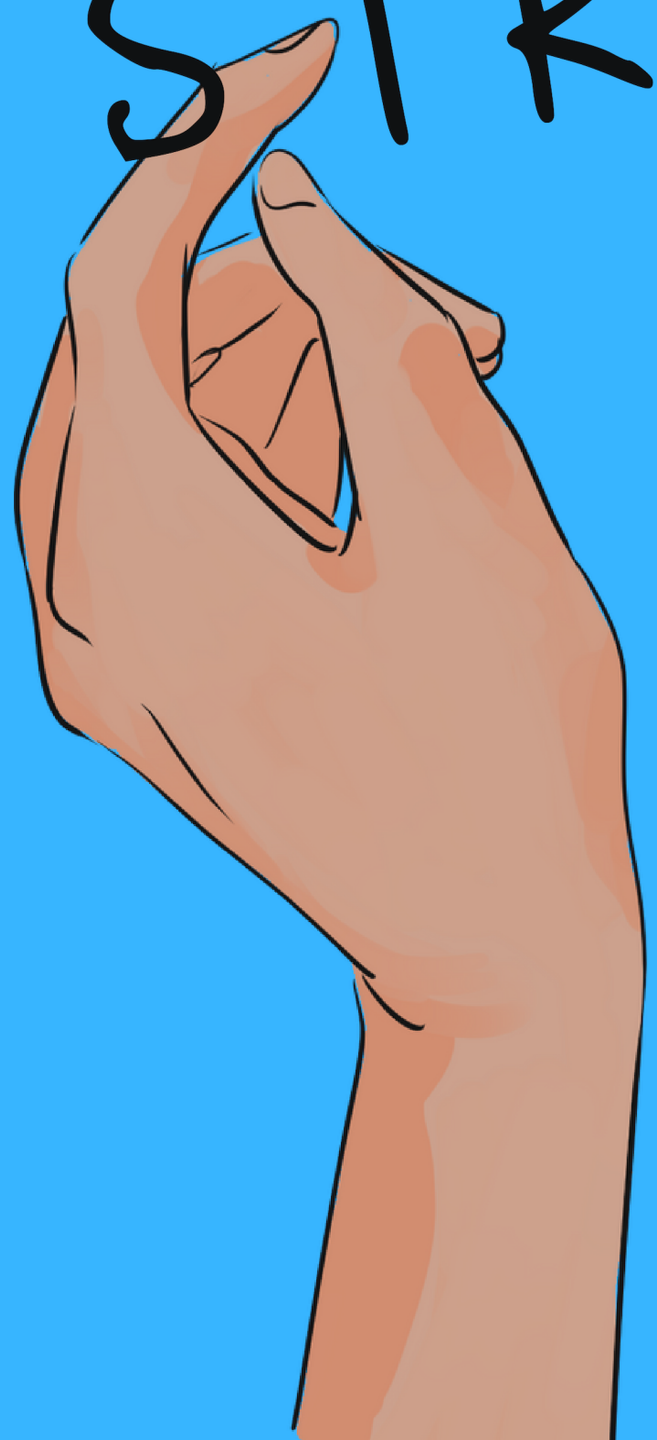
a textbook
by Chris Ibe

ALGORITHMS & DATA STRUCTURES



DataStructuresAlgorithm_Master

In this repo we explore data structures and algorithms in depth. Please enjoy!

Table Of Content

- Arrays & Strings
- Stacks & Queues
- Linked Lists
- Binary Trees
- Binary Search Trees
- Graphs
- Dynamic Programming
- Tries
- Extras
- Tips & Tricks

Arrays & Strings

1. [**Max Value**] Write a function that takes in a list/ array of numbers and returns the maximum value in the array. Solve without any build in methods.

```
def max_value(nums):
 max_num = -float('inf')
 for n in nums:
   if n > max_num:
     max_num = n
 return max_num
```

2. [**Prime Numbers**] Write a function that takes in a list/ array of numbers and returns an array with only the prime numbers from the original list/ array. If there are no primes, it returns an empty list.

```
def is_prime(n):
 if n < 2: return False
 for i in range (2, int(n**(1/2)) + 1):
   if n % i == 0: return False
 return True
```

```
 nums = [1,7, 17,4,4,2,8,10,9]
 prime_nums = list(filter(is_prime, nums))
```

3. [**Uncompress**] Write a function that takes in a compessed version of a string and uncompresses/ expands it out completely. For example, if the input is `2a3b` it'll return `aabbb`.

```
def uncompress(s):
 s_out = ''
 l,r = 0,0
 while r < len(s):
   while s[r].isnumeric():
     r += 1
   s_out += s[r] * int(s[l:r])
   r += 1
   l = r
 return s_out
```

```
 n is the length of s
 Time: O(n)
 Space: O(n)
```

4. [**Compress**] Write a function that takes in a uncompessed version of a string and compresses it. For example, if the input is `ccaaatsss` it'll return `2c3at3s`.

```
def uncompress(s):
 s_out = ''
 l,r = 0,0
 while r < len(s):
   while r< len(s) and s[l]==s[r]:
     r += 1
   s_out += str(r-l) + s[l] if (r-l) > 1 else s[l]
   l = r
 return s_out
```

```
 n is the length of s
 Time: O(n)
 Space: O(n)
```

5. [**Reverse String**] [Leetcode 151] Write a function that takes in a list of strings and returns the reverse of the list of strings. For example, if the input is `[hello]` it'll return `[olleh]`.

```
def reverseString(s):
 l,r = 0,len(s)-1
 while l < r:
   s[l], s[r] = s[r], s[l]
   l += 1
```

```
    r -= 1
  return s
```

```
def reverseString(s):
  return (' '.join(s.split()[::-1]))
```

```
def reverseString(s):
  return ' '.join(reversed(s.split()))
```

```
  n is the length of s
  Time: O(n)
  Space: O(1)
```

6. [**Anagrams**] [**Leetcode 242**] Write a function that takes in two strings and returns a boolean indicating whether both strings are anagrams or not. For example, the strings `monkeyswrite` and `newyorktimes` are anagrams so the function should return `True`.

```
def anagrams(s1, s2):
  memo = {}
  for c in s1:
    if c not in memo:
      memo[c] = 0
    memo[c] += 1
  for c in s2:
    if c not in memo:
      memo[c] = 0
    memo[c] -= 1
  for item in memo:
    if memo[item] != 0:
      return False
  return True
```

`METHOD #2`

```
def anagrams2(s1, s2):
  from collections import Counter
  return Counter(s1) == Counter(s2)
```

`METHOD #3`

```

```

```
def anagrams3(s1, s2):
 return sorted(s1) == sorted(s2)
```

```
def anagrams3(s1, s2):
 if len(s) != len(t): return False
 memo = [0]*26
 for c in s: memo[ord(c) - ord('a')] += 1
 for c in t: memo[ord(c) - ord('a')] -= 1
 for item in memo:
   if memo[item] != 0: return False
 return True
```

```
  n is the length of s1
  m is the length of s2
  Time: O(n+m)
  Space: O(n+m)
```

7. [**First Unique Character**] [**Leetcode 387**] Write a function that takes a string and returns the index of the first unique character in the string.

```
def firstUniqueCharacter(s):
 memo = {}
 for i,char in enumerate(s):
   if char not in memo:
     memo[char] = i
   else:
     memo[char] = -1
 for item in memo:
   if memo[item] != -1:
     return memo[item]
 return -1
```

```
  n is the length of s
  Time: O(n)
  Space: O(n)
```

8. [**Contains Duplicates**] [**Leetcode 217**] Write a function that takes in a list of numbers and returns `True` if the list contains a duplicate and `False` otherwise.

```
def containsDuplicate(nums):
 memo = {}
```

```
  for n in nums:
    if n in memo:
      return True
    memo[n] = 1
  return False
```

```
def containsDuplicate(nums):
 memo = set(nums)
 return len(memo) != len(nums)
```

```
def containsDuplicate(nums):
 from collections import Counter
 c = Counter(nums)
 for item in c:
   if c[item] > 1: return True
 return False
```

```
  n is the length of the list
  Time: O(n)
  Space:O(n)
```

9. [**Contains Duplicates II**] [**Leetcode 219**] Write a function that takes in a list of numbers and a constant `k` and returns `True` if the list contains a set duplicates whose indicies are also at most `k` distance but returns `False` otherwise.

```
def containsDuplicateII(nums, k):
 memo = {}
 for i,n in enumerate(nums):
   if n in memo and i - memo[n] <= k:
     return True
   memo[n] = i
 return False
```

```
  n is the length of the list
  Time: O(n)
  Space:O(n)
```

10. [**Find Duplicates**] Write a function that takes a list of numbers and returns a list containing all the duplicates (occurs exactly twice) in the input list.

```
def findDuplicates(nums):
 memo = {}
 for n in nums:
   if n not in memo:
     memo[n] = 0
   memo[n] += 1
 return [key for (key, value) in memo.items() if value == 2]
```

```
 n is the length of s
 Time: O(n)
 Space: O(n)
```

11. [**Most Frequent Character**] Write a function that takes a string and returns the most frequent character in that string and its number of occurance. For example, the string `mississippi` has the most frequent character `i` or `s` and they both occur `4` times.

```
def mostFrequentCharacter(s):
 memo = {}
 for c in s:
   if c not in memo:
     memo[c] = 0
   memo[c] += 1
 # finding the max
 max_key, max_value = None, 0
 for item in memo:
 if memo[item] > max_value:
   max_key, max_value = item, memo[item]
 return (max_key, max_value)
```

`METHOD #2`

```
def mostFrequentCharacter(s):
 from collections import Counter
 memo = Counter(s)
 # finding the max
 max_key, max_value = None, 0
 for c in s:
 if best is None or memo[c] > memo[max_key]:
   max_key, max_value = c, memo[c]
 return (max_key, max_value)
```

```
 n is the length of s
 Time: O(n)
```

```
  Space: O(n)
```

12. [**Find Difference**] [Leetcode 389] Write a function that takes a string and returns the index of the first unique character in the string.

```
def findTheDifference(s, t):
  memo = {}
  for c in t:
    if c not in memo: memo[c] = 0
    memo[c] += 1
  for c in s:
    memo[c] -= 1
  for item in memo:
    if memo[item] != 0:
      return item
```

```
def findTheDifference(s, t):
  memo = {}
  for char in (s+t):
    if char in memo:
      memo[char] += 1
    else:
      memo[char] = 1
  for item in memo:
    if memo[item] % 2 == 1:
      return item
```

```
  n is the length of s
  Time: O(n)
  Space: O(n)
```

13. [Two Sum] Write a function that takes a list and a target sum and returns a pair of unique indices of numbers that add up to the target sum.

```
def twoSum(nums, target):
 memo = {}
 for i, n in enumerate(nums):
   diff = target - n
   if diff in memo:
     return (memo[diff], i)
   memo[n] = i
```

```
  n is the length of the list
  Time: O(n)
  Space: O(n)
```

14. [**Two Sum II**] [**Leetcode 167**] Write a function that takes a sorted list and a target sum and returns a pair of unique indices (1-indexed) of numbers that add up to the target sum.

```
def twoSum(nums, target):
  l, r = 0, len(nums)-1
  while l < r:
    two_sum = nums[l] + nums[r]
    if two_sum < target:
      l += 1
    elif two_sum > target:
      r -= 1
    else:
      return [l+1, r+1]
```

```
  n is the length of the list
  Time: O(n)
  Space: O(n)
```

15. [**Two Prod**] Write a function that takes a list and a target product and returns a pair of unique indices of numbers that multiply up to the target product.

```
def twoProd(nums, target):
 memo = {}
 for i, n in enumerate(nums):
   comp = int(target / n)
   if comp in memo:
     return (memo[comp], i)
   memo[n] = i
```

```
  n is the length of the list
  Time: O(n)
  Space: O(n)
```

16. [**Intersection**] [**Leetcode 349**] Write a function that takes in two lists and returns a new list containing elements that are in both lists.

```

```

```
def intersection(a, b):
 result = []
 for item in b:
   if item in a:
     result.append(item)
 return result
```

```
def intersection(a, b):
 a = set(a)
 return [n for n in b if n in a]
```

```
  n is the length of list a
  m is the length of list b
  Time: O(n*m)
  Space: O(min(n,m))
```

17. [**Move Zeros**] Write a function that takes a list of numbers and rearranges the elements such that 0s appear at the end. This should be done inplace without creating a new list.

```
def moveZeros(nums):
  l,r = 0, len(nums)-1
  while l < r:
    while nums[l] != 0:
      l += 1
    while nums[r] == 0:
      r -= 1
    nums[l], nums[r] = nums[r], nums[l]
    l += 1
    r -= 1
  return nums
```

```
def moveZeros(nums):
  idx = 0
  for i in range(len(nums)):
    if nums[i] != 0:
      nums[idx] = nums[i]
      idx += 1
  for i in range(idx, len(nums)):
    nums[i] = 0
  return nums
```

```
  n is the length of list
  Time: O(n)
  Space: O(1)
```

