrix[0]) and
            self.matrix[pos[0]][pos[1]] is not None)

  def spiralPrint(self):
    remaining = len(self.matrix) * len(self.matrix[0])
    current_direction = RIGHT
    current_position = (0, 0)
    result = ''
    while remaining > 0:
      remaining -= 1
      result += str(self.matrix[current_position[0]]
                    [current_position[1]]) + ' '
      self.matrix[current_position[0]]
[current_position[1]] = None

      next_position =
self.__next_position(current_position,
current_direction)
```

```python
      if not self.__is_valid_position(next_position):
        current_direction =
self.__next_direction(current_direction)
        current_position = self.__next_position(
            current_position, current_direction)
      else:
        current_position = self.__next_position(
            current_position, current_direction)

    return result


grid = [[1,  2,  3,  4,  5],
        [6,  7,  8,  9,  10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20]]

print(Grid(grid).spiralPrint())
# 1 2 3 4 5 10 15 20 19 18 17 16 11 6 7 8 9 14 13
12
```

__**Unique Paths**__

```python
class Solution(object):
  def uniquePaths(self, m, n):
    if m == 1 or n == 1:
      return 1
    return self.uniquePaths(m - 1, n) +
self.uniquePaths(m, n - 1)

  def uniquePathsDP(self, m, n):
    cache = [[0] * n] * m
    for i in range(m):
      cache[i][0] = 1
    for j in range(n):
      cache[0][j] = 1

    for c in range(1, m):
      for r in range(1, n):
        cache[c][r] = cache[c][r-1] + cache[c-1][r]
    return cache[-1][-1]

print(Solution().uniquePaths(5, 3))
# 15

print(Solution().uniquePathsDP(5, 3))
# 15
```

__**Queue Using Stacks**__

```python
class Queue(object):
  def __init__(self):
    self.s1 = []
    self.s2 = []

  def enqueue(self, val):
    self.s1.append(val)

  def dequeue(self):
    if self.s2:
      return self.s2.pop()

    if self.s1:
      while self.s1:
        self.s2.append(self.s1.pop())
      return self.s2.pop()

    return None
```

```python
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
print(q.dequeue())
print(q.dequeue())
print(q.dequeue())
print(q.dequeue())
# 1 2 3 4
```

Remove Zero Sum Consecutive Nodes

```python
import collections

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

  def __repr__(self):
    n = self
    ret = ''
    while n:
      ret += str(n.val) + ' '
      n = n.next
    return ret


class Solution(object):
  def removeZeroSumSublists(self, node):
    sumToNode = collections.OrderedDict()
    sum = 0
    dummy = Node(0)
    dummy.next = node
    n = dummy
    while n:
      sum += n.val
      if sum not in sumToNode:
        sumToNode[sum] = n
      else:
        prev = sumToNode[sum]
        prev.next = n.next
        while list(sumToNode.keys())[-1] != sum:
          sumToNode.popitem()
      n = n.next
    return dummy.next


# 3, 1, 2, -1, -2, 4, 1
n = Node(3)
n.next = Node(1)
n.next.next = Node(2)
n.next.next.next = Node(-1)
n.next.next.next.next = Node(-2)
n.next.next.next.next.next = Node(4)
n.next.next.next.next.next.next = Node(1)
print(Solution().removeZeroSumSublists(n))
# 3, 4, 1
```

Merge K Sorted Linked Lists

```python
class Node(object):
  def __init__(self, val, next=None):
    self.val = val
    self.next = next

  def __str__(self):
```

```
  c = self
  answer = ''
  while c:
    answer += str(c.val) if c.val else ""
    c = c.next
  return answer


def merge(lists):
  arr = []
  for node in lists:
    while node:
      arr.append(node.val)
      node = node.next
  head = root = None
  for val in sorted(arr):
    if not root:
      head = root = Node(val)
    else:
      root.next = Node(val)
      root = root.next
  return head


def merge2(lists):
  head = current = Node(-1)

  while any(list is not None for list in lists):
    current_min, i = min((list.val, i)
                    for i, list in enumerate(lists) if list
is not None)
    lists[i] = lists[i].next
    current.next = Node(current_min)
    current = current.next

  return head.next
```

```
a = Node(1, Node(3, Node(5)))
b = Node(2, Node(4, Node(6)))

print(a)
# 135
print(b)
# 246
print(merge2([a, b]))
# 123456
```

Generate Parentheses

```
class Solution(object):
  def _genParensHelper(self, n, left, right, str):
    if left + right == 2 * n:
      return [str]

    result = []
    if left < n:
      result += self._genParensHelper(n, left + 1,
right, str+'(')

    if right < left:
      result += self._genParensHelper(n, left, right +
1, str+')')
    return result

  def genParens(self, n):
    return self._genParensHelper(n, 0, 0, '')


print(Solution().genParens(3))
# ['((()))', '(()())', '(())()', '()(())', '()()()']
```

Depth of a Binary Tree

```python
class Node(object):
  def __init__(self, val):
    self.val = val
    self.left = None
    self.right = None

  def __repr__(self):
    return self.val


def deepest(node):
  if not node:
    return 0
  return 1 + max(deepest(node.left),
deepest(node.right))


def deepest2(node, depth=0):
  if not node:
    return depth + 0

  if not node.left and not node.right:
    return depth + 1

  if not node.left:
    return deepest2(node.right, depth + 1)

  if not node.right:
    return deepest2(node.left, depth + 1)

  return max(deepest2(node.left, depth + 1),
             deepest2(node.right, depth + 1))

#    a
#   / \
#  b   c
# /
# d
#  \
#   e
root = Node('a')
root.left = Node('b')
root.left.left = Node('d')
root.left.left.right = Node('e')
root.right = Node('c')

print(deepest2(root))
# 4
```

Intersection of Two Linked Lists

```python
class Node(object):
  def __init__(self, value, next=None):
    self.value = value
    self.next = next

class Solution(object):
  def _length(self, n):
    len = 0
    curr = n
    while curr:
      curr = curr.next
      len += 1
    return len

  def intersection(self, a, b):
    lenA = self._length(a)
    lenB = self._length(b)
```

```python
  currA = a
  currB = b

  if lenA > lenB:
    for _ in range(lenA - lenB):
      currA = currA.next
  else:
    for _ in range(lenB - lenA):
      currB = currB.next

  while currA != currB:
    currA = currA.next
    currB = currB.next

  return currA

a = Node(1)
a.next = Node(2)
a.next.next = Node(3)
a.next.next.next = Node(4)

b = Node(6)
b.next = a.next.next

print(Solution().intersection(a, b).value)
# 3
```

First Missing Positive Integer