18. [**Container With Most Water**] [**Leetcode 11**] Given an integer array where each element represents vertical lines with tha x-axis. Write a function that returns the container that can contain the most water.

```
def maxArea(height):
  max_area = 0
  l,r = 0, len(height)-1
  while (l < r):
    if height[l] < height[r]:
      max_area = max(max_area, height[l]*(r-l))
      l += 1
    else:
      max_area = max(max_area, height[r]*(r-l))
      r -= 1
  return max_area
```

```
  n is the length of list
  Time: O(n)
  Space: O(1)
```

Stacks & Queues

1. [**Remove Consecutive Duplicates**] Given a string, return a string where ll consecutive duplicates have been removed. Example, for the input string `abbccwaabba` the function would return `awa`.

```
def removeDuplicates(s):
 stack = [s[0]]
 for i in range(1, len(s)):
   if len(stack) and stack[-1] == s[i]:
     while i < len(s) and s[i-1] == s[i]:
       i += 1
     stack.pop()
   else:
     stack.append(s[i])
 return "".join(stack)
```

```
  n is the length of the string
  Time: O(n)
```

```
  Space: O(n)
```

2. [**Is Subsequence**] [Leetcode 392] Write a function that takes in two strings `s` and `t` and returns a boolean indicating whether `s` is a subsequence of `t`.

```
def isSubsequence(s, t):
  if len(s) == 0: return True
  if len(t) == 0: return False
  q = []
  for c in s: q.append(c)
  for c in t:
    if len(q) == 0: return True
    if c == q[0]: q.pop(0)
  return False if len(q) != 0 else True
```

```
def isSubsequence(s, t):
  idx_t, idx_s = 0, 0
  while idx_t < len(t) and idx_s < len(s):
    if t[idx_t] == s[idx_s]:
      idx_s += 1
    idx_t += 1
  return idx_s == len(s)
```

```
def isSubsequence(s, t):
  t = iter(t)
  return all(c in t for c in s)
```

```
  n is the length of string s
  m is the length of string t
  Time: O(max(n,m))
  Space: O(1)
```

Linked Lists

1. [**Create Linked List**] Create a linked list of the following values `A->B->C->D`.

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

```
a, b, c, d = Node('A'), Node('B'), Node('C'), Node('D')
a.next, b.next, c.next, = b, c, d
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

2. [**Append Value**] Given the head of a linked list like `A->B->C->D` and a Node value to append like `z`. Write a function to append the value to the end of the linked list.

```
def append(head, data):
 '''Iterative Approach'''
 new_node, cur = Node(data), head
 while cur.next:
   cur = cur.next
 cur.next = new_node
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(1)
```

3. [**Prepend Value**] Given the head of a linked list like `A->B->C->D` and a Node value to prepend (add to front) like `z`. Write a function to prepend the value to the front of the linked list.

```
def prepend(head, data):
 '''Iterative Approach'''
 new_node = Node(data)
 new_node.next = head
 return new_node
```

```
  n is the length of the linked list.
  Time: O(1)
  Space: O(1)
```

4. [**Print Linked List**] Given the head of a linked list like `A->B->C->D`. Write a function to print all values in the list in the right order.

```
def printList(head):
 '''Iterative Approach'''
 cur = head
```

```
 while cur:
   print(cur.val, end="->")
   cur = cur.next
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(1)
```

```
def printList(head):
 '''Recursieve Approach'''
 if not head: return None
 print(head.val, end="->")
 printList(head.next)
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

5. [**Linked List Values**] Given the head of a linked list like `A->B->C->D`. Write a function to return all values in the list in the right order in an array/ list.

```
def LinkedListValues(head):
 '''Iterative Approach'''
 cur, result = head, []
 while cur:
   result.append(cur.val)
   cur = cur.next
 return result
```

```
def LinkedListValues(head):
 '''Recursieve Approach'''
 if not head: return []
 return [head.val, *LinkedListValues(head.next)]
```

```
def LinkedListValues(head):
 '''Recursieve Approach'''
 result = []
 def fillValues(head, values):
   if not head: return
   values.append(head.val)
```

```
    fillValues(head.next, values)
  fillValues(head, result)
  return result
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

6. [**Create Linked List**] Given a list/ array of values like `[1, 2, 3, 5]`. Write a function to create a linked list containing each item of the list as nodes and return the head of the linked list. The example above should return `1->2->3->5`.

```
def createLinkedList(values):
  '''Iterative Approach'''
  dummy = Node(None)
  cur = dummy
  for item in values:
    cur.next = Node(item)
    cur = cur.next
  return dummy.next
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

```
def createLinkedList(values):
  '''Recursieve Approach'''
  if not values or len(values)==0: return None
  head = Node(values[0])
  head.next = createLinkedList(values[1:])
  return head
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n^2)
```

```
def createLinkedList(values, i=0):
  '''Recursieve Approach'''
  if i >= len(values): return None
  head = Node(values[i])
```

```
 head.next = createLinkedList(values, i+1)
 return head
```

```
 n is the length of the linked list.
 Time: O(n)
 Space: O(n)
```

7. [**Linked List to String**] Given the head of a linked list like `A->B->C->D`. Write a function to return all values in the list in the right order as a string

```
def LinkedListValues(head):
 '''Iterative Approach'''
 cur, result = head, ''
 while cur:
   result += cur.val
   cur = cur.next
 return result
```

```
def LinkedListValues(head):
 '''Recursieve Approach'''
 if not head: return ''
 return head.val + LinkedListValues(head.next)
```

```
 n is the length of the linked list.
 Time: O(n)
 Space: O(n)
```

8. [**Sum Linked List**] Given the head of a linked list like `1->2->3->4`. Write a function to return the sum of all values in the list.

```
def sumList(head):
 '''Iterative Approach'''
 cur, result = head, 0
 while cur:
   result += cur.val
   cur = cur.next
 return result
```

```
 n is the length of the linked list.
 Time: O(n)
```

```
  Space: O(1)
```

```
def sumList(head):
 '''Recursieve Approach'''
 if not head: return 0
 return head.val + sumList(head.next)
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

9. [**Palindrome List**] Given the head of a linked list like `r->a->c->e->c->a->r`. Write a function that returns a boolean indicating whether or not the linked list is a palindrome. A palindrome is a sequence that reads the same forward and backwards.

```
def isPalindrome(head):
 '''Iterative Approach'''
 cur, values = head, []
 while cur:
   values.append(cur.val)
   cur = cur.next
 return values == values[::-1]
```

```
def isPalindrome2(head):
 '''Recursieve Approach'''
 def listValues(head):
   if not head: return []
   return [head.val, *LinkedListValues(head.next)]
 values = listValues(head)
 return values == values[::-1]
```

```
def isPalindrome3(head):
 '''Iterative Approach'''
 values = []
 while head:
   values.append(head.val)
   head = head.next

 l, r = 0, len(values)-1
 while l < r:
   if values[l] != values[r]:
     return False
```

```
    l += 1
    r -= 1
  return True
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

10. [**Find Value**] Given the head of a linked list and a target value like `a->d->c->d->e` and `c`. Write a function that returns a boolean indicating whether or not the traget value is contained in the linked list.

```
def findValue(head, target):
 '''Iterative Approach'''
 if not head: return False
 cur = head
 while cur:
   if cur.val == target:
     return True
   cur = cur.next
 return False
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def findValue(head, target):
 '''Recursieve Approach'''
 if not head: return False
 if head.val == target: return True
 return findValue(head.next, target)
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

11. [**Get Node Value**] Given the head of a linked list and an index value like `a->d->c->d->e` and `2`. Write a function that returns the value at that specific index. In this example, the value `c` is at index `2`. Return `None` otherwise.

```
def getNodeValue(head, index):
 '''Iterative Approach'''
 cur, count = head, 0
 while cur:
   if count == index:
     return cur.val
   cur, count = cur.next, count + 1
 return None
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def getNodeValue(head, index):
 '''Recursieve Approach'''
 if not head: return None
 if index == 0: return head.val
 return getNodeValue(head.next, index-1)
```

```
 n is the length of the linked list.
 Time: O(n)
 Space: O(n)
```

12. [**Middle Value**] Given the head of a linked list like `a->d->c->d->e`. Write a function that returns the value of the middle node in the linked list. The example above should return `c`. if there's an even number of nodes, it should return the second middle.

```
def middleValue(head):
 '''Iterative Approach'''
 cur, values = head, []
 while cur:
   values.append(cur.val)
   cur = cur.next
 return values[len(values)//2]
```

```
n is the length of the linked list.
Time: O(n)
Space: O(n)
```

```
def middleValue(head):
 '''Runner Approach'''
 slow, fast = head, head
 while fast and fast.next:
   slow, fast = slow.next, fast.next.next
 return slow.val
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(1)
```

13. [**Linked List Cycle**] Given the head of a linked list like `a->b->c->d->b`. Write a function that returns a boolean indicating whether or not the list contains a cycle.

```
def listHasCycle(head):
  '''Iterative Approach'''
  cur, memo = head, set()
  while cur:
    if cur in memo: return True
    memo.add(cur)
    cur = cur.next
  return False
```

```
n is the length of the linked list.
Time: O(n)
Space: O(n)
```

```
def listHasCycle(head):
  '''Runner Approach'''
  slow, fast = head, head
  while fast and fast.next:
    slow, fast = slow.next, fast.next.next
    if slow == fast: return True
  return slow.val
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(1)
```

14. [**Reverse List**] Given the head of a linked list like `a->b->c->d->e`. Write a function that reverses the order of the nodes in the list. The example above should return `e->d->c->b->a`.

```
def reverseList(head):
  '''Iterative Approach'''
  prev, cur = None, head
  while cur:
    cur.next, prev, cur = prev, cur, cur.next
  return prev
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def reverseList(head, prev=None):
  '''Recursieve Approach'''
  if not head: return prev
  next = head.next
  head.next = prev
  return reverseList(next, head)
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

15. [**Remove Duplicates**] Given the head of a linked list like `a->b->b->c->d->d->e`. Write a function that returns the list but with each item occuring only once. The example above should return `a->b->c->d->e`.

```
def removeDuplicates(head):
  '''Works only if the list is sorted'''
  dummy, dummy.next = Node(None), head
  prev, cur = dummy, head
  while cur:
    if cur.val == prev.val:
      prev.next = cur.next
    else:
      prev = cur
    cur = cur.next
  return dummy.next
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def removeDuplicates(head):
  '''Works for unsorted list also'''
  prev, cur, memo = None, head, set()
  while cur:
    if cur.val in memo:
      prev.next = cur.next
    else:
      memo.add(cur.val)
      prev = cur
    cur = cur.next
  return head
```

```
  n is the length of the linked list.
  Time: O(n)
  Space: O(n)
```

16. [**Zipper List**] Given the heads of two linked lists `a->b` and `w->x->y->z`. Write a function that returns a zippered list containing alternating nodes of both lists. The example above should return `a->w->b->x->y->z`.

```
def zipperLists(head1, head2):
  '''Iterative Approach'''
  cur1, cur2, tail = head1.next, head2, head1
  count = 0
  while cur1 and cur2:
    if count % 2 == 0:
      tail.next = cur2
      cur2 = cur2.next
    else:
      tail.next = cur1
      cur1 = cur1.next
    count += 1
    tail = tail.next
  if cur1: tail.next = cur1
  if cur2: tail.next = cur2
  return head1
```

```
n is the length of list1.
m is the length of list2
```

```
Time: O(min(n,m))
Space: O(1)
```

```
def zipperLists(head1, head2):
  '''Recursive Approach'''
  if not head1 and not head2: return None
  if not head1: return head2
  if not head2: return head1

  n1 = head1.next
  n2 = head2.next
  head1.next = head2
  head2.next = zipperLists(n1, n2)
  return head1
```

```
n is the length of list1.
m is the length of list2
Time: O(min(n,m))
Space: O(min(n,m))
```

17. [**Merge List**] Given the heads of two sorted linked lists `1->2->6->10` and `3->7->8->12`. Write a function that returns a single sorted linked list from the two inpute lists. The example above should return `1->2->3->6->7->8->10->12`.

```
def mergeLists(head1, head2):
  '''Iterative Approach'''
  dummy = Node(None)
  cur = dummy
  while head1 and head2:
    if head1.val < head2.val:
      cur.next = head1
      head1 = head1.next
    else:
      cur.next = head2
      head2 = head2.next
    cur = cur.next
  if head1: cur.next = head1
  if head2: cur.next = head2
  return dummy.next
```

```
n is the length of list1.
m is the length of list2
Time: O(min(n,m))
Space: O(1)
```

```
def mergeLists(head1, head2):
  '''Recursive Approach'''
  if not head1 and not head2: return None
  if not head1: return head2
  if not head2: return head1

  if head1.val < head2.val:
    n1 = head1.next
    head1.next = mergeLists(n1, head2)
    return head1
  else:
    n2 = head2.next
    head2.next = mergeLists(head1, n2)
    return head2
```

```
n is the length of list1.
m is the length of list2
Time: O(min(n,m))
Space: O(min(n,m))
```