```python
class Solution(object):
 def first_missing_position(self, nums):
  hash = {}
  for n in nums:
    hash[n] = 1

  for i in range(1, len(nums)):
   if i not in hash:
    return i

  return -1


print(Solution().first_missing_position([3, 4, -1,
1]))
# 2
```

Meeting Rooms

```python
import heapq

def meeting_rooms(meetings):
 meetings.sort(key=lambda x: x[0])
 meeting_ends = []
 max_rooms = 0

 for meeting in meetings:
  while meeting_ends and meeting_ends[0] <=
meeting[0]:
   heapq.heappop(meeting_ends)
  heapq.heappush(meeting_ends, meeting[1])
  max_rooms = max(max_rooms,
len(meeting_ends))
 return max_rooms

print(meeting_rooms([[0, 10], [10, 20]]))
# 1

print(meeting_rooms([[20, 30], [10, 21], [0, 50]]))
# 3
```

Sort Colors

```
from collections import defaultdict

class Solution(object):
  def sortColors(self, colors):
    colorsMap = defaultdict(int)
    for c in colors:
      colorsMap[c] += 1

    index = 0
    for i in range(colorsMap[0]):
      colors[index] = 0
      index += 1
    for i in range(colorsMap[1]):
      colors[index] = 1
      index += 1
    for i in range(colorsMap[2]):
      colors[index] = 2
      index += 1

  def sortColor2(self, colors):
    lowIndex = 0
    highIndex = len(colors) - 1
    currIndex = 0

    while currIndex <= highIndex:
      if colors[currIndex] == 0:
        colors[lowIndex], colors[currIndex] =
colors[currIndex], colors[lowIndex]
        lowIndex += 1
        currIndex += 1
      elif colors[currIndex] == 2:
        colors[highIndex], colors[currIndex] =
colors[currIndex], colors[highIndex]
        highIndex -= 1
      else:
        currIndex += 1

colors = [0, 2, 1, 0, 1, 1, 2]
Solution().sortColors(colors)
print(colors)
# [0, 0, 1, 1, 1, 2, 2]

colors = [0, 2, 1, 0, 1, 1, 2]
Solution().sortColor2(colors)
print(colors)
# [0, 0, 1, 1, 1, 2, 2]
```

Number of Islands

```
class Solution(object):
  def num_islands(self, grid):
    if not grid or not grid[0]:
      return 0
    numRows, numCols = len(grid), len(grid[0])
    count = 0

    for row in range(numRows):
      for col in range(numCols):
        if self._is_land(grid, row, col):
          count += 1
          self._sinkLand(grid, row, col)
    return count

  def _sinkLand(self, grid, row, col):
    if not self._is_land(grid, row, col):
      return
    grid[row][col] = 0
    for d in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
```

```
      self._sinkLand(grid, row + d[0], col + d[1])

  def _is_land(self, grid, row, col):
    if row < 0 or col < 0 or row >= len(grid) or col
>= len(grid[0]):
      return False
    return grid[row][col] == 1

grid = [[1, 1, 0, 0, 0],
        [0, 1, 0, 0, 1],
        [1, 0, 0, 1, 1],
        [0, 0, 0, 0, 0]]

print(Solution().num_islands(grid))
# 3
```

Get all Values at a Certain Height in a Binary Tree

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


def valuesAtLevel(node, depth):
  if not node:
    return []

  if depth == 1:
    return [node.value]

  return valuesAtLevel(node.left, depth - 1) +
valuesAtLevel(node.right, depth - 1)


#    1
#   / \
#  2   3
# / \   \
# 4   5   7
node = Node(1)
node.left = Node(2)
node.right = Node(3)
node.right.right = Node(7)
node.left.left = Node(4)
node.left.right = Node(5)

print(valuesAtLevel(node, 3))
# [ 4, 5, 7]
```

Balanced Binary Tree

```
class Node(object):
  def __init__(self, val, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right


class Solution(object):
  # return value (isBalanced, height)
  def _is_balanced_helper(self, n):
    if not n:
      return (True, 0)
```

```
    lBalanced, lHeight =
self._is_balanced_helper(n.left)
    rBalanced, rHeight =
self._is_balanced_helper(n.right)
    return (lBalanced and rBalanced and
abs(lHeight - rHeight) <= 1,
            max(lHeight, rHeight) + 1)

  def is_balanced(self, n):
    return self._is_balanced_helper(n)[0]


n = Node(1)
n.left = Node(2)
n.left.left = Node(4)
n.right = Node(3)
#    1
#   / \
#  2   3
# /
#4
print(Solution().is_balanced(n))

n.right = None
#    1
#   /
#  2
# /
#4
print(Solution().is_balanced(n))
# False
```

Count Number of Unival Subtrees

```
class Node(object):
  def __init__(self, val):
    self.val = val
    self.left = None
    self.right = None


def count_unival_subtrees(node):
  count, is_unival =
count_unival_subtrees_helper(node)
  return count

# total_count, is_unival
def count_unival_subtrees_helper(node):
  if not node:
    return 0, True

  left_count, is_left_unival =
count_unival_subtrees_helper(node.left)
  right_count, is_right_unival =
count_unival_subtrees_helper(node.right)

  if (is_left_unival and is_right_unival and
        (not node.left or node.val == node.left.val)
and
        (not node.right or node.val ==
node.right.val)):
    return left_count + right_count + 1, True

  return left_count + right_count, False


#    0
#   / \
#  1   0
#     / \
#    1   0
```

```
#  / \
#  1   1
a = Node(0)
a.left = Node(1)
a.right = Node(0)
a.right.left = Node(1)
a.right.right = Node(0)
a.right.left.left = Node(1)
a.right.left.right = Node(1)

print(count_unival_subtrees(a))
# 5
```

Maximum Depth of a Tree

```
# Note - Recursion won't work on large trees, due to the limit on stack limit size.
# Iteration, on the other hand, uses heap space which is limited only by how
# much memory is in the computer.

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

class Solution(object):
  def maxDepth(self, n):
    stack = [(1, n)]

    max_depth = 0
    while len(stack) > 0:
      depth, node = stack.pop()
      if node:
        max_depth = max(max_depth, depth)
        stack.append((depth + 1, node.left))
        stack.append((depth + 1, node.right))
    return max_depth

  def maxDepthRecursive(self, n):
    if not n:
      return 0
    return max(self.maxDepthRecursive(n.left) + 1,
               self.maxDepthRecursive(n.right) + 1)

n = Node(1)
n.left = Node(2)
n.right = Node(3)
n.left.left = Node(4)

print(Solution().maxDepth(n))
# 3

print(Solution().maxDepthRecursive(n))
# 3
```

Group Words that are Anagrams

Note: The solution was corrected to use "+= 1" in order to account for repeating characters.