18. [**Univalue List**] Given the head of a linked list like `2->2->2->2`. Write a function that returns a boolean indicating whether the linked list contains exactly one unique value.

```
def isUniqueValue(head):
  '''Iterative Approach'''
  cur = head
  while cur:
    if head.val != cur.val: return False
    cur = cur.next
  return True
```

```
def isUniqueValue(head):
  '''Iterative Approach'''
  while head.next:
    if head.val != head.next.val: return False
    head = head.next
  return True
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def isUniqueValue(head, prev_val=None):
  '''Recursive Approach'''
  if not head: return True
  if prev_val is None or head.val == prev_val:
    return isUniqueValue(head.next, prev_val=head.val)
  else:
    return False
```

```
def isUniqueValue(head, prev_val=None):
  '''Recursive Approach'''
  if not head: return True
  if prev_val is not None and head.val != prev_val:
    return False
  else:
    return isUniqueValue(head.next, prev_val=head.val)
```

```
n is the length of the linked list.
Time: O(n)
Space: O(n)
```

19. [**Longest Streak**] Given the head of a linked list like `3->3->9->9->9->2`. Write a function that returns the length of the longest consecutive streak of the same value. The example above should return `3`.

```
def longestStreak(head):
  '''Iterative Approach'''
  count, max_count = 0, 0
  p1, p2 = head, head
  while p1 and p2:
    if p2.val == p1.val:
      count += 1
      p2 = p2.next
    else:
      count = 0
      p1 = p2
    max_count = max(max_count, count)
  return max_count
```

```
def longestStreak(head):
  '''Iterative Approach'''
  count, max_count = 0, 0
  p1, p2 = head, head
  while p1 and p2:
```

```
    while p2 is not None and p1.val == p2.val:
      count += 1
      p2 = p2.next
    max_count = max(max_count, count)
    count = 0
    p1 = p2
  return max_count
```

```
def longestStreak(head):
  '''Iterative Approach'''
  count, max_count = 0, 0
  prev_val, cur = None, head
  while cur:
    if cur.val == prev_val:
      count += 1
    else:
      count = 1
    max_count = max(max_count, count)
    prev_val = cur.val
    cur = cur.next
  return max_count
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

20. [**Remove Node**] Given the head of a linked list like `3->3->9->` and a node value to delete like `9`. Write a function that deletes the first occuring node with that value. The example above should return `3->3->`.

```
def removeNode(head, target_val):
  '''Iterative Approach'''
  if head.val == target_val:
    return head.next
  prev, cur = None, head
  while cur:
    if cur.val == target_val:
      prev.next = cur.next
      break
    prev, cur = cur, cur.next
  return head
```

```
n is the length of the linked list.
Time: O(n)
```

```
Space: O(1)
```

```
def removeNode(head, target_val):
  '''Recursive Approach'''
  if not head: return None
  if head.val == target_val:
    return head.next
  head.next = remove_node(head.next, target_val)
  return head
```

```
n is the length of the linked list.
Time: O(n)
Space: O(n)
```

21. [**Remove Nodes**] Given the head of a linked list like `3->9->3->9->` and a node value to delete like `9`. Write a function that deletes all ocurrance of the nodes with that value. The example above should return `3->3->`.

```
def removeNode(head, target_val):
  '''Iterative Approach'''
  if head.val == target_val:
    return head.next
  prev, cur = None, head
  while cur:
    if cur.val == target_val:
      prev.next = cur.next
    prev, cur = cur, cur.next
  return head
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

22. [**Insert Node**] Given the head of a linked list like `3->3->9->`, a value like `6` and and index like `2`. Write a function that insertes the given value in the specified index of the linked list. The example above should return `3->3->6->9`.

```
def insertNode(head, value, index):
  '''Iterative Approach'''
  value_node = Node(value)
  if index == 0:
    value_node.next = head
```

```
    return value_node
  cur, count = head, 0
  while cur:
    if count = index-1:
      temp = cur.next
      cur.next = value_node
      value_node.next = temp
    count += 1
    cur = cur.next
  return head
```

```
n is the length of the linked list.
Time: O(n)
Space: O(1)
```

```
def insertNode(head, value, index, count=0):
  '''Recursive Approach'''
  if index == 0:
    new_head = Node(value)
    new_head.next = head
    return new_head
  if not head: return None
  if count = index-1:
    temp = head.next
    head.next = Node(value)
    head.next.next = temp
  insertNode(head.next, value, index, count+1)
  return head
```

```
n is the length of the linked list.
Time: O(n)
Space: O(n)
```

23. [**Add Lists**] Given the heads of two linked lists (with each list representing a number written in the reverse order. For example the numbers `621` and `354` would be given as `1->2->6` and `4->5->3`. Write a function that returns the head of a new linked list representing the sum of both input lists. The example above would result in `5->7->9` which is `975`.

```
def addLists(head1, head2):
  '''Iterative Approach'''
  cur1, cur2, carry = head1, head2, 0
  dummy = Node(None)
  tail = dummy
  while cur1 or cur2 or carry:
```

```
    val1 = cur1.val if cur1 else 0
    val2 = cur2.val if cur2 else 0

    value = (val1 + val2 + carry) % 10
    carry = (val1 + val2 + carry) // 10
    tail.next = Node(value)

    cur1 = cur1.next if cur1 else None
    cur2 = cur2.next if cur2 else None
    tail = tail.next
  return dummy.next
```

```
n is the length of list1.
m is the length of list2
Time: O(max(n,m))
Space: O(max(n,m))
```

```
def addLists(head1, head2, carry=0):
  '''Recursive Approach'''
  if not head1 and not head2 and not carry: return None
  val1 = head1.val if head1 else 0
  val2 = head2.val if head2 else 0

  value = (val1 + val2 + carry) % 10
  carry = (val1 + val2 + carry) // 10
  result = Node(value)

  n1 = head1.next if head1 else None
  n2 = head2.next if head2 else None
  result.next = addLists(n1, n2, carry)

  return result
```

```
n is the length of list1.
m is the length of list2
Time: O(max(n,m))
Space: O(max(n,m))
```

Binary Trees

1. [**Create Binary Tree**] Create a binary tree of the below structure.

```
  '''
      A
```

```
      / \
     B   C
    / \   \
   D   E   F
  '''
```

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

a,b,c,d,e,f = Node('A'), Node('B'), Node('C'), Node('D'), Node('E'),
Node('F')
a.left, a.right = b, c
b.left, b.right = d, e
c.right = f
```

```
  n is the number of nodes.
  Time: O(n)
  Space: O(n)
```

2. [**Depth First Values**] Given the root of a binary tree, write a function to print all values in the tree using a depth first search approach (DFS).

```
def depthFirstValues(root):
  '''Iterative Approach'''
  s = [root]
  while s:
    cur = s.pop()
    print(cur.val, end=",")
    if cur.right: s.append(cur.right)
    if cur.left: s.append(cur.left)
```

```
def depthFirstValues(root):
 '''Recursive Approach'''
 if not root: return
 print(root.val, end=",")
 depthFirstValues(root.left)
 depthFirstValues(root.right)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

3. [**Breadth First Values**] Given the root of a binary tree, write a function to print all values in the tree using a breadth first search approach (BFS).

```
def breadthFirstValues(root):
  '''Iterative Approach'''
  q = [root]
  while q:
    cur = q.pop(0)
    print(cur.val, end=",")
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
```

```
n is the number of nodes
Time: O(n)
Space: O(n^2)
```

```
def breadthFirstValues(root):
  '''Iterative Approach'''
  from collections import deque
  q = deque([root])
  while q:
    cur = q.popleft()
    print(cur.val, end=",")
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

4. [**Depth First List**] Given the root of a binary tree, write a function to return a list of all values in the tree using a depth first search approach (dfs).

```
def depthFirstValues(root):
 '''Iterative Approach'''
  if not root: return []
  res = []
```

```
  s = [root]
  while s:
    cur = s.pop()
    res.append(cur.val)
    if cur.right: s.append(cur.right)
    if cur.left: s.append(cur.left)
  return res
```

```
def depthFirstValues(root):
 '''Recursieve Approach'''
 if not root: return []
 return [root.val, *depthFirstValues(root.left),
*depthFirstValues(root.right)]
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

5. [**Breadth First List**] Given the root of a binary tree, write a function to return a list of all values in the tree using a depth first search approach (dfs).

```
def breadthFirstValues(root):
 '''Iterative Approach'''
  if not root: return []
  res = []
  q = [root]
  while q:
    cur = q.pop(0)
    res.append(cur.val)
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
  return res
```

```
n is the number of nodes
Time: O(n)
Space: O(n^2)
```

6. [**Bottom Left Value**] Given the root of a binary tree, write a function to return the leftmost value in the last row of the tree.

Iterative Approach (BFS)

```

```

```
def findBottomLeftValue(root):
  if not root: return
  q = [root]
  while q:
    cur = q.pop(0)
    if cur.right: q.append(cur.right)
    if cur.left: q.append(cur.left)
  return cur.val
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

7. [**Bottom Right Value**] Given the root of a binary tree, write a function to return the rightmost value in the last row of the tree.

   Iterative Approach (BFS)

```
def findBottomRightValue(root):
 from collections import deque
  if not root: return
  q = deque([root])
  while q:
    cur = q.popleft()
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
  return cur.val
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

8. [**Tree Sum**] Given the root of a binary tree that contains all number values, write a function to return the sum of all the values in the tree.

```
def treeSum(root):
 '''Iterative DFS Approach'''
  if not root: return 0
  res = 0
  s = [root]
  while s:
    cur = s.pop()
    res += cur.val
    if cur.right: s.append(cur.right)
```

```
    if cur.left: s.append(cur.left)
  return res
```

```
def treeSum(root):
 '''Recursieve DFS Approach'''
 if not root: return 0
 return root.val + treeSum(root.left) + treeSum(root.right)
```

```
def treeSum(root):
 '''Recursieve DFS Approach'''
 def treeValues(node):
   if not root: return []
   return [root.val, *treeValues(node.left), *treeValues(node.right)]
 return sum(treeValues(root))
```

```
def treeSum(root):
 '''Iterative BFS Approach'''
  if not root: return 0
  res = 0
  q = [root]
  while q:
    cur = q.pop(0)
    res += cur.val
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
  return res
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

9. [**Find Value**] Given the root of a binary tree and a target value, write a function to return a boolean indicating whether or not the target value is in the tree.

```
def findTarget(root, target):
 '''Iterative DFS Approach'''
  if not root: return False
  s = [root]
  while s:
    cur = s.pop()
    if cur.val == target: return True
    if cur.right: s.append(cur.right)
```

```
    if cur.left: s.append(cur.left)
  return False
```

```
def findTarget(root, target):
 '''Recursieve DFS Approach'''
 if not root: return False
 if root.val == target: return True
 return findTarget(root.left, target) or findTarget(root.right,
target)
```

```
def findTarget(root, target):
 '''Recursieve DFS Approach'''
 def treeValues(node):
   if not root: return []
   return [root.val, *treeValues(node.left), *treeValues(node.right)]
 return (target in treeValues(root))
```

```
def findTarget(root, target):
 '''Iterative BFS Approach'''
  if not root: return False
  q = [root]
  while q:
    cur = q.pop(0)
    if cur.val == target: return True
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
  return False
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

10. [**Min Value**] Given the root of a binary tree containing all numerical values, write a function to return the minimum value in the tree. Assume an non-empty tree.

```
def findMinValue(root):
 '''Iterative DFS Approach'''
 min_val = float("inf")
 s = [root]
 while s:
   cur = s.pop()
   min_val = min(cur.val, min_val)
```

```
    if cur.right: s.append(cur.right)
    if cur.left: s.append(cur.left)
  return min_val
```

```
def findMinValue(root):
 '''Recursieve DFS Approach'''
 if not root: return float("inf")
 return min(root.val, findMinValue(root.left),
findMinValue(root.rightt))
```

```
def findMinValue(root):
 '''Recursieve DFS Approach'''
 def treeValues(node):
    if not root: return []
    return [root.val, *treeValues(node.left), *treeValues(node.right)]
 return min(treeValues(root))
```

```
def findMinValue(root):
 '''Iterative BFS optimal Approach'''
 from collection import deque
 if not root: return
 min_val = float("inf")
 q = [root]
 while q:
    cur = q.popleft()
    min_val = min(cur.val, min_val)
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
 return min_val
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

11. [**Max Value**] Given the root of a binary tree containing all numerical values, write a function to return the maximum value in the tree. Assume an non-empty tree.

```
def findMaxValue(root):
 '''Recursieve DFS Approach'''
 if not root: return float("-inf")
 return max(root.val, findMaxValue(root.left),
findMaxValue(root.rightt))
```

```
def findMaxValue(root):
 '''Iterative BFS optimal Approach'''
 from collection import deque
 if not root: return
 max_val = float("-inf")
 q = [root]
 while q:
   cur = q.popleft()
   max_val = max(cur.val, min_val)
   if cur.left: q.append(cur.left)
   if cur.right: q.append(cur.right)
 return max_val
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

12. [**Min Path Sum**] Given the root of a binary tree containing all numerical values, write a function to return the minimum sum of any root to leaf path within the tree. Assume an non-empty tree.

```
def minPathSum(root):
 '''Recursieve DFS Approach'''
 if not root: return float("inf")
 if not root.left and not root.right: return root.val
 return root.val + min(minPathSum(root.left), minPathSum(root.rightt))
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

13. [**Max Path Sum**] Given the root of a binary tree containing all numerical values, write a function to return the maximum sum of any root to leaf path within the tree. Assume an non-empty tree.