There is also a small technicality in the time-complexity analysis. In the video, I defined "n" as the number of words, and "m" as the average number of characters in each word. Therefore, in the optimal solution, we must scan through each character of each word, so the time-complexity is O(n * m). (In the video, I stated it to be linear time

"O(n)", which is also correct in a sense if you were to redefine "n" to be the "size of the input," which would be the total number of characters in all the words).

```
import collections


def hashkey(str):
  return "".join(sorted(str))


def hashkey2(str):
  arr = [0] * 26
  for c in str:
    arr[ord(c) - ord('a')] += 1
  return tuple(arr)


def groupAnagramWords(strs):
  groups = collections.defaultdict(list)
  for s in strs:
    groups[hashkey2(s)].append(s)

  return tuple(groups.values())


print(groupAnagramWords(['abc', 'bcd', 'cba', 'cbd', 'efg']))
# (['abc', 'cba'], ['bcd', 'cbd'], ['efg'])
```

Minimum Subarray Length

```
class Solution(object):
  def minSubArray(self, k, nums):
    leftIndex = rightIndex = 0
    sum = 0
    minLen = float('inf')

    while rightIndex < len(nums):
      sum += nums[rightIndex]
      while sum >= k:
        minLen = min(minLen, rightIndex - leftIndex + 1)
        sum -= nums[leftIndex]
        leftIndex += 1
      rightIndex += 1

    if minLen == float('inf'):
      return 0
    return minLen

print(Solution().minSubArray(7, [2, 3, 1, 2, 4, 3]))
# 2
```

Merge List Of Number Into Ranges

```
def makerange(low, high):
  return str(low) + '-' + str(high)



def findRanges(nums):
  if not nums:
    return []

  ranges = []
  low = nums[0]
  high = nums[0]

  for n in nums:
    if high + 1 < n:
      ranges.append(makerange(low, high))
      low = n
    high = n
  ranges.append(makerange(low, high))
```

```
 return ranges


print(findRanges([0, 1, 2, 5, 7, 8, 9, 9, 10, 11,
15]))
# ['0-2', '5-5', '7-11', '15-15']
```

Maximum Subarray

```
class Solution:
 def maxSubArray(self, nums):
  maxSum = 0
  sum = 0
  for n in nums:
   sum += n
   if sum < 0:
    sum = 0
   else:
    maxSum = max(maxSum, sum)
  return maxSum

print(Solution().maxSubArray([-2, 1, -3, 4, -1, 2,
1, -5, 4]))
# 6

print(Solution().maxSubArray([-1, -4, 3, 8, 1]))
# 12
```

Array Intersection

```
class Solution:
 def intersection(self, nums1, nums2):
  results = {}
  for num in nums1:
   if num in nums2 and num not in results:
    results[num] = 1
  return list(results.keys())

 def intersection2(self, nums1, nums2):
  set1 = set(nums1)
  set2 = set(nums2)
  return [x for x in set1 if x in set2]

 def intersection3(self, nums1, nums2):
  hash = {}
  duplicates = {}
  for i in nums1:
   hash[i] = 1
  for i in nums2:
   if i in hash:
    duplicates[i] = 1

  return tuple(duplicates.keys())

print(Solution().intersection3([4, 9, 5], [9, 4, 9, 8,
4]))
# (9, 4)
```

Invert a Binary Tree

```
class Node(object):
 def __init__(self, val, left=None, right=None):
  self.val = val
  self.left = left
  self.right = right
```

```
 def __repr__(self):
   result = self.val
   result += f"{self.left}" if self.left else ''
   result += f"{self.right}" if self.right else ''
   return result

class Solution(object):
 def invert(self, n):
   if not n:
     return None
   left = self.invert(n.left)
   right = self.invert(n.right)
   n.right = left
   n.left = right
   return n

n = Node('a')
n.left = Node('b')
n.right = Node('c')
n.left.left = Node('d')
n.left.right = Node('e')
n.right.left = Node('f')

#      a
#    / \
#   b    c
#  /\  /
# d  e  f

print(n)

#      a
#    / \
#   c    b
#    \  /\
#    f e   d

print(Solution().invert(n))
# acfbed
```

Angles of a Clock

```
def calcAngle(h, m):
  hour_angle = (360 / (12 * 60.0)) * (h * 60 + m)
  min_angle = 360 / 60.0 * m
  angle = abs(hour_angle - min_angle)
  return min(angle, 360 - angle)


print(calcAngle(3, 15))
# 7.50
print(calcAngle(3, 00))
# 90
```

Climbing Stairs

```
def staircase(n):
  if n <= 1:
    return 1
  return staircase(n-1) + staircase(n-2)


def staircase2(n):
  prev = 1
  prevprev = 1
  curr = 0

  for i in range(2, n + 1):
    curr = prev + prevprev
```

```
    prevprev = prev
    prev = curr
  return curr


print(staircase(5))
# 8

print(staircase2(5))
# 8
```

Tree Serialization

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

  def __str__(self):
    result = ''
    result += str(self.val)
    if self.left:
      result += str(self.left)
    if self.right:
      result += str(self.right)
    return result


def serialize(node):
  if node == None:
    return '#'
  return str(node.val) + ' ' + serialize(node.left) + ' '
+ serialize(node.right)


def deserialize(str):
  def deserialize_helper(values):
    value = next(values)
    if value == '#':
      return None
    node = Node(int(value))
    node.left = deserialize_helper(values)
    node.right = deserialize_helper(values)
    return node
  values = iter(str.split())
  return deserialize_helper(values)


#      1
#     / \
#    3   4
#   / \   \
#  2   5   7
tree = Node(1)
tree.left = Node(3)
tree.right = Node(4)
tree.left.left = Node(2)
tree.left.right = Node(5)
tree.right.right = Node(7)
string = serialize(tree)
print(deserialize(string))
# 132547
```

Longest Substring Without Repeating

Characters

```
def lengthOfLongestSubstring(str):
  letter_pos = {}
  start = -1
  end = 0
  max_length = 0

  while end < len(str):
    c = str[end]
    if c in letter_pos:
      start = max(start, letter_pos[c])

    max_length = max(max_length, end - start)

    letter_pos[c] = end
    end += 1
  return max_length


print(lengthOfLongestSubstring('aabcbbeacc'))
```

Circle of Chained Words

```
import collections

def is_cycle_dfs(symbol, current_word, start_word, length, visited):
  if length == 1:
    return start_word[0] == current_word[-1]

  visited.add(current_word)
  for neighbor in symbol[current_word[-1]]:
    if (neighbor not in visited and
        is_cycle_dfs(symbol, neighbor, start_word, length - 1, visited)):
      return True
  visited.remove(current_word)
  return False

def chainedWords(words):
  symbol = collections.defaultdict(list)
  for word in words:
    symbol[word[0]].append(word)

  return is_cycle_dfs(symbol, words[0], words[0], len(words), set())

print(chainedWords(['apple', 'eggs', 'snack', 'karat', 'tuna']))
# True

print(chainedWords(['apple', 'eggs', 'snack', 'karat', 'tunax']))
# False
```

Merge Intervals

```
def merge(intervals):
  results = []
  for start, end in sorted(intervals, key=lambda x: x[0]):
    if results and start <= results[-1][1]:
      prev_start, prev_end = results[-1]
      results[-1] = (prev_start, max(prev_end, end))
    else:
      results.append((start, end))

  return results
```

```
print(merge(([1, 3], [5, 8], [4, 10], [20, 25])))
# [(1, 3), (4, 10), (20, 25)]
```

Best Time to Buy And Sell Stock