```
def maxPathSum(root):
 '''Recursieve DFS Approach'''
 if not root: return float("-inf")
 if not root.left and not root.right: return root.val
 return root.val + max(maxPathSum(root.left), maxPathSum(root.rightt))
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

14. [**Min Path**] Given the root of a binary tree containing all numerical values, write a function to return the minimum path from the root to any leaf within the tree. Assume an non-empty tree.

```
def minPath(root):
 '''Recursieve DFS Approach'''
 if not root: return [float("inf")]
 if not root.left and not root.right: return [root.val]
 left_path, right_path = minPath(root.left), minPath(root.right)
 if sum(left_path) < sum(right_path):
  return [root.val, *left_path]
 else:
  return [root.val, *right_path]
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

15. [**Max Path**] Given the root of a binary tree containing all numerical values, write a function to return the maximum path from the root to any leaf within the tree. Assume an non-empty tree.

```
def maxPath(root):
 '''Recursieve DFS Approach'''
 if not root: return [float("-inf")]
 if not root.left and not root.right: return [root.val]
 left_path, right_path = maxPath(root.left), maxPath(root.right)
 if sum(left_path) > sum(right_path):
  return [root.val, *left_path]
 else:
  return [root.val, *right_path]
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

16. [**Path Finder**] Given the root of a binary tree and a target value, write a function to return an array representing the path to the target value. Assume an non-empty tree containing unique values.

```

```

```
def pathFinder(root, target):
 '''Recursieve DFS Approach'''
 if not root: return None
 if root.val == target: return [root.val]
 left_path, right_path = pathFinder(root.left, target),
pathFinder(root.right, target)
 if left_path: return [root.val, *left_path]
 if right_path: return [root.val, *right_path]
 return None
```

```
n is the number of nodes
Time: O(n^2)
Space: O(n)
```

```
def pathFinder(root, target):
 '''Recursieve DFS Approach'''
 def helper_pathFinder(root, target):
  if not root: return None
  if root.val == target: return [root.val]
  left_path, right_path = helper_pathFinder(root.left, target),
helper_pathFinder(root.right, target)
  if left_path:
    left_path.append(root.val)
    return left_path
  if right_path:
    right_path.append(root.val)
    return right_path
  return None

 path = helper_pathFinder(root, target)
 return path[::-1] if path else None
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

17. [**Lowest Common Ancestor**] [Leetcode 236] Given the root of a binary tree and a target value, write a function to return an array representing the path to the target value. Assume an non-empty tree containing unique values.

```
def lowestCommonAncestor(root, val1, val2):
  path1 = pathFinder(root, val1)
  path2 = set(pathFinder(root, val2))
```

```
  for val in path1:
    if val in path2:
      return val

def pathFinder(root, target):
  if not root: return None
  if root.val == target: return [root.val]

  left_path = pathFinder(root.left, target)
  if left_path:
    left_path.append(root.val)
    return left_path

  right_path = pathFinder(root.right, target)
  if right_path:
    right_path.append(root.val)
    return right_path
  return None
```

```
def lowestCommonAncestor(root, p, q):
  if not root: return None
  if root.val == p or root.val == q: return root.val

  left = lowestCommonAncestor(root.left, p, q)
  right = lowestCommonAncestor(root.right, p, q)

  if (not left and not right ): return None
  if (left and right): return root.val

  if left: return left
  else: return right
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

18. [**Path Sum**] Given the root of a binary tree and an integer target value, write a function to return a boolean indicating wheater or not theres a root-to-lead path that adds up to the target sum.

```
def hasPathSum(root, target):
  '''Recursive Approach'''
  if not root: return False
  rem = root.val - target
  if not root.left and not root.right and rem == 0: return True
  return self.hasPathSum(root.left, rem) or
self.hasPathSum(root.right, rem)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

19. [**Invert Tree**] Given the root of a binary tree, write a function to invert the tree and return it's root.

```
def invertTree(root):
  if not root: return None
  root.left, root.right = invertTree(root.right),
invertTree(root.left)
  return root
```

```
def invertTree(root):
  if not root: return None
  s = [root]
  while s:
    cur = s.pop()
    cur.left, cur.right = cur.right, cur.left
    if cur.left: s.append(cur.left)
    if cur.right: s.append(cur.right)
  return root
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

20. [**Count Value**] Given the root of a binary tree and a target value, write a function to return the number of times the target occurs in the tree.

Iterative Approach (DFS)

```
def countValue(root, target):
  if not root: return 0
  count = 0
  s = [root]
  while s:
    cur = s.pop()
    if cur.val == target: count += 1
    if cur.left: s.append(cur.left)
    if cur.right: s.append(cur.right)
  return count
```

Recursive Approach (DFS)

```
def countValue(root, target):
  if not root: return 0
  match = 1 if root.val == target else 0
  return match + countValue(root.left, target) +
countValue(root.right, target)
```

Iterative Approach (BFS)

```
def countValue(root, target):
  from collections import deque
  if not root: return 0
  count = 0
  q = deque([root])
  while q:
    cur = q.popleft()
    if cur.val == target: count += 1
    if cur.left: q.append(cur.left)
    if cur.right: q.append(cur.right)
  return count
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

21. [**Tree Height**] Given the root of a binary, write a function to return a number representing the height of the tree. An empty tree should return `-1` and a singleton tree should return `0`.

```
def treeHeight(root):
  if not root: return -1
  return 1 + max(treeHeight(root.left), treeHeight(root.right))
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

22. [**Tree Diameter**] Given the root of a binary, write a function to return the length of the diameter of the tree. The diameter is the length of the longest path between any two nodes measured by the number of edges.

```
def dfs(root):
  if not root: return (0, 0)
  left_height, left_diameter = dfs(root.left)
  right_height, right_diameter = dfs(root.right)

  height = 1 + max(left_height, right_height)
  diameter = max(left_diameter, right_diameter, left_height +
right_height)
  return (height, diameter)

def diameterOfTree(root):
  height, diameter = dfs(root)
  return diameter
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

23. [**All Paths**] Given the root of a binary tree, write a function to return a 2-D list containing all possible root-to-leaf paths in correct order.

```
def allPaths(root):
  if not root: return []
  if not root.left and not root.right: return [[root.val]]

  paths = []
  left_paths = allPaths(root.left)
  for sub_path in left_paths:
    paths.append([root.val, *sub_path])

  right_paths = allPaths(root.right)
  for sub_path in right_paths:
    paths.append([root.val, *sub_path])

  return paths
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

24. [**Binary Tree Paths**] Given the root of a binary tree, write a function to return a 2-D list containing all possible root-to-leaf paths in correct order as a string.

```
def binaryTreePaths(root):
  if not root: return []
  if not root.left and not root.right: return [str(root.val)]

  paths = []
  left_paths = binaryTreePaths(root.left)
  for sub_path in left_paths:
    paths.append(f"{root.val}->{sub_path}")

  right_paths = binaryTreePaths(root.right)
  for sub_path in right_paths:
    paths.append(f"{root.val}->{sub_path}")

  return paths
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

25. [**Tree Levels**] Given the root of a binary tree, write a function to return a 2-D list where each sublist is a level of the tree.

```
def treeLevels(root):
  if not root: return []
  levels = []
  s = [(root, 0)]
  while s:
    (cur, cur_level) = s.pop()
    if len(levels) == cur_level: levels.append([cur.val])
    else: levels[cur_level].append(cur.val)
    if cur.right: s.append((cur.right, cur_level+1))
    if cur.left: s.append((cur.left, cur_level+1))
  return levels
```

```
def treeLevels(root):
  def fillLevels(root, levels, cur_level):
    if not root: return None
    if len(levels) == cur_level: levels.append([root.val])
    else: levels[cur_level].append(root.val)
    fillLevels(root.left, levels, cur_level+1)
    fillLevels(root.right, levels, cur_level+1)
  levels = []
  fillLevels(root, levels, 0)
  return levels
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

26. [**Tree Levels II**] Given the root of a binary tree, write a function to return a dictionary/ hash map where each list in the dictionary is a level of the tree.

```
def treeLevels(root):
  if not root: return {}
  levels = {}
  s = [(root, 0)]
  while s:
    (cur, cur_level) = s.pop()
    if len(levels) == cur_level: levels[cur_level] = [cur.val]
    else: levels[cur_level].append(cur.val)
    if cur.right: s.append((cur.right, cur_level+1))
    if cur.left: s.append((cur.left, cur_level+1))
  return levels
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

27. [**Level Averages**] Given the root of a binary tree, write a function to return a list containing the average of each level of the tree.

```
def levelAverages(root):
  def fillLevels(root, levels, cur_level):
    if not root: return None
    if len(levels) == cur_level: levels.append([root.val])
    else: levels[cur_level].append(root.val)
    fillLevels(root.left, levels, cur_level+1)
    fillLevels(root.right, levels, cur_level+1)
  levels = []
  fillLevels(root, levels, 0)
  return [sum(level)/len(level) for level in levels]
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

28. [**Lefty Nodes**] Given the root of a binary tree, write a function to return a list containing all the leftmost nodes on every level of the tree.

```
def leftyNodes(root):
  levels = treeLevels(root)
  return [level[-1] for level in levels]

def treeLevels(root):
  def fillLevels(root, levels, cur_level):
    if not root: return None
    if len(levels) == cur_level: levels.append([root.val])
    else: levels[cur_level].append(root.val)
    fillLevels(root.left, levels, cur_level+1)
    fillLevels(root.right, levels, cur_level+1)
  levels = []
  fillLevels(root, levels, 0)
  return levels
```

```
def leftyNodes(root):
  values = []
  dft(root, 0, values)
  return values

def dft(root, level, values):
  if not root: return None
  if len(values) == level: values.append(root.val)
  dft(root.left, level+1, values)
  dft(root.right, level+1, values)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

29. [**Leaf Lists**] Given the root of a binary tree, write a function to return a list containing the values of all leaf nodes in left-to-right order.

```
def leafList(root):
  if not root: return []
  leafs = []
  s = [root]
  while s:
    cur = s.pop()
    if not cur.right and not cur.left: leafs.append(cur.val)
    if cur.right: s.append(cur.right)
    if cur.left: s.append(cur.left)
  return leafs
```

```
def leafList(root):
  if not root: return []
  if not root.left and not root.right: return [root.val]
  return [*leafList(root.left), *leafList(root.right)]
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

Binary Search Trees

1. [**Add Node**] [**leetcode 701**] Given the root of a binary search tree and a value to insert, write a function to insert the node to the BST and return the root node of the BST.

```
def addNode(root, value):
  if not root: return Node(value)
  if root.val < value: root.right = addNode(root.right, value)
  elif root.val > value: root.left = addNode(root.left, value)
  return root
```

```
n is the number of nodes
Time: O(log(n))
Space: O(1)
```

2. [**Delete Node**] [**leetcode 450**] Given the root of a binary search tree and a value to delete, write a function to delete the node of the BST and return the root node.

```
def findMinNode(root):
 cur = root
 while cur.left:
   cur = cur.left
 return cur

def deleteNode(root, value):
 if not root: return None
 if root.val > value: root.left = deleteNode(root.left, value)
 elif root.val < value: root.right = deleteNode(root.right, value)
 else:
   if not root.left:
     temp = root.right
```

```
      root = None
      return temp
    else not root.right:
      temp = root.left
      root = None
      return temp
    else:
      temp = findMinNode(root.right)
      root.val = temp.val
      root.right = deleteNode(root.right, temp.val)
  return root
```

```
n is the number of nodes
Time: O(log(n))
Space: O(1)
```

3. [**Lowest Common Ancestor**] Given the root of a binary search tree and two nodes, write a function to return the lowest common ancestor of both nodes.

```
def lowestCommonAncestor(root, p, q):
  if p.val < root.val and q.val < root.val: return
lowestCommonAncestor(root.left, p, q)
  elif p.val > root.val and q.val > root.val: return
lowestCommonAncestor(root.right, p, q)
  else: return root
```

```
def lowestCommonAncestor(root, p, q):
 while (root.val - p.val) * (root.val - q.val) > 0:
   root = (root.left, root.right)[p.val > root.val]
 return root
```

```
n is the number of nodes
Time: O(log(n))
Space: O(1)
```

4. [**Validate Tree**] Given the root of a binary search tree, write a function to return a boolean representing if it's a valid binary search tree (BST).

```
def isValidBST(root):
 def validate(node, left, right):
   if not node: return True
   if not (left < node.val < right): return False
```

```
    return (validate(node.left, left, node.val) and
validate(node.right, node.val, right))
 return valid(root, float("-inf"), float("inf"))
```

```
n is the number of nodes
Time: O(log(n))
Space: O(n)
```

```
def isValidBST(root):
 if not root: return True
 s = [(root, float("-inf"), float("inf"))]
 while s:
   cur, left, right = s.pop()
   if not (left < cur.val < right): return False
   if cur.left: s.append((cur.left, left, cur.val))
   if cur.right: s.append((cur.right, cur.val, right))
 return True
```

```
n is the number of nodes
Time: O(log(n))
Space: O(1)
```

Graphs

1. [**Build Graph**] Given a graph as an edge list, write a function to convert/ build it into an adjacency list.

```
  '''
     edges list = [
       ['i','j'],
       ['k','i'],
       ['m','k'],
       ['k','l'],
       ['o','n']
             ]

     adjancecy list = {
       'i': ['j', 'k'],
       'j': ['i'],
       'k': ['i', 'm', 'l'],
       'm': ['k'],
       'l': ['k'],
       'o': ['n'],
       'n': ['o']
```

```
      }
  '''
```

```
def buildGraph(edges):
 graph = {}
 for edge in edges:
   (a,b) = edge
   if a not in graph: graph[a] = []
   if b not in graph: graph[b] = []
   graph[a].append(b)
   graph[b].append(a)
 return graph
```

```
  n is the number of nodes.
  Time: O(n)
  Space: O(n)
```

2. [**Depth First Values**] Given a graph as a dictionary/adjacency list and a source node a, write a function to print all values in the graph in a depth first search (DFS) manner.

```
def depthFirstTraversal(graph, src):
 s = [src]
 while s:
   cur = s.pop()
   print(cur, end=", ")
   for n in graph[cur]:
     s.append(n)
```

```
def depthFirstTraversal(graph, src):
 print(cur, end=", ")
 for n in graph[src]:
   depthFirstTraversal(graph, n)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

3. [**Breadth First Values**] Given a graph as a dictionary/adjacency list and a source node a, write a function to print all values in the graph in a breadth first search (BFS) manner.