```
def buy_and_sell(arr):
  max_profit = 0

  for i in range(len(arr)):
    for j in range(i, len(arr)):
      max_profit = max(max_profit, arr[j] - arr[i])
  return max_profit


print(buy_and_sell([9, 11, 8, 5, 7, 10]))


def buy_and_sell2(arr):
  max_current_price = 0
  max_profit = 0

  for price in arr[::-1]:
    max_current_price = max(max_current_price,
price)
    max_profit = max(max_profit,
max_current_price - price)

  return max_profit


print(buy_and_sell2([9, 11, 8, 5, 7, 10]))
```

Phone Numbers

```
lettersMaps = {
    1: [],
    2: ['a', 'b', 'c'],
    3: ['d', 'e', 'f'],
    4: ['g', 'h', 'i'],
    5: ['j', 'k', 'l'],
    6: ['m', 'n', 'o'],
    7: ['p', 'q', 'r', 's'],
    8: ['t', 'u', 'v'],
    9: ['w', 'x', 'y', 'z'],
    0: []
}

validWords = ['dog', 'fish', 'cat', 'fog']


def makeWords_helper(digits, letters):
  if not digits:
    word = ''.join(letters)
    if word in validWords:
      return [word]
    return []

  results = []
  chars = lettersMaps[digits[0]]
  for char in chars:
    results += makeWords_helper(digits[1:], letters
+ [char])
  return results


def makeWords(phone):
  digits = []
  for digit in phone:
    digits.append(int(digit))
  return makeWords_helper(digits, [])
```

```
print(makeWords('364'))
```

Quickselect (iterative)

```
import heapq


def findKthLargest(arr, k):
 for i in range(0, k):
  (max_value, max_index) = (arr[0], 0)
  for j in range(0, len(arr)):
   if max_value < arr[j]:
    (max_value, max_index) = arr[j], j
  arr = arr[:max_index] + arr[max_index + 1:]
 for j in range(0, len(arr)):
  if max_value < arr[j]:
   (max_value, max_index) = arr[j], j
 return max_value


def findKthLargest2(arr, k):
 return sorted(arr)[-k]


def findKthLargest2(arr, k):
 arr = list(map(lambda x: -x, arr))
 heapq.heapify(arr)
 for i in range(0, k - 1):
  heapq.heappop(arr)
 return -arr[0]
```

```
def partition(arr, low, high):
 pivot = arr[high]
 i = low
 for j in range(low, high):
  if arr[j] <= pivot:
   arr[i], arr[j] = arr[j], arr[i]
   i += 1
 arr[i], arr[high] = arr[high], arr[i]
 return i


def quickselect(arr, k):
 k = len(arr) - k
 left = 0
 right = len(arr) - 1

 while left <= right:
  pivotIndex = partition(arr, left, right)
  if pivotIndex == k:
   return arr[pivotIndex]
  elif pivotIndex > k:
   right = pivotIndex - 1
  else:
   left = pivotIndex + 1
 return -1


print(quickselect([8, 7, 2, 3, 4, 1, 5, 6, 9, 0], 3))
```

Clone Trees

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

```python
    self.left = None
    self.right = None

  def __str__(self):
    return str(self.val)

def findNode(a, b, node):
  if a == node:
    return b
  if a.left and b.left:
    found = findNode(a.left, b.left, node)
    if found:
      return found
  if a.right and b.right:
    found = findNode(a.right, b.right, node)
    if found:
      return found
  return None

def findNode2(a, b, node):
  stack = [(a, b)]
  while len(stack):
    (a,b) = stack.pop()
    if a == node:
      return b
    if a.left and b.left:
      stack.append((a.left, b.left))
    if b.right and b.right:
      stack.append((a.right, b.right))
  return None

#  1
# / \
#2   3
#   / \
#  4*  5

a = Node(1)
a.left = Node(2)
a.right = Node(3)
a.right.left = Node(4)
a.right.right = Node(5)

b = Node(1)
b.left = Node(2)
b.right = Node(3)
b.right.left = Node(4)
b.right.right = Node(5)

print(findNode2(a, b, a.right.left))
# 4
```

Level by Level Trees

```python
from collections import deque

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

def levelPrint(node):
  q = deque([node])
  result = ''
  while len(q):
    num = len(q)
    while num > 0:
      node = q.popleft()
      result += str(node.val)
      for child in node.children:
        q.append(child)
      num -= 1
    result += "\n"
```

```
  return result

tree = Node('a', [])
tree.children = [Node('b', []), Node('c', [])]
tree.children[0].children = [Node('g', [])]
tree.children[1].children = [Node('d', []), Node('e',
[]), Node('f', [])]

print(levelPrint(tree))
```

Max Connected Colors in a Grid

```
class Grid:
  def __init__(self, grid):
    self.grid = grid

  def max_connected_colors(self):
    max_n = 0
    for y in range(len(self.grid)):
      for x in range(len(self.grid[y])):
        # max_n = max(max_n, self.dfs(x, y, {}))
        max_n = max(max_n, self.dfsIterative(x, y,
{}))
    return max_n

  def colorAt(self, x, y):
    if x >= 0 and x < len(self.grid[0]) and y >= 0
and y < len(self.grid):
      return self.grid[y][x]
    return -1

  def neighbors(self, x, y):
    POSITIONS = [[-1, 0], [0, -1], [0, 1], [1, 0]]
    n = []
    for pos in POSITIONS:
      if self.colorAt(x + pos[0], y + pos[1]) ==
self.colorAt(x, y):
        n.append((x + pos[0], y + pos[1]))
    return n

  def dfs(self, x, y, visited):
    key = str(x) + ','+str(y)
    if key in visited:
      return 0
    visited[key] = True
    result = 1
    for neighbor in self.neighbors(x, y):
      result += self.dfs(neighbor[0], neighbor[1],
visited)
    return result

  def dfsIterative(self, x, y, visited):
    stack = [(x, y)]
    result = 0
    while len(stack) > 0:
      (x, y) = stack.pop()
      key = str(x) + ', ' + str(y)
      if key in visited:
        continue
      visited[key] = True

      result += 1
      for neighbor in self.neighbors(x, y):
        stack.append(neighbor)
    return result


grid = [[1, 0, 0, 1],
    [1, 1, 1, 1],
    [0, 1, 0, 0]]
```

```
print(Grid(grid).max_connected_colors())
# 7
```

Closest Points to the Origin

```
import heapq

def calcDistance(p):
  return p[0]*p[0] + p[1]*p[1]

def findClosestPoints2(points, k):
  points = sorted(points, key = lambda x:
calcDistance(x))
  return points[:k]

def findClosestPoints2(points, k):
  # ( distance, object )
  data = []
  for p in points:
    data.append((calcDistance(p), p))
  heapq.heapify(data)

  result = []
  for i in range(k):
    result.append(heapq.heappop(data)[1])
  return result

print (findClosestPoints2([[1, 1], [3, 3], [2, 2], [4,
4], [-1, -1]], 3))
```

Autocompletion

```
class Node:
  def __init__(self, isWord, children):
    self.isWord = isWord
    # {'a': Node, 'b': Node, ...}
    self.children = children

class Solution:
  def build(self, words):
    trie = Node(False, {})
    for word in words:
      current = trie
      for char in word:
        if not char in current.children:
          current.children[char] = Node(False, {})
        current = current.children[char]
      current.isWord = True
    self.trie = trie

  def autocomplete(self, word):
    current = self.trie
    for char in word:
      if not char in current.children:
        return []
      current = current.children[char]

    words = []
    self.dfs(current, word, words)
    return words

  def dfs(self, node, prefix, words):
    if node.isWord:
      words.append(prefix)
    for char in node.children:
      self.dfs(node.children[char], prefix + char,
words)
```

```
s = Solution()
s.build(['dog', 'dark', 'cat', 'door', 'dodge'])
print(s.autocomplete('do'))
# ['dog', 'door', 'dodge']
```

Fibonacci Number

```
def fib(n):
  a = 0
  b = 1
  if n == 0:
    return a
  if n == 1:
    return b

  for _ in range(2, n+1):
    value = a + b

    a = b
    b = value
  return value


print(fib(10))
# 55
```

Roman Numerals to Decimal

```
class Solution():
  def romanToInt(self, s):
    romanNumerals = {'I': 1, 'V': 5, 'X': 10,
                  'L': 50, 'C': 100, 'D': 500, 'M': 1000}
    prev = 0
    sum = 0
    for i in s[::-1]:
      curr = romanNumerals[i]
      if prev > curr:
        sum -= curr
      else:
        sum += curr
      prev = curr
    return sum

n = 'MCMIV'
print(Solution().romanToInt(n))
# 1904
```

Subarray With Target Sum

```
# def find_continuous_k(list, k):
#   for start in range(len(list)):
#     sum = 0
#     for end in range(start, len(list)):
#       sum += list[end]
#       if sum == k:
#         return list[start:end + 1]
#   return None


def find_continuous_k(list, k):
  previous_sums = {0: -1}
  sum = 0
  for index, n in enumerate(list):
    sum += n
    previous_sums[sum] = index
    if previous_sums.get(sum - k):
      return list[previous_sums[sum-k] + 1: index +
1]
```

```python
  return None


print(find_continuous_k([1, 3, 2, 5, 7, 2], 14))
```

Absolute Paths

```python
def clean_path(path):
  folders = path.split('/')
  stack = []

  for folder in folders:
    if folder == '.':
      pass
    elif folder == '..':
      stack.pop()
    else:
      stack.append(folder)
  return '/'.join(stack)

path = '/users/tech/docs/.././desk/../'
print(clean_path(path))
# /users/tech/
```

Mark As Complete

Consecutive Bit Ones

```python
def longest_run(n):
  longest_run = 0
  current_run = 0
  BITMASK = 1

  while n != 0:
    if n & BITMASK == 0:
      longest_run = max(longest_run, current_run)
      current_run = 0
    else:
      current_run += 1
    n = n >> 1
  longest_run = max(longest_run, current_run)
  return longest_run

print(longest_run(242))
# 4
```

Anagrams in a String

```python
from collections import defaultdict


def find_anagrams(a, b):
  char_map = defaultdict(int)

  for c in b:
    char_map[c] += 1

  results = []
  for i in range(len(a)):
    c = a[i]

    if i >= len(b):
      c_old = a[i - len(b)]
      char_map[c_old] += 1
      if char_map[c_old] == 0:
        del char_map[c_old]

    char_map[c] -= 1
    if char_map[c] == 0:
```

```
    del char_map[c]

  if i + 1 >= len(b) and len(char_map) == 0:
    results.append(i - len(b) + 1)

 return results


print(find_anagrams('acdbacdacb', 'abc'))
# [3, 7]
```

Check for Palindrome

```
from collections import defaultdict

def find_palindrome(str):
 char_counts = defaultdict(int)

 for c in str:
  char_counts[c] += 1

 pal = ''
 odd_char = ''
 for c, cnt in char_counts.items():
  if cnt % 2 == 0:
   pal += c * (cnt // 2)
  elif odd_char == '':
   odd_char = c
   pal += c * (cnt // 2)
  else:
   return False
 return pal + odd_char + pal[::-1]


print(find_palindrome('foxfo'))
# foxof
```

Rectangle Intersection

```
class Rectangle(object):
 def __init__(self, min_x=0, min_y=0, max_x=0,
max_y=0):
   self.min_x = min_x
   self.min_y = min_y
   self.max_x = max_x
   self.max_y = max_y

 def area(self):
   if self.min_x >= self.max_x:
     return 0
   if self.min_y >= self.max_y:
     return 0
   return (self.max_x - self.min_x) * (self.max_y -
self.min_y)


def intersect_rect(a, b):
 return Rectangle(max(a.min_x, b.min_x),
                  max(a.min_y, b.min_y),
                  min(a.max_x, b.max_x),
                  min(a.max_y, b.max_y))


a = Rectangle(0, 0, 3, 2)
b = Rectangle(1, 1, 3, 3)

intersection = intersect_rect(a, b)
print(intersection.area())
```

Find Subtree

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


n = Node(1)
n.left = Node(4)
n.right = Node(5)
n.left.left = Node(3)
n.left.right = Node(2)
n.right.left = Node(4)
n.right.right = Node(1)

b = Node(4)
b.left = Node(3)
b.right = Node(2)


def pre(n):
  if not n:
    return 'null'
  return '-' + str(n.value) + '-' + pre(n.left) + '-' +
pre(n.right)


def find_subtree(a, b):
  return pre(b) in pre(a)


def find_subtree2(a, b):
  if not a:
    return False

  is_match = a.value == b.value
  if is_match:
    is_match_left = (not a.left or not b.left) or
find_subtree2(a.left, b.left)
    if is_match_left:
      is_match_right = (not a.right or not b.right) or
find_subtree2(
          a.right, b.right)
      if is_match_right:
        return True

  return find_subtree2(a.left, b) or
find_subtree2(a.right, b)


print(find_subtree(n, b))
# True

print(find_subtree2(n, b))
# True
```

Determine if Number

```
from enum import Enum


class DigitState(Enum):
  BEGIN = 0
  NEGATIVE1 = 1
```

```
  DIGIT1 = 2
  DOT = 3
  DIGIT2 = 4
  E = 5
  NEGATIVE2 = 6
  DIGIT3 = 7


STATE_VALIDATOR = {
    DigitState.BEGIN: lambda x: True,
    DigitState.DIGIT1: lambda x: x.isdigit(),
    DigitState.NEGATIVE1: lambda x: x == '-',
    DigitState.DIGIT2: lambda x: x.isdigit(),
    DigitState.DOT: lambda x: x == '.',
    DigitState.E: lambda x: x == 'e',
    DigitState.NEGATIVE2: lambda x: x == '-',
    DigitState.DIGIT3: lambda x: x.isdigit(),
}

NEXT_STATES_MAP = {
    DigitState.BEGIN: [DigitState.NEGATIVE1,
DigitState.DIGIT1],
    DigitState.NEGATIVE1: [DigitState.DIGIT1,
DigitState.DOT],
    DigitState.DIGIT1: [DigitState.DIGIT1,
DigitState.DOT, DigitState.E],
    DigitState.DOT: [DigitState.DIGIT2],
    DigitState.DIGIT2: [DigitState.DIGIT2,
DigitState.E],
    DigitState.NEGATIVE2: [DigitState.DIGIT3],
    DigitState.DIGIT3: [DigitState.DIGIT3],
}


def parse_number(str):
  state = DigitState.BEGIN

  for c in str:
    found = False
    for next_state in NEXT_STATES_MAP[state]:
      if STATE_VALIDATOR[next_state](c):
        state = next_state
        found = True
        break
    if not found:
      return False

  return state in [DigitState.DIGIT1,
DigitState.DIGIT2, DigitState.DIGIT3]


print(parse_number('12.3'))
# True

print(parse_number('12a'))
# False
```