```
def breadthFirstTraversal(graph, src):
 q = [src]
 while q:
   cur = q.pop(0)
   print(cur, end=", ")
   for n in graph[cur]:
     q.append(n)
```

```
n is the number of nodes
Time: O(n^2)
Space: O(n)
```

```
def breadthFirstTraversal(graph, src):
 from collections import deque
 q = deque([src])
 while q:
   cur = q.popleft()
   print(cur, end=", ")
   for n in graph[cur]:
     q.append(n)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

4. [**Has Path**] Given a directed acyclic graph (DAG) as a dictionary/adjacency list, a source node `src`, and a destination node `dst`. Write a function that returns a boolean indicating whether or not there is a path from `src` to `dst`.

```
def hasPath(graph, src, dst):
 s = [src]
 while s:
   cur = s.pop()
   if cur == dst: return True
   for n in graph[cur]: s.append(n)
   return False
```

```
def hasPath(graph, src, dst):
 if src == dst: return True
 for n in graph[src]:
```

```
    if hasPath(graph, n, dst): return True
  return False
```

```
def hasPath(graph, src, dst):
  q = [src]
  while q:
    cur = q.pop(0)
    if cur == dst: return True
    for n in graph[cur]: q.append(n)
    return False
```

```
def breadthFirstTraversal(graph, src, dst):
  from collections import deque
  q = deque([src])
  while q:
    cur = q.popleft()
    if cur == dst: return True
    for n in graph[cur]: q.append(n)
    return False
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

5. [**Has Path**] [**Leetcode 1971**] Given a bi-directional possibly cyclic graph as a an edge list, a source node `src`, and a destination node `dst`. Write a function that returns a boolean indicating whether or not there is a path from `src` to `dst`.

```
def buildGraph(edges):
  graph = {}
  for (a,b) in edges:
    if a not in graph: graph[a] = []
    if b not in graph: graph[b] = []
    graph[a].append(b)
    graph[b].append(a)
  return graph

def hasPath(graph, src, dst, visited=set()):
  s, visited = [src], set()
  while s:
    cur = s.pop()
    visited.add(cur)
    if cur == dst: return True
    for n in graph[cur]:
```

```
      if n not in visited:
        s.append(n)
 return False

def undirectedHasPath(edges, nodeA, nodeB):
 graph = buildGraph(edges)
 return hasPath(graph, nodeA, nodeB)
```

```
def buildGraph(edges):
 graph = {}
 for (a,b) in edges:
   if a not in graph: graph[a] = []
   if b not in graph: graph[b] = []
   graph[a].append(b)
   graph[b].append(a)
 return graph

def hasPath(graph, src, dst, visited=set()):
 if src in visited: return False
 if src == dst: return True
 visited.add(src)
 for n in graph[src]:
   if hasPath(graph, n, dst, visited): return True
 return False

def undirectedHasPath(edges, nodeA, nodeB):
 graph = buildGraph(edges)
 return hasPath(graph, nodeA, nodeB)
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

6. [**Connected Components**] Given an undirected graph as a dictionary/adjacency list, write a function that returns the number of connected components within the graph.

```
def explore(graph, node, visited=set()):
 if node in visited: return False
 visited.add(node)
 for n in graph[node]:
   explore(graph, n, visited)
 return True

def connectedComponents(graph):
 count = 0
 for n in graph:
   if explore(graph, n):
```

```
      count += 1
 return count
```

```
def explore(graph, node, visited=set()):
 if node in visited: return 0
 visited.add(node)
 for n in graph[node]:
   explore(graph, n, visited)
 return 1

def connectedComponents(graph):
 count = 0
 for n in graph:
   count += explore(graph, n)
 return count
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

7. [**Connected Components**] [Leetcode 323] Given an undirected graph as an edge list, write a function that returns the number of connected components within the graph.

```
def buildGraph(edges):
 graph = {}
 for n in range(0, n): graph[n] = []
 for (a,b) in edges:
   graph[a].append(b)
   graph[b].append(a)
 return graph

def explore(graph, node, visited=set()):
 if node in visited: return 0
 visited.add(node)
 for n in graph[node]:
   explore(graph, n, visited)
 return 1

def connectedComponents(edges):
 graph = buildGraph(edges)
 count = 0
 for n in graph:
   count += explore(graph, n)
 return count
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

8. [**Largest Component**] Given an undirected graph as a dictionary/adjacency list, write a function that returns the size of the largest connected component within the graph.

```
def explore(graph, node, visited=set()):
 if node in visited: return 0
 visited.add(node)
 size = 1
 for n in graph[node]:
   size += explore(graph, n, visited)
 return size

def largestComponent(graph):
 largest = 0
 for n in graph:
   size = explore(graph, n)
   largest = max(largest, size)
 return largest
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

9. [**Smallest Component**] Given an undirected graph as a dictionary/adjacency list, write a function that returns the size of the smallest connected component within the graph.

```
def explore(graph, node, visited=set()):
 if node in visited: return 0
 visited.add(node)
 size = 1
 for n in graph[node]:
   size += explore(graph, n, visited)
 return size

def smallestComponent(graph):
 smallest = float("inf")
 for n in graph:
   size = explore(graph, n)
   if size: smallest = min(smallest, size)
 return smallest
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

10. [**Shortest Path**] Given an undirected graph as an edge list and two nodes `(nodeA, nodeB)`, write a function that returns the length of the shortest path between those nodes. if there's no path, return `-1`.

```
def buildGraph(edges):
  graph = {}
  for (a,b) in edges:
    if a not in graph: graph[a] = []
    if b not in graph: graph[b] = []
    graph[a].append(b)
    graph[b].append(a)
  return graph

def shortestPath(edges, nodeA, nodeB):
  graph = buildGraph(edges)
  q = [(nodeA, 0)]
  visited = set(nodeA)
  while q:
    cur, dist = q.pop(0)
    if cur == nodeB: return dist
    for n in graph[cur]:
      if n not in visited:
        visited.add(n)
        q.append((n, dist+1))
  return -1
```

```
def buildGraph(edges):
  graph = {}
  for (a,b) in edges:
    if a not in graph: graph[a] = []
    if b not in graph: graph[b] = []
    graph[a].append(b)
    graph[b].append(a)
  return graph

def shortestPath(edges, nodeA, nodeB):
  graph = buildGraph(edges)
  s = [(nodeA, 0)]
  visited = set(nodeA)
  min_dist = float("inf")
  while s:
    cur, dist = s.pop()
    if cur == nodeB:
      min_dist = min(min_dist, dist)
```

```
      return min_dist
    for n in graph[cur]:
      if n not in visited:
        visited.add(n)
        s.append((n, dist+1))
  return -1
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

11. [**Number of Islands**] [Leetcode 200] Given a grid containing Ws and Ls where W represents water and L represents land. Write a function to return the number of islands on the grid.

```
def explore(grid, r, c, visited=set()):
  rowInbounds = 0 <= r < len(grid)
  colInbounds = 0 <= c < len(grid[r])
  if (not rowInbounds or not colInbounds): return False

  if grid[r][c] == 'W': return False

  pos = str(r) + ',' + str(c)
  if pos in visited: return False
  visited.add(pos)

  explore(grid, r-1, c, visited)
  explore(grid, r+1, c, visited)
  explore(grid, r, c-1, visited)
  explore(grid, r, c+1, visited)
  return True

def islandCount(grid):
  count = 0
  for r in range(len(grid)):
    for c in range(len(grid[0])):
      if explore(grid, r, c):
        count += 1
  return count
```

```
def explore(grid, r, c):
  rowInbounds = 0 <= r < len(grid)
  colInbounds = 0 <= c < len(grid[r])
  if (not rowInbounds or not colInbounds or grid[r][c]=='W'): return 0

  grid[r][c] = 'W'

  explore(grid, r-1, c)
```

```
  explore(grid, r+1, c)
  explore(grid, r, c-1)
  explore(grid, r, c+1)
  return 1

def islandCount(grid):
  count = 0
  for r in range(len(grid)):
    for c in range(len(grid[r])):
      if grid[r][c] == 'L':
        count += explore(grid, r, c)
  return count
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

12. [**Max Area of Island**] [Leetcode 695] Given a grid containing Ws and Ls where W represents water and L represents land. Write a function to return the size of the largest island.

```
def explore(grid, r, c, visited=set()):
  rowInbounds = 0 <= r < len(grid)
  colInbounds = 0 <= c < len(grid[0])
  if (not rowInbounds or not colInbounds or grid[r][c] == 'W'): return
0

  pos = (r,c)
  if pos in visited: return 0
  visited.add(pos)

  size = 1
  size += explore(grid, r-1, c, visited)
  size += explore(grid, r+1, c, visited)
  size += explore(grid, r, c-1, visited)
  size += explore(grid, r, c+1, visited)
  return size

def maxIsland(grid):
  max_size = 0
  for r in range(len(grid)):
    for c in range(len(grid[0])):
      size = explore(grid, r, c)
      max_size = max(max_size, size)
  return max_size
```

```
def explore(grid, r, c):
  rowInbounds = 0 <= r < len(grid)
```

```
  colInbounds = 0 <= c < len(grid[0])
  if (not rowInbounds or not colInbounds or grid[r][c] == 'W'): return
0

  grid[r][c] = 'W'

  size = 1
  size += explore(grid, r-1, c)
  size += explore(grid, r+1, c)
  size += explore(grid, r, c-1)
  size += explore(grid, r, c+1)
  return size

def maxIsland(grid):
  max_size = 0
  for r in range(len(grid)):
    for c in range(len(grid[0])):
      if grid[r][c] == 'L':
        max_size = max(max_size, explore(grid, r, c))
  return max_size
```

```
n is the number of nodes
Time: O(n)
Space: O(n)
```

13. [**Min Area of Island**] Given a grid containing Ws and Ls where W represents water and L represents land. Write a function to return the size of the smallest island.

```
def explore(grid, r, c, visited=set()):
  rowInbounds = 0 <= r < len(grid)
  colInbounds = 0 <= c < len(grid[0])
  if (not rowInbounds or not colInbounds or grid[r][c] == 'W'): return
0

  pos = (r,c)
  if pos in visited: return 0
  visited.add(pos)

  size = 1
  size += explore(grid, r-1, c, visited)
  size += explore(grid, r+1, c, visited)
  size += explore(grid, r, c-1, visited)
  size += explore(grid, r, c+1, visited)
  return size

def minIsland(grid):
  min_size = float('inf')
  for r in range(len(grid)):
    for c in range(len(grid[0])):
```

```
        size = explore(grid, r, c)
        if min_size > 0: min_size = min(min_size, size)
    return min_size
```

```
def explore(grid, r, c):
    rowInbounds = 0 <= r < len(grid)
    colInbounds = 0 <= c < len(grid[0])
    if (not rowInbounds or not colInbounds or grid[r][c] == 'W'): return
0

    grid[r][c] = 'W'

    size = 1
    size += explore(grid, r-1, c)
    size += explore(grid, r+1, c)
    size += explore(grid, r, c-1)
    size += explore(grid, r, c+1)
    return size

def minIsland(grid):
    min_size = float('inf')
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == 'L':
                min_size = min(min_size, explore(grid, r, c))
    return min_size
```

```
r is the number of rows in the grid
c is the number of cols in the grid
Time: O(r*c)
Space: O(1)
```