First Recurring Character

```
def first_recurring_character(str):
  seen = set()

  for c in str:
    if c in seen:
      return c
    seen.add(c)

  return None
```

```
print(first_recurring_character('qwertty'))
# t

print(first_recurring_character('qwerty'))
# None
```

Inorder Successor

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

  def __repr__(self):
    return f"({self.value}, {self.left}, {self.right}"


tree = Node(4)
tree.left = Node(2)
tree.right = Node(8)
tree.left.parent = tree
tree.right.parent = tree
tree.left.left = Node(1)
tree.left.left.parent = tree.left
tree.right.right = Node(7)
tree.right.right.parent = tree.right
tree.right.left = Node(5)
tree.right.left.parent = tree.right
tree.right.left.right = Node(7)
tree.right.left.right.parent = tree.right.left
tree.right.right = Node(9)
tree.right.right.parent = tree.right
```

```
#     4
#    / \
#   2   8
#  /   / \
# 1   5   9
#      \
#       7


def in_order_successor(node):
  if node.right:
    curr = node.right
    while curr.left:
      curr = curr.left
    return curr

  curr = node
  parent = curr.parent
  while parent and parent.left != curr:
    curr = parent
    parent = parent.parent
  return parent

print(in_order_successor(tree.right))
# 9

print(in_order_successor(tree.left))
# 4

print(in_order_successor(tree.right.left.right))
# 8
```

Rotate Linked List

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

  def __repr__(self):
    return f"({self.value}, {self.next})"

def rotate(node, n):
  length = 0
  curr = node
  while curr != None:
    curr = curr.next
    length +=1
  n = n % length

  slow, fast = node, node
  for i in range(n):
    fast = fast.next

  while fast.next != None:
    slow = slow.next
    fast = fast.next

  fast.next = node
  head = slow.next
  slow.next = None

  return head

node = Node(1, Node(2, Node(3, Node(4, Node(5)))))

print(rotate(node, 2))
# 4, 5, 1, 2, 3
```

Remove Duplicate From Linked List

```python
class Node(object):
  def __init__(self, value, next=None):
    self.value = value
    self.next = next

  def __repr__(self):
    return f"({self.value}, {self.next})"


def remove_duplicates(node):
  curr = node

  while curr and curr.next:
    if curr.value == curr.next.value:
      curr.next = curr.next.next
    else:
      curr = curr.next


node = Node(1, Node(2, Node(2, Node(3, Node(3)))))
remove_duplicates(node)
print(node)
# (1, (2, (3, None)))
```

Optimized List Sum

```python
class ListFastSum(object):
  def __init__(self, nums):
```

```
    self.pre = [0]

    sum = 0
    for num in nums:
      sum += num
      self.pre.append(sum)

  def sum(self, start, end):
    return self.pre[end] - self.pre[start]


print(ListFastSum([1, 2, 3, 4, 5, 6, 7]).sum(2, 5))
# 12
```

Sorted Square Numbers

```
def square_numbers(nums):
  neg_i = -1
  i = 0

  result = []
  for n in nums:
    if n >= 0:
      if neg_i == -1:
        neg_i = i - 1

      while neg_i >= 0 and nums[neg_i] < 0 and
-nums[neg_i] < nums[i]:
        val = nums[neg_i]
        result.append(val * val)
        neg_i -= 1

      val = nums[i]
      result.append(val * val)
    i += 1

  while neg_i >= 0 and nums[neg_i] < 0:
    val = nums[neg_i]
    result.append(val * val)
    neg_i -= 1

  return result


print(square_numbers([-5, -3, -1, 0, 1, 4, 5]))
```

String to Integer

```
def convert_to_int(str):
  is_negative = False
  start_index = 0
  if str[0] == '-':
    is_negative = True
    start_index = 1

  result = 0
  for c in str[start_index:]:
    result = result * 10 + ord(c) - ord('0')

  if is_negative:
    result *= -1
  return result


print(convert_to_int('-105') + 1)
# -104
```

Shortest Unique Prefix

```
class Node:
  def __init__(self):
    self.count = 0
    self.children = {}


class Trie:
  def __init__(self):
    self.root = Node()

  def insert(self, word):
    node = self.root

    for c in word:
      if c not in node.children:
        node.children[c] = Node()
      node = node.children[c]
      node.count = node.count + 1

  def unique_prefix(self, word):
    node = self.root
    prefix = ''

    for c in word:
      if node.count == 1:
        return prefix
      else:
        node = node.children[c]
        prefix += c
    return prefix


def shortest_unique_prefix(words):
  trie = Trie()
  for word in words:
    trie.insert(word)

  unique_prefixes = []
  for word in words:
    unique_prefixes.append(trie.unique_prefix(word))

  return unique_prefixes


print(shortest_unique_prefix(['jon', 'john', 'jack', 'techlead']))
```

Make the Largest Number

```
from functools import cmp_to_key

def largestNum(nums):
  sorted_nums = sorted(nums, key=cmp_to_key(
      lambda a, b:
      1 if str(a) + str(b) < str(b) + str(a)
      else -1)
  )
  return ''.join(str(n) for n in sorted_nums)


print(largestNum([17, 7, 2, 45, 72]))
# 77245217
```

N Queens

```
def nqueens_helper(n, row, col, asc_diag,
desc_diag, queen_pos):
  if len(queen_pos) == n:
    return queen_pos

  curr_row = len(queen_pos)
  for curr_col in range(n):
    if col[curr_col] and row[curr_row] and
asc_diag[curr_row + curr_col] and
desc_diag[curr_row - curr_col]:
      col[curr_col] = False
      row[curr_row] = False
      asc_diag[curr_row + curr_col] = False
      desc_diag[curr_row - curr_col] = False

      queen_pos.append((curr_row, curr_col))
      nqueens_helper(n, row, col, asc_diag,
desc_diag, queen_pos)

      if len(queen_pos) == n:
        return queen_pos

      # backtrack
      col[curr_col] = True
      row[curr_row] = True
      asc_diag[curr_row + curr_col] = True
      desc_diag[curr_row - curr_col] = True
      queen_pos.pop()

  return queen_pos


def nqueens(n):
  col = [True] * n
  row = [True] * n
  asc_diag = [True] * (n * 2 - 1)
  desc_diag = [True] * (n * 2 - 1)
  return nqueens_helper(n, col, row, asc_diag,
desc_diag, [])


print(nqueens(5))
# Q . . . .
# . . . Q .
# . Q . . .
# . . . . Q
# . . Q . .
# [(0, 0), (1, 2), (2, 4), (3, 1), (4, 3)]
```

Sum of Squares

```
def square_sums(n):
  squares = []

  i = 1
  while i*i <= n:
    squares.append(i*i)
    i = i + 1

  min_sums = [n] * (n + 1)
  min_sums[0] = 0

  for i in range(n+1):
    for s in squares:
      if i+s < len(min_sums):
        min_sums[i+s] = min(min_sums[i+s],
min_sums[i] + 1)
```

```
  return min_sums[-1]


print(square_sums(13))
# 2
```