14. [**Closest Carrot**] Given a grid where `O`s are open spaces, `X`s are walls and `C`s are carrots and a starting row and column. Write a function to return the length of the shortest path from the starting point to the carrot.

```
def closestCarrot(grid, starting_row, starting_col):
    visited = set((starting_row, starting_col))
    q = [(starting_row, starting_col, 0)]
    while q:
        (r, c, dist) = q.pop(0)
        if grid[r][c] == 'C': return dist

        deltas = [(1,0), (-1,0), (0,1), (0,-1)]
        for delta_row, delta_col in deltas:
            neig_row = r + delta_row
            neig_col = c + delta_col
```

```
      row_inbounds = 0 <= neig_row < len(grid)
      col_inbounds = 0 <= neig_col < len(grid[0])

      pos = (neig_row, neig_col)
      if row_inbounds and col_inbounds and grid[neig_row, neig_col] !=
'X' and pos not in visited:
        visited.add(pos)
        q.append((neig_row, neig_col, dist+1))
  return -1
```

```
r is the number of rows in the grid
c is the number of cols in the grid
Time: O(r*c)
Space: O(r*c)
```

15. [**Best Bridge**] Given a grid where `W` is water and `L` is land, with exactly two islands, write a function to return the minimum length bridge needed to connect both islands.

```
def bestBridge(grid):
  mainIsland = None
  for r in range(len(grid)):
    for c in range(len(grid[0])):
      potentialIsland = explore(grid, r, c, set())
      if len(potentialIsland) > 0:
        mainIsland = potentialIsland
        break

  visited = set(mainIsland)
  q = []
  for r,c in mainIsland:
    q.append((r,c,0))
  while q:
    r,c,dist = q.pop(0)
    if grid[r][c] == 'L' and (r,c) not in mainIsland:
      return dist - 1
    deltas = [(1,0), (-1,0), (0,1), (0,-1)]
    for delta_row, delta_col in deltas:
      neig_row, neig_col = r+delta_row, c+delta_col
      neig_pos = (neig_row, neig_col)
      if isInbounds(grid, neig_row, neig_col) and neig_pos not in
visited:
        visited.add(neig_pos)
        q.append((neig_row, neig_col, dist+1))

def isInbounds(grid, r, c):
  rowInbounds = 0 <= r < len(grid)
  colInbounds = 0 <= c < len(grid[0])
  return rowInbounds and colInbounds
```

```
def explore(grid, r, c, visited):
  pos = (r,c)
  if not isInbounds(grid, r, c) or grid[r][c] == 'W' or pos in
visited: return visited
  visited.add(pos)

  explore(grid, r-1, c, visited)
  explore(grid, r+1, c, visited)
  explore(grid, r, c-1, visited)
  explore(grid, r, c+1, visited)
  return visited
```

```
r is the number of rows in the grid
c is the number of cols in the grid
Time: O(r*c)
Space: O(r*c)
```

16. [**Longest Path**] Given a DAG as an adjacency list, write a function to return the length of the longest path within the graph.

```
def longestPath(graph):
  dist = {}
  for n in graph:
    if len(graph[n]) == 0: dist[n] = 0
  for n in graph:
    dist[n] = explore(graph, n, dist)
  return max(dist.values())

def explore(graph, node, dist):
  if node in dist: return dist[node]
  max_length = 0
  for n in graph[node]:
    length = explore(graph, n, dist)
    max_length = max(max_length, length)
  return 1 + max_length
```

```
def longestPath(graph):
  dist = {}
  for n in graph:
    if len(graph[n]) == 0: dist[n] = 0
  for n in graph:
    explore(graph, n, dist)
  return max(dist.values())

def explore(graph, node, dist):
  if node in dist: return dist[node]
```

```
  max_length = 0
  for n in graph[node]:
    length = explore(graph, n, dist)
    max_length = max(max_length, length)
  dist[node] = 1 + max_length
  return dist[node]
```

```
n is the number of nodes in the graph
Time: O(n)
Space: O(n)
```

17. [**Semesters Required**] Given a number of courses n, and a list of prerequisites, write a function to return the minimum number of semesters required to take all n courses.

```
def semesterRequired(num_courses, prereqs):
  graph = buildGraph(num_courses, prereqs)
  dist = {}
  for n in graph:
    if len(graph[n]) == 0:
      dist[n] = 1
  for n in graph:
    dist[n] = explore(graph, n, dist)
  return max(dist.values())

def buildGraph(num_courses, prereqs):
  graph = {n:[] for n in range(num_courses)}
  for a,b in prereqs:
    graph[a].append(b)
  return graph

def explore(graph, node, dist):
  if node in dist: return dist[node]
  max_length = 0
  for n in graph[node]:
    length = explore(graph, n, dist)
    max_length = max(max_length, length)
  return 1 + max_length
```

```
n is the number of nodes in the graph
Time: O(n)
Space: O(n)
```

18. [**Has Cycle**] Given a directed graph as an object/ asjacency list, write a function to return a boolean indicating whether or not the graph has cycles.

```
def hasCycle(graph):
  visiting = set()
  visited = set()

  for node in graph:
    if detectCycle(graph, node, visited, visiting): return True
  return False

def detectCycle(graph, node, visited, visiting):
  if node in visited: return False
  if node in visiting: return True
  visiting.add(node)

  for n in graph[node]:
    if detectCycle(graph, n, visited, visiting): return True
  visiting.remove(node)
  visited.add(node)
  return False
```

```
n is the number of nodes in the graph
Time: O(n^2)
Space: O(n)
```

19. [**Prereqs Possible**] Given a number of courses `n` (0 - n-1) and and edge list of prereqs as arguments, write a function to return a boolean indicating whether or not it is possible to complete all courses.

```
def prereqsPossibel(num_courses, prereqs):
  visiting = set()
  visited = set()
  graph = buildGraph(num_courses, prereqs)
  for node in graph:
    if detectCycle(graph, node, visited, visiting): return False
  return True

def detectCycle(graph, node, visited, visiting):
  if node in visited: return False
  if node in visiting: return True
  visiting.add(node)

  for n in graph[node]:
    if detectCycle(graph, n, visited, visiting): return True
  visiting.remove(node)
  visited.add(node)
  return False

def buildGraph(num_courses, prereqs):
  graph = {n:[] for n in range(num_courses)}
  for a,b in prereqs:
```

```
    graph[a].append(b)
  return graph
```

```
n is the number of nodes in the graph
Time: O(n^2)
Space: O(n)
```

20. [**Knight Moves**] Given a knight and a pawn on a chess board, wite a function to return the minimum possible number of moves the knight can travel to get to the pawn's position.

```
def knight_attack(n, kr, kc, pr, pc):
  visited = set([(kr,kc)])
  q = [(kr,kc,0)]
  while q:
    r,c,step = q.pop(0)
    if (r,c) == (pr,pc):
      return step
    possible_moves = getValidMoves(n, r, c)
    for pos in possible_moves:
      pos_r, pos_c = pos
      if pos not in visited:
        q.append((pos_r, pos_c, step+1))
        visited.add(pos)
  return None

def getValidMoves(n, r, c):
  positions = [
    (r+2, c+1),(r-2, c+1),(r+2, c-1),(r-2, c-1),
    (r+1, c+2),(r-1, c+2),(r+1, c-2),(r-1, c-2),
  ]
  inbound_positions = []
  for pos in positions:
    new_r, new_c = pos
    if 0 <= new_r < n and 0 <= new_c < n:
      inbound_positions.append(pos)
  return inbound_positions
```

```
n is the number of nodes in the graph
Time: O(n^2)
Space: O(n^2)
```

Dynamic Programming

1. [**Factorial Trailing Zeros**] [Leetcode 509] Given an integer `n`, return the number of trailing zeros in `n!`.

```
def trailingZeros(n):
 count = 0
 while n > 0:
   n //= 5
   count += n
 return count
```

```
def trailingZeros(n):
 if n == 0: return 0
 return n//5 + trailingZeros(n//5)
```

```
def trailingZeros(n):
 return 0 if n == 0 else n//5 + trailingZeros(n//5)
```

```
  n is the length of the string
  Time: O(n)
  Space: O(n)
```

2. [**Fibonacci**] [Leetcode 509] Given a number n, writhe a function to return the n-th number in the Fibonacci sequence.

```
def fib(n, memo={}):
 if n in memo: return memo[n]
 if n <= 2: return 1
 memo[n] = fib(n-1, memo) + fib(n-2, memo)
 return memo[n]
```

```
def fib(n):
 if n < 1: return 0
 dp = [0]*(n+1)
 dp[1] = 1
 for i in range (2, n+1):
   dp[i] = dp[i-1] + dp[i-2]
 return dp[-1]
```

```

```

```
  n is the length of the string
  Time: O(n)
  Space: O(n)
```

```
def fib(n):
 if n < 2: return n
 one, two = 0, 1
 for i in range (2, n+1):
   one, two = two, one + two
 return two
```

```
  n is the length of the string
  Time: O(n)
  Space: O(1)
```

3. [**Climbing Stairs**] [Leetcode 70]] Given a number n, write a function to return the n-th number in the Fibonacci sequence.

```
def climbingStairs(n, memo={}):
 if n in memo: return memo[n]
 if n <= 2: return n
 memo[n] = climbingStairs(n-1, memo) + climbingStairs(n-2, memo)
 return memo[n]
```

```
def climbingStairs(n):
 if n < 1: return 0
 dp = [0]*(n+1)
 dp[0], dp[1] = 1, 1
 for i in range (2, n+1):
   dp[i] = dp[i-1] + dp[i-2]
 return dp[-1]
```

```
  n is the length of the string
  Time: O(n)
  Space: O(n)
```

```
def climbingStairs(n):
 one, two = 1, 1
 for i in range (2, n+1):
```

```
    one, two = two, one + two
  return two
```

```
  n is the length of the string
  Time: O(n)
  Space: O(1)
```

4. [**Tribonacci**] Given a number n, write a function to return the n-th number in the Tribonacci sequence.

```
def trib(n, memo={}):
 if n in memo: return memo[n]
 if n <= 1: return 0
 if n == 2: return 1
 memo[n] = trib(n-1, memo) + trib(n-2, memo) + trib(n-3, memo)
 return memo[n]
```

```
def trib(n):
 if n < 2: return 0
 dp = [0]*(n+1)
 dp[2] = 1
 for i in range (3, n+1):
   dp[i] = dp[i-1] + dp[i-2] + dp[i-3]
 return dp[-1]
```

```
  n is the length of the string
  Time: O(n)
  Space: O(n)
```

```
def trib(n):
 if n < 2: return 0
 one, two, three = 0, 0, 1
 for i in range (3, n+1):
   one, two, three = two, three, (one + two + three)
 return three
```

```
  n is the length of the string
  Time: O(n)
  Space: O(1)
```

5. [**Non Adjcent Sum**] Given a list of numbers, write a function to return the maximum sum of non-adjacent items in the list. There is no limit to how many times items from the list can be used.

```
def nonAdjacentSum(nums, memo={}):
 key = ','.join(str(nums))
 if key in memo: return memo[key]
 if not nums: return 0
 include_first = nums[0] + nonAdjacentSum(nums[2:], memo)
 exclude_first = nonAdjacentSum(nums[1:], memo)
 memo[key] = max(include_first, exclude_first)
 return memo[key]
```

```
def nonAdjacentSum(nums):
 return _nonAdjacentSum(nums, 0)

def _nonAdjacentSum(nums, i, memo={}):
 if i in memo: return memo[i]
 if i >= len(nums): return 0
 include_first = nums[i] + _nonAdjacentSum(nums, i+2, memo)
 exclude_first = _nonAdjacentSum(nums, i+1, memo)
 memo[i] = max(include_first, exclude_first)
 return memo[i]
```

```
def nonAdjacentSum(nums):
 if not nums: return 0
 if len(nums) < 3: return max(nums)
 dp = [0]*len(nums)
 dp[0], dp[1] = nums[0], max(nums[0], nums[1])
 for i in range(2, len(dp)):
   dp[i] = max(nums[i] + dp[i-2], dp[i-1])
 return dp[-1]
```

```
def nonAdjacentSum(nums):
 r1, r2 = 0, 0
 for num in nums:
   r1, r2 = r2, max(r1+num, r2)
 return r2
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

6. [**House Robber**] [Leetcode 198]] Given a list of numbers representing houses with money in them, write a function to return the maximum amount of money you can get by robbing non-adjacent houses.