Swap Every Two Nodes

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

  def __repr__(self):
    return f"{self.value}, ({self.next.__repr__()})"


def swap_every_two(node):
  curr = node
  while curr != None and curr.next != None:
    curr.value, curr.next.value = curr.next.value,
curr.value
    curr = curr.next.next
  return node


node = Node(1, Node(2, Node(3, Node(4,
Node(5)))))
print(swap_every_two(node))
# 2, 1, 4, 3, 5
```

Multitasking

```
def findTime(tasks, cooldown):
  lastPos = {}
  current = 0

  for task in tasks:
    if task in lastPos:
      if current - lastPos[task] <= cooldown:
        # add cooldown
        current = cooldown + lastPos[task] + 1
    lastPos[task] = current
    current = current + 1
  return current

print(findTime([1, 1, 2, 1], 2))
# 7
```

Generate Binary Search Trees

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

  def __repr__(self):
    result = str(self.value)
    if self.left:
      result = result + str(self.left)
    if self.right:
      result = result + str(self.right)
    return result


def gen_tree(nums):
  if len(nums) == 0:
```

```
    return [None]
  if len(nums) == 1:
    return [Node(nums[0])]
  bsts = []

  for n in nums:
    lefts = gen_tree(range(nums[0], n))
    rights = gen_tree(range(n + 1, nums[-1] + 1))

    for left in lefts:
      for right in rights:
        tree = Node(n, left, right)
        bsts.append(tree)

  return bsts


def generate_bst(n):
  return gen_tree(range(1, n + 1))


print(generate_bst(3))
# 5 trees
```

Zig-Zag Binary Trees

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


def zigzag_order(node):
  result = []
  currLevel = [node]
  nextLevel = []
  leftToRight = False

  while len(currLevel) > 0:
    node = currLevel.pop()
    result.append(node.value)

    if leftToRight:
      if node.left:
        nextLevel.append(node.left)
      if node.right:
        nextLevel.append(node.right)
    if leftToRight != True:
      if node.right:
        nextLevel.append(node.right)
      if node.left:
        nextLevel.append(node.left)

    if len(currLevel) == 0:
      leftToRight = not leftToRight
      currLevel = nextLevel
      nextLevel = []

  return result


n7 = Node(7)
n6 = Node(6)
n5 = Node(5)
n4 = Node(4)
n3 = Node(3, n6, n7)
n2 = Node(2, n4, n5)
n1 = Node(1, n2, n3)

print(zigzag_order(n1))
```

Balanced Binary Trees

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


def tree_height(node):
  if node is None:
    return 0

  heightLeft = tree_height(node.left)
  heightRight = tree_height(node.right)

  if heightLeft >= 0 and heightRight >= 0 and
abs(heightLeft - heightRight) <= 1:
    return max(heightLeft, heightRight) + 1
  return -1


def is_tree_balanced(node):
  return tree_height(node) != -1


n4 = Node(4)
n3 = Node(3)
n2 = Node(2, n4)
n1 = Node(1, n2, n3)

#     1
#    / \
#   2   3
#  /
# 4
print(is_tree_balanced(n1))
# True

n4 = Node(4)
n2 = Node(2, n4)
n1 = Node(1, n2, None)

#     1
#    /
#   2
#  /
# 4
print(is_tree_balanced(n1))
# False
```

Character Mapping

```
def has_character_map(s1, s2):
  if len(s1) != len(s2):
    return False

  chars = {}
  for i in range(len(s1)):
    if s1[i] not in chars:
      chars[s1[i]] = s2[i]
    else:
      if chars[s1[i]] != s2[i]:
        return False
  return True


print(has_character_map('abc', 'def'))
# True
```

```
print(has_character_map('aac', 'def'))
# False
```

Reverse Polish Notation Calculator

```
def calc(inputs):
  stack = []

  for i in inputs:
    if i in ('-', '+', '*', '/'):
      b = stack.pop()
      a = stack.pop()
      if i == '-':
        stack.append(a - b)
      if i == '+':
        stack.append(a + b)
      if i == '*':
        stack.append(a * b)
      if i == '/':
        stack.append(a / b)
    else:
      stack.append(i)
  return stack[0]

print(calc([1, 2, 3, '+', 2, '*', '-']))
```

Maze Paths

```
def paths_through_maze(maze):
  paths = [[0] * len(maze[0]) for _ in
range(len(maze))]
  paths[0][0] = 1
  for i, row in enumerate(maze):
    for j, val in enumerate(row):
      if val == 1 or (i == 0 and j == 0):
        continue

      leftPaths = 0
      topPaths = 0
      if i > 0:
        leftPaths = paths[i - 1][j]
      if j > 0:
        topPaths = paths[i][j-1]
      paths[i][j] = leftPaths + topPaths
  print(paths)
  return paths[-1][-1]


print(paths_through_maze([[0, 1, 0],
                          [0, 0, 1],
                          [0, 0, 0]]))
```

Filter Leaves of a Binary Tree

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

  def __repr__(self):
    return f"{self.value}, ({self.left.__repr__()}),
({self.right.__repr__()})"


def filter(node, n):
  if not node:
```

```
    return None

  node.left = filter(node.left, n)
  node.right = filter(node.right, n)

  if node.value != n and not node.left and not
node.right:
    return None

  return node


#     1
#    / \
#   2   1
#  /   /
# 2   1
n1 = Node(1, Node(2, Node(2), Node(1,
Node(1))))
print(filter(n1, 2))
```

Frequent Subtree Sum

```
from collections import defaultdict


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


def _build_frequencies(root, counter):
  if root == None:
    return 0
  total = root.val + \
    _build_frequencies(root.left, counter) + \
    _build_frequencies(root.right, counter)
  counter[total] += 1
  return total


def most_freq_subtree_sum(root):
  counter = defaultdict(int)
  _build_frequencies(root, counter)
  most_common_sum = 0
  for k in list(counter):
    if counter[k] > counter[most_common_sum]:
      most_common_sum = k
  return most_common_sum


root = Node(3, Node(1), Node(-3))
print(most_freq_subtree_sum(root))
# 1
```

Partition a List

```
def partition(nums, k):
  low = 0
  high = len(nums) - 1

  i = 0
  while i <= high:
    n = nums[i]
    if n > k:
      nums[high], nums[i] = nums[i], nums[high]
      high -= 1
    if n < k:
```

```python
    nums[low], nums[i] = nums[i], nums[low]
    low += 1
    i += 1
  if n == k:
    i += 1

 return nums


def partitionSort(nums, k):
 return sorted(nums)


def partitionCopy(nums, k):
 a = []
 b = []
 for n in nums:
  if n < k:
    a.append(n)
  else:
    b.append(n)
 return a + b


print(partition([8, 9, 9, 2, 4, 1, 1, 0], 3))
```

Arithmetic Binary Tree

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

def evaluate(node):
 operators = {
    '+': lambda a, b: a + b,
    '-': lambda a, b: a - b,
    '/': lambda a, b: a / b,
    '*': lambda a, b: a * b,
 }

 if node.value in operators:
   fn = operators[node.value]
   return fn(evaluate(node.left),
evaluate(node.right))
 else:
   return node.value

 return 0


node = Node('*')
node.left = Node('+')
node.right = Node('+')
node.left.left = Node(3)
node.left.right = Node(2)
node.right.left = Node(4)
node.right.right = Node(5)

print(evaluate(node))
```