```
def rob(nums, memo={}):
 key = ','.join(str(nums))
 if key in memo: return memo[key]
 if not nums: return 0
 include_first = nums[0] + rob(nums[2:], memo)
 exclude_first = rob(nums[1:], memo)
 memo[key] = max(include_first, exclude_first)
 return memo[key]
```

```
def rob(nums):
 def helper(nums, i, memo={}):
   if i in memo: return memo[i]
   if i >= len(nums): return 0
   include_first = nums[i] + helper(nums, i+2, memo)
   exclude_first = rob(nums, i+1, memo)
   memo[i] = max(include_first, exclude_first)
   return memo[i]
 return helper(nums, 0)
```

```
def rob(nums):
 if not nums: return 0
 if len(nums) < 3: return max(nums)
 dp = [0]*len(nums)
 dp[0], dp[1] = nums[0], max(nums[0], nums[1])
 for i in range(2, len(dp)):
   dp[i] = max(nums[i] + dp[i-2], dp[i-1])
 return dp[-1]
```

```
def rob(nums):
 r1, r2 = 0, 0
 for num in nums:
   r1, r2 = r2, max(r1+num, r2)
 return r2
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

7. [**House Robber II**] [Leetcode 213]] Given a circular list of numbers representing houses with money in them, write a function to return the maximum amount of money you can get by robbing non-adjacent houses (cant rob both first and last houses as they are adjacent in a circular street).

```
def rob(nums):
 def helper(nums):
   r1, r2 = 0, 0
   for num in nums:
     r1, r2 = r2, max(r1+num, r2)
   return r2
 return max(helper(nums[:-1]), helper(nums[1:]))
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

8. [**House Robber III**] [Leetcode 337]] Given a binary tree representing houses with money in them, write a function to return the maximum amount of money you can get by robbing non-adjacent houses (houses with direct link).

```
@lru_cache
def rob(root, canRob=True):
 if not root: return 0
 include_root = root.val + rob(root.left, False) + rob(root.right,
False) if canRob else 0
 exclude_root = rob(root.left, True) + rob(root.right, True)
 return max(include_root, exclude_root)
```

```
def rob(root):
 if root in memo: return memo[root]
 if not root: return 0
 include_root, exclude_root = root.val, rob(root.left) +
rob(root.right)
 if root.left: include_root += rob(root.left.left) +
rob(root.left.right)
 if root.right: include_root += rob(root.right.left) +
rob(root.right.right)
 memo[root] = max(include_root, exclude_root)
 return memo[root]
```

```
  a is the amount given
  n is the length of the array
```

```
    Time: O(a*n)
    Space: O(a)
```

9. [**Unique Paths**] [Leetcode 62]] Given a `m x n` grid and a robot positioned at the top-left corner, write a function to return the number of possible unique paths to get to the bottom-right corner.

```
def gridTravler(m, n, memo={}):
 key = (m,n)
 if key in memo: return memo[key]
 if m==0 or n==0: return 0
 if m==1 or n==1: return 1
 memo[key] = gridTravler(m, n-1, memo) + gridTravler(m-1, n, memo)
 return memo[key]
```

```
def gridTravler(m, n, memo={}):
 grid = [[0 for j in range(n+1)] for i in range(m+1)]
 grid[1][1] = 1
 for r in range(1, m+1):
   for c in range(1, n+1):
     if r==1 and c==1: continue
     grid[r][c] = grid[r-1][c] + grid[r][c-1]
 return grid[-1][-1]
```

```
  m is the number of rows in the grid
  n is the number of columns in the grid
  Time: O(m*n)
  Space: O(n)
```

10. [**Unique Paths II**] [Leetcode 63]] Given a `m x n` integer grid where `1` represents obstacles and `0`represents open spaces; and a robot positioned at the top-left corner, write a function to return the number of possible unique paths to get to the bottom-right corner.

```
def uniquePaths(grid):
 return countPaths(grid, r, c)

def countPaths(grid, r, c, memo={}):
 key = (r,c)
 if key in memo: return memo[key]
 if r >= len(grid) or c >= len(grid[0]) or grid[r][c]=="1": return 0
 if r == len(grid)-1 or c == len(grid[0])-1: return 1
 memo[key] = countPaths(grid, r+1, c, memo) + countPaths(grid, r, c+1,
memo)
 return memo[key]
```

```
  m is the number of rows in the grid
  n is the number of columns in the grid
  Time: O(m*n)
  Space: O(n*n)
```

11. [**Min Path Sum**] [[Leetcode 64]] Given a `m x n` integer grid, write a function the return the minimum sum possible by traveling from the top-left corner to the bottom-right corner. You can only travel by moving down or right.

```
def minPathSum(grid):
 def dfs(grid, r, c, memo={}):
   key = (r,c)
   if key in memo: return memo[key]
   if r >= len(grid) or c >= len(grid[0]): return float('inf')
   if r == len(grid)-1 and c == len(grid[0])-1: return grid[r][c]
   min_path = min(dfs(grid, r+1, c, memo), dfs(grid, r, c+1, memo))
   memo[key] = grid[r][c] + min_path
   return memo[key]
 return dfs(grid, 0, 0)
```

```
  m is the number of rows in the grid
  n is the number of columns in the grid
  Time: O(m*n)
  Space: O(n*n)
```

12. [**Max Path Sum**] Given a `m x n` integer grid, write a function the return the maximum sum possible by traveling from the top-left corner to the bottom-right corner. You can only travel by moving down or right.

```
def maxPathSum(grid):
 def dfs(grid, r, c, memo={}):
   key = (r,c)
   if key in memo: return memo[key]
   if r >= len(grid) or c >= len(grid[0]): return float('-inf')
   if r == len(grid)-1 and c == len(grid[0])-1: return grid[r][c]
   max_path = max(dfs(grid, r+1, c, memo), dfs(grid, r, c+1, memo))
   memo[key] = grid[r][c] + max_path
   return memo[key]
 return dfs(grid, 0, 0)
```

```
  m is the number of rows in the grid
  n is the number of columns in the grid
```

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

13. [**Can Sum**] Given a target amount and a list of positive numbers. Write a function to return a boolean indicating whether or not it's possible to create the amount. You can reuse numbers of the list as needed.

```
def canSum(target, nums, memo={}):
 if target in memo: return memo[target]
 if target == 0: return True
 if target < 0: return False
 for n in nums:
   rem = target - n
   if canSum(rem, nums, memo):
     memo[target] = True
     return memo[target]
 memo[target] = False
 return memo[target]
```

```
def canSum(target, nums):
 dp = [False]*(target+1)
 dp[0] = True
 for i in range(1, target+1):
   for n in nums:
     if i-n >=0: dp[i] = dp[i-n]
 return dp[-1]
```

```
def canSum(target, nums):
 q = [target]
 visited = set([target])
 while q:
   cur_amt = q.pop(0)
   if cur_amt == 0: return True
   for n in nums:
     new_cur_amt = cur_amt - n
     if new_cur_amt not in visited and new_cur_amt >= 0:
       q.append(new_cur_amt)
       visited.add(new_cur_amt)
 return False
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

14. [**Count Sum**] Given a target amount and a list of positive numbers. Write a function to return an integer representing the total number of ways to create the target amount by adding numbers in the list. You can reuse numbers of the list as needed.

```
def countSum(target, nums, memo={}):
 if target in memo: return memo[target]
 if target == 0: return 1
 if target < 0: return 0
 count = 0
 for n in nums:
   rem = target - n
   count += countSum(rem, nums, memo)
 memo[target] = count
 return memo[target]
```

```
def countSum(target, nums):
 dp = [0]*(target+1)
 dp[0] = 1
 for i in range(1, target+1):
   for n in nums:
     if i-n >=0: dp[i] += dp[i-n]
 return dp[-1]
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

15. [**How Sum**] Given a target amount and a list of positive numbers. Write a function to return an array (of any combination) that adds up to exactly the target amount. If no such array exists, then return None.

```
def howSum(target, nums, memo={}):
 if target in memo: return memo[target]
 if target == 0: return []
 if target < 0: return None
 for n in nums:
   rem = target - n
   rem_combo = howSum(rem, nums, memo)
   if rem_combo != None:
     memo[target] = [*rem_combo, n]
     return memo[target]
 memo[target] = None
 return memo[target]
```

```
def howSum(target, nums):
 dp = [None]*(target+1)
 dp[0] = []
 for i in range(1, target+1):
   for n in nums:
     if i-n >= 0 and dp[i-n] != None: dp[i] = [*dp[i-n], n]
 return dp[-1]
```

```
def howSum(target, nums):
 q = [(target, [])]
 visited = set([target])
 while q:
   cur_amt, cur_combo = q.pop(0)
   if cur_amt == 0: return cur_combo
   for n in nums:
     new_cur_amt = cur_amt - n
     if new_cur_amt not in visited and new_cur_amt >= 0:
       q.append((new_cur_amt, [*cur_combo, n]))
       visited.add(new_cur_amt)
 return None
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

16. [**Best Sum**] Given a target amount and a list of positive numbers. Write a function to return the shortest array (of any combination) that adds up to exactly the target amount. If no such array exists, then return None.

```
def bestSum(target, nums, memo={}):
 if target in memo: return memo[target]
 if target == 0: return []
 if target < 0: return None
 shortest_combo = None
 for n in nums:
   rem = target - n
   rem_combo = bestSum(rem, nums, memo)
   if rem_combo != None:
     target_combo = [*rem_combo, n]
     if shortest_combo == None or len(target_combo) < len(shortest_combo):
       shortest_combo = target_combo
 memo[target] = shortest_combo
 return memo[target]
```

```
def bestSum(target, nums):
 dp = [None]*(target+1)
 dp[0] = []
 for i in range(1, target+1):
   shortest_combo = None
   for n in nums:
     if i-n >= 0 and dp[i-n] != None:
       target_combo = [*dp[i-n], n]
       if shortest_combo == None or len(target_combo) <
len(shortest_combo):
         shortest_combo = target_combo
     dp[i] = shortest_combo
 return dp[-1]
```

```
def bestSum(target, nums):
 q = [(target, [])]
 visited = set([target])
 while q:
   cur_amt, cur_combo = q.pop(0)
   if cur_amt == 0: return cur_combo
   for n in nums:
     new_cur_amt = cur_amt - n
     if new_cur_amt not in visited and new_cur_amt >= 0:
       q.append((new_cur_amt, [*cur_combo, n]))
       visited.add(new_cur_amt)
 return None
```

```
 a is the amount given
 n is the length of the array
 Time: O(a^2 *n)
 Space: O(a)
```

17. [**Combination Sum**] [Leetcode 39] Given a target amount and a list of distinct numbers. Write a function to return all the possible unique combinations to make change for the given amount.

```
def combinationSum(amount, nums):
 if amount == 0: return [[]]
 if amount < 0: return None
 all_combo = []
 for n in nums:
   rem = amount-n
   rem_combo = combinationSum(rem, nums)
   if rem_combo != None:
     target_combo = [[n, *way] for way in rem_ways]
```

```
      for item in target_combo:
        item.sort()
        if item not in all_combo: all_combo.append(item)
  return all_combo
```

```
  a is the amount given
  c is the number of coins
  Time: O(a*c)
  Space: O(ac)
```

18. [**Target Sum**] [Leetcode 494] Given a target amount and a list of numbers. Write a function to return the number of different expressions that can be built to add up to the target by adding a - or + in front of each item in the list.

```
def targetSumWays(target, nums):
 def dfs(idx, total, memo={}):
   if (idx, total) in memo: return memo[(idx, total)]
   if idx == len(nums): return 1 if (total==target) else 0
   memo[(idx, total)] = dfs(idx+1, total+nums[idx]) + dfs(idx+1, total-
nums[idx])
   return memo[(idx, total)]
 return dfs(0,0)
```

```
  a is the amount given
  n is the length of the array
  Time: O(a*n)
  Space: O(a)
```

19. [**Coin Change**] [Leetcode 322] Given a target amount and a list of numbers representing coins. Write a function to return the fewest number of coins to create the amount. You can reuse as mayn coins as needed.

```
def coinChange(amount, coins):
 def dfs(amount, coins, memo={}):
   if amount in memo: return memo[amount]
   if amount < 0: return float('inf')
   if amount == 0: return 0
   min_coins = float('inf')
   for c in coins:
     num_coins = 1 + dfs(amount-c, coins)
     min_coins = min(min_coins, num_coins)
   memo[amount] = min_coins
   return min_coins
```

```
 result = dfs(amount, coins)
 return result if result != float('inf') else -1
```

```
def coinChange(amount, coins):
 dp = [float('inf')]*(amount+1)
 dp[0] = 0
 for a in range(1, amount+1):
   for c in coins:
     if a - c >= 0: dp[a] = min(dp[a], 1+dp[a-c])
 return dp[-1] if dp[-1] != float('inf') else -1
```

```
def coinChange(amount, coins):
 q = deque([(amount, 0)])
 visited = set()
 while q:
   cur_amt, num_coins = q.popleft()
   if cur_amt == 0: return num_coins
   for c in coins:
     new_cur_amt = cur_amt - c
     if new_cur_amt not in visited and new_cur_amt >= 0:
       q.append((new_cur_amt, 1+num_coins))
       visited.add(new_cur_amt)
 return -1
```

```
 a is the amount given
 n is the length of the array
 Time: O(a*n)
 Space: O(a)
```

20. [**Coin Change II**] [Leetcode 518] Given a target amount and a list of numbers representing coins. Write a function to return the number of different ways to make change for the given amount.

```
def coinChangeII(amount, coins):
 def dfs(amount, coins, i, memo={}):
   key = (amount,i)
   if key in memo: return memo[key]
   if amount == 0: return 1
   if i >= len(coins): return 0
   total, coin = 0, coins[i]
   for qty in range(0, (amount//coin)+1):
       rem = amount - (qty*coin)
       total += dfs(rem, coins, i+1, memo)
   memo[key] = total
   return total
 return dfs(amount, coins, 0)
```

```
def coinChangeII(amount, coins):
 dp = [0]*(amount+1)
 dp[0] = 1
 for c in coins:
   for a in range(1, amount+1):
     if a - c >= 0:
       dp[a] += dp[a-c]
 return dp[-1]
```

```
 a is the amount given
 c is the number of coins
 Time: O(a*c)
 Space: O(ac)
```

21. [**Coin Change III**] Given a target amount and a list of distinct numbers representing coins. Write a function to return all the possible unique to make change for the given amount.

```
def coinChangeWays(amount, coins):
 if amount == 0: return [[]]
 if amount < 0: return None
 res = []
 for c in coins:
   rem = amount-c
   rem_ways = coinChangeWays(rem, coins)
   if rem_ways != None:
     all_ways = [[c, *way] for way in rem_ways]
     for item in all_ways:
       item.sort()
       if item not in res: res.append(item)
 return res
```

```
 a is the amount given
 c is the number of coins
 Time: O(a*c)
 Space: O(ac)
```

22. [**Word Break**] [**Leetcode 139**] Given a target string and a list of words. Write a function to return a boolean indicating whether or not it's possible to create the target string by concatenating words of the list together. You can reuse words as needed.

```

```

```
def canConstruct(target, word_bank, memo={}):
 if target in memo: return memo[target]
 if target == "": return True
 for w in word_bank:
   if target.startswith(w):
     surfix = target[len(w):]
     if canConstruct(surfix, word_bank, memo):
       memo[target] = True
       return memo[target]
 memo[target] = False
 return memo[target]
```

```
def canConstruct(target, word_bank):
 dp = [False]*(len(target)+1)
 dp[0] = True
 for i in range(0, len(target)+1):
   for w in word_bank:
     if target[i:i+len(w)] == w:
       dp[i + len(w)] = True
 return dp[-1]
```

```
  a is the length of the target string
  n is the length of the word array
  Time: O(a^2 * n)
  Space: O(a)
```

23. [**Count Construct**] Given a target string and a list of words. Write a function to return an integer representing the total number of ways to create the target string by concatenating words of the list together. You can reuse words as needed.

```
def countConstruct(target, word_bank, memo={}):
 if target in memo: return memo[target]
 if target == "": return 1
 count = 0
 for w in word_bank:
   if target.startswith(w):
     surfix = target[len(w):]
     count += countConstruct(surfix, word_bank, memo)
 memo[target] = count
 return memo[target]
```

```
def canConstruct(target, word_bank):
 dp = [0]*(len(target)+1)
 dp[0] = 1
```

```
  for i in range(0, len(target)+1):
    for w in word_bank:
      if target[i: i+len(w)] == w:
        dp[i + len(w)] += dp[i]
  return dp[-1]
```

```
  a is the length of the target string
  n is the length of the word array
  Time: O(a^2 * n)
  Space: O(a)
```

24. [**Best Construct**] Given a target string and a list of words. Write a function to return an integer representing the minimum number of words needed to create the target string by concatenating words of the list together. You can reuse words as needed.

```
def bestConstruct(target, word_bank, memo={}):
  if target in memo: return memo[target]
  if target == "": return 0
  min_words = float('inf')
  for w in word_bank:
    if target.startswith(w):
      surfix = target[len(w):]
      num_words = 1 + bestConstruct(surfix, word_bank, memo)
      min_words = min(min_words, num_words)
  memo[target] = min_words
  return memo[target]
```

```
def bestConstruct(target, word_bank):
  dp = [float('inf')]*(len(target)+1)
  dp[0] = 0
  for i in range(0, len(target)+1):
    for w in word_bank:
      if target[i: i+len(w)] == w:
        dp[i + len(w)] = min(dp[i + len(w)], 1+dp[i])
  return dp[-1]
```

```
  a is the length of the target string
  n is the length of the word array
  Time: O(an)
  Space: O(a)
```

25. [**All Construct**] Given a target string and a list of words. Write a function to return a 2D array containing all the ways to create the target string by concatenating words of the list together. You can

reuse words as needed.

```
def allConstruct(target, word_bank, memo={}):
 if target in memo: return memo[target]
 if target == "": return [[]]
 all_ways = []
 for w in word_bank:
   if target.startswith(w):
     surfix = target[len(w):]
     surfix_ways = allConstruct(surfix, word_bank, memo)
     target_ways = [[word, *way] for way in surfix_ways]
     for item in target_ways:
       all_ways.append(item)
 memo[target] = all_ways
 return memo[target]
```

```
 a is the length of the target string
 n is the length of the word array
 Time: O(a^2 * n)
 Space: O(a)
```

26. [**Word Break II**] [**Leetcode 140**] Given a target string `s` and a list of words `wordDict`. Write a function to add spaces in `s` to construct a sentence where each word is a vaild word in the `wordDict`. You can reuse words as needed.

```
def wordBreak(s, wordDict):
 if s == '': return ['']
 all_ways = []
 for w in wordDict:
   if s.startswith(w):
     surfix = s[len(w):]
     surfix_ways = wordBreak(surfix, wordDict)
     target_ways = [f'{w} {way}' for way in surfix_ways]
     for item in target_ways:
       all_ways.append(item)
 return all_ways
```

```
 a is the length of the target string
 n is the length of the word array
 Time: O(a^2 * n)
 Space: O(a)
```

27. [**Summing Squares**] Given a number `n`, write a function to return the minimum number of perfect squares that sum to the target.

```
def summingSquares(n, memo={}):
 if n in memo: return memo[n]
 if n == 0: return 0
 min_count = float("inf")
 for i in range(1, int(n**(1/2))+1):
   rem = n - i*i
   count = 1 + summingSquares(rem, memo)
   min_count = min(min_count, count)
 memo[n] = min_count
 return memo[n]
```

```
 n is the number given
 Time: O(n * sqrt(n))
 Space: O(n)
```

28. [**Jump Game**] [**Leetcode 55**] Given a list of numbers where each number represents the max number of steps to take, write a function to return a boolean indicating whether or not it is possible to get to the end of the list from the begining.

```
def canJump(nums, i=0, memo={}):
 if i in memo: return memo[i]
 if i >= len(nums)-1: return True
 for step in range(1, nums[i]+1):
   if canJump(nums, i+step, memo):
     memo[i] = True
     return memo[i]
 memo[i] = False
 return memo[i]
```

```
def canJump(nums):
 m = 0
 for i, step in enumerate(nums):
   if i > m: return False
   m = max(m, i+step)
 return True
```

```
 n is the number given
 Time: O(n^2)
 Space: O(n)
```

29. [**Max Palindromic Subsequence**] Given a string `n`, write a function to return the length of the longest subsequence of the string that is also a palindrome.

```
def maxPalinSubsequence(s, memo={}):
 if s in memo: return memo[s]
 if not s: return 0
 if len(s)==1: return 1

 if s[0] == s[-1]: memo[s] = 2 + maxPalinSubsequence(s[1:-1], memo)
 else: memo[s] = max(maxPalinSubsequence(s[0:-1], memo),
maxPalinSubsequence(s[1:], memo))
 return memo[s]
```

```
def maxPalinSubsequence(s):
 def _maxPalinSubsequence(string, l=0, r=len(s)-1, memo={}):
   key = (l,r)
   if key in memo: return memo[key]
   if l > r: return 0
   if l == r: return 1

   if string[l] == string[r]: memo[key] = 2 + _maxPalinSubsequence(string,
l+1, r-1, memo)
   else: memo[key] = max(_maxPalinSubsequence(string, l+1, r, memo),
_maxPalinSubsequence(string, l, r-1, memo))
   return memo[key]
 return _maxPalinSubsequence(s)
```

```
  n is the nlength of the given string
  Time: O(n^2)
  Space: O(n^2)
```

30. [**Max Overlapping Subsequence**] Given two strings `s1` and `s2`, write a function to return the length of the longest overlapping subsequence of the string.

```
def overlap_subsequence(s1, s2, memo={}):
 key = s1+s2
 if key in memo: return memo[key]
 if len(s1) == 0 or len(s2) == 0: return 0
 if s1[0] == s2[0]: memo[key] = 1 + overlap_subsequence(s1[1:], s2[1:])
 else: memo[key] = max(overlap_subsequence(s1, s2[1:]),
overlap_subsequence(s1[1:], s2))
 return memo[key]
```

```
def overlap_subsequence(s1, s2, i=0, j=0, memo={}):
 key = (i,j)
 if key in memo: return memo[key]
```

```
  if i == len(s1) or j == len(s2): return 0
  if s1[i] == s2[j]: memo[key] = 1 + overlap_subsequence(s1, s2, i+1, j+1)
  else: memo[key] = max(overlap_subsequence(s1, s2, i+1, j),
overlap_subsequence(s1, s2, i, j+1))
  return memo[key]
```

```
  n is the nlength of the given string
  Time: O(n^2)
  Space: O(n^2)
```

31. [**Edit Distance**] Given two strings, write a function to compute the edit distance between both strings. Meaning how many changes to be made on one string to make it identical to the other string. Example, the edit distance of the two strings `pale` and `bale` is `1` because replacing the `p` with a `b` makes them equal.

```
def computeEditDistance(s, t):
 memo = {}
 def recurse(i, j):
   if (i,j) in memo:
     return memo[(i,j)]
   if i == 0:
     result = j
   elif j == 0:
     result = i
   elif s[i-1] == t[j-1]:
     result = recurse(i-1, j-1)
   else:
     subCost = 1 + recurse(i-1, j-1)
     delCost = 1 + recurse(i-1, j)
     insCost = 1 + recurse(i, j-1)
     result = min(subCost, delCost, insCost)
   memo[(i,j)] = result
   return result
 return recurse(len(s), len(t))
```

METHOD #2

```
def computeEditDistance2(s, t):
 dp = [[0 for j in range(len(t)+1)] for i in range(len(s)+1)]
 for i in range(len(s)+1):
   for j in range(len(t)+1):
     if i == 0:
       dp[i][j] = j
     elif j == 0:
       dp[i][j] = i
     elif s[i-1] == t[j-1]:
       dp[i][j] = dp[i-1][j-1]
```

```
      else:
        subCost = 1 + dp[i-1][j-1]
        delCost = 1 + dp[i-1][j]
        insCost = 1 + dp[i][j-1]
        dp[i][j] = min(subCost, delCost, insCost)
  return dp[-1][-1]
```

Tries

1. [**Build Trie**] Given a string, implement a Trie data structure capable of storing the string/ word and also searching the tri if a word exists. For example, the words `hello`, `help`, and `hey` can be stored in the tri as below.

```
  '''
   {h: {
     e: {
       l: {
         l: {o: {'*': True}}
         p: {'*': True}
       }
       y: {'*': True}
     }
   }}
  '''
```

```
  class Trie:
    head = {}
    def add(self, word):
      cur = self.head
      for ch in word:
        if ch not in cur: cur[ch] = {}
        cur = cur[ch]
      cur['*'] = True

    def search(self, word):
      cur = self.head
      for ch in word:
        if ch not in cur: return False
        cur = cur[ch]
      if '*' in cur: return True
      else: return False
```

```
  n is the length of the word.
  Time: O(n)
  Space: O(n)
```

2. [**Search Trie**] Given the head of a trie and a string, write a function to return a boolean indicating whether or not the string is present in the trie.

```
def search(self, head, word):
  cur = head
  for ch in word:
    if ch not in cur: return False
    cur = cur[ch]
  if '*' in cur: return True
  else: return False
```

```
n is the length of the word.
Time: O(n)
Space: O(1)
```

Extras

1. [**Greatest Common Divisor**] Given two numbers as arguments, Write a function that calculates the greatest common divisor (GCD).

```
def GCD(A,B):
 while B:
   if A > B:
     A = A - B
   else:
     B = B - A
 return A
```

```
A is the first input to the function
B is the second input to the function
Time: O(min(A, B))
Space: O(1)
```

```
def GCD(a,b):
 if b == 0: return a
 return GCD(b, a%b)
```

```
A is the first input to the function
B is the second input to the function
```

```
Time: O(min(A, B))
Space: O(min(A, B))
```

2. [**Lowest Common Multiple**] Given two numbers as arguments, Write a function that calculates the lowest common multiple (LCM).

```
def GCD(a,b):
  if b == 0: return a
  return GCD(b, a%b)

def LCM(a,b):
  if a > b: return (a/gcd(a,b)) * b
  else: return (b/gcd(a,b)) * a
```

```
A is the first input to the function
B is the second input to the function
Time: O(min(A, B))
Space: O(min(A, B))
```

3. [**Fizz Buzz**] [**leetcode 412**] Given an integer `n`, write a function to return a string array in the Fizz-Buzz pattern.

```
def fizzBuzz(n):
 answer = []
 for i in range(1, n+1):
   if (i%3 == 0) and (i%5 == 0):
     answer