Searching A Matrix

```python
def searchMatrix(mat, value):
 if len(mat) == 0:
   return False

 rows = len(mat)
```

```python
    cols = len(mat[0])

    low = 0
    high = rows * cols
    while low < high:
        mid = (low + high) // 2
        mid_value = mat[mid // cols][mid % cols]

        if mid_value == value:
            return True
        if mid_value < value:
            low = mid + 1
        else:
            high = mid

    return False


mat = [
    [1, 3, 5, 8],
    [10, 11, 15, 16],
    [24, 27, 30, 31],
]

print(searchMatrix(mat, 4))
# False

print(searchMatrix(mat, 10))
# True
```

H Index

```python
def hIndex(pubs):
    n = len(pubs)
    freqs = [0] * (n + 1)

    for pub in pubs:
        if pub >= n:
            freqs[n] += 1
        else:
            freqs[pub] += 1

    total = 0
    i = n
    while i >= 0:
        total += freqs[i]
        if total >= i:
            return i
        i -= 1
    return 0


print(hIndex([5, 3, 3, 1, 0]))
# 3
```

Number of 1 Bits

```python
def one_bits(n):
    count = 0
    while n > 0:
        if n & 1:
            count = count + 1
        n = n >> 1
    return count

print(one_bits(23))
# 0b10111
```

Jump To The End

```
def jumpToEnd(nums):
  hops = [float('inf')] * len(nums)
  hops[0] = 0

  for i, n in enumerate(nums):
    for j in range(1, n + 1):
      if i + j < len(hops):
        hops[i + j] = min(hops[i + j], hops[i] + 1)
      else:
        break
  return hops[-1]


print(jumpToEnd([3, 2, 5, 1, 1, 9, 3, 4]))
# 2
```

Fixed Point

```
def find_fixed_point_helper(low, high, nums):
  if low == high:
    return None

  mid = int((low + high) / 2)
  if nums[mid] == mid:
    return mid
  if nums[mid] < mid:
    return find_fixed_point_helper(mid+1, high,
nums)
  else:
    return find_fixed_point_helper(low, mid, nums)

def find_fixed_point(nums):
  return find_fixed_point_helper(0, len(nums),
nums)


def find_fixed_point_iterative(nums):
  low = 0
  high = len(nums)

  while (low != high):
    mid = int((low + high) / 2)
    if nums[mid] == mid:
      return mid
    if nums[mid] < mid:
      low = mid + 1
    else:
      high = mid

  return None


print(find_fixed_point([-5, 1, 3, 4]))
# 1

print(find_fixed_point_iterative([-5, 1, 3, 4]))
# 1
```

Number of Cousins

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

```python
    self.left = left
    self.right = right


class Solution(object):
  def _nodes_at_height(self, node, height, exclude):
    if node == None or node == exclude:
      return []
    if height == 0:
      return [node.value]
    return (self._nodes_at_height(node.left, height - 1, exclude) +
            self._nodes_at_height(node.right, height - 1, exclude))

  def _find_node(self, node, target, parent, height):
    if not node:
      return False
    if node == target:
      return (height, parent)
    return (self._find_node(node.left, target, node, height + 1) or
            self._find_node(node.right, target, node, height + 1))

  def list_cousins(self, node, target):
    height, parent = self._find_node(node, target, None, 0)
    return self._nodes_at_height(node, height, parent)


#     1
#    / \
#   2   3
#  / \   \
# 4   6   5
root = Node(1)
root.left = Node(2)
root.left.left = Node(4)
root.left.right = Node(6)
root.right = Node(3)
root.right.right = Node(5)
print(Solution().list_cousins(root, root.right.right))
# [4, 6]
```

Mark As Complete

Longest Increasing Subsequence

```python
def longest_increasing_subsequence(arr):
  cache = [1] * len(arr)
  for i in range(1, len(arr)):
    for j in range(i):
      if arr[i] > arr[j]:
        cache[i] = max(cache[i], cache[j] + 1)
  return max(cache)


print(longest_increasing_subsequence(
    [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3]))
# 5
```

Distribute Bonuses

```
def getBonuses(performances):
  count = len(performances)
  bonuses = [1] * count

  for i in range(1, count):
    if performances[i - 1] < performances[i]:
      bonuses[i] = bonuses[i - 1] + 1

  for i in range(count - 2, -1, -1):
    if performances[i + 1] < performances[i]:
      bonuses[i] = max(bonuses[i], bonuses[i + 1] +
1)

  return bonuses

print(getBonuses([1, 2, 3, 4, 3, 1]))
# [1, 2, 3, 4, 2, 1]
```

Word Concatenation

```
class Solution(object):
  def findAllConcatenatedWords(self, words):
    wordDict = set(words)
    cache = {}
    return [word for word in words if
self._canForm(word, wordDict, cache)]

  def _canForm(self, word, wordDict, cache):
    if word in cache:
      return cache[word]
    for index in range(1, len(word)):
      prefix = word[:index]
      suffix = word[index:]
      if prefix in wordDict:
        if suffix in wordDict or self._canForm(suffix,
wordDict, cache):
          cache[word] = True
          return True
    cache[word] = False
    return False

input = ['cat', 'cats', 'dog', 'catsdog']
print(Solution().findAllConcatenatedWords(input))
# ['catsdog']
```

Running Median

```
import heapq

def add(num, min_heap, max_heap):
  if len(min_heap) + len(max_heap) <= 1:
    heapq.heappush(max_heap, -num)
    return

  median = get_median(min_heap, max_heap)
  if num > median:
    heapq.heappush(min_heap, num)
  else:
    heapq.heappush(max_heap, -num)

def rebalance(min_heap, max_heap):
  if len(min_heap) > len(max_heap) + 1:
    root = heapq.heappop(min_heap)
    heapq.heappush(max_heap, -root)
  elif len(max_heap) > len(min_heap) + 1:
```

```
    root = -heapq.heappop(max_heap)
    heapq.heappush(min_heap, root)

def print_median(min_heap, max_heap):
  print(get_median(min_heap, max_heap))


def get_median(min_heap, max_heap):
  if len(min_heap) > len(max_heap):
    return min_heap[0]
  elif len(min_heap) < len(max_heap):
    return -max_heap[0]
  else:
    return (min_heap[0] + -max_heap[0]) / 2.0


def running_median(stream):
  min_heap = []
  max_heap = []
  answer = []
  for num in stream:
    add(num, min_heap, max_heap)
    rebalance(min_heap, max_heap)
    answer.append(get_median(min_heap,
max_heap))
  return answer

print(running_median([2, 1, 4, 7, 2, 0, 5]))
# [2, 1.5, 2, 3, 2, 2, 2]
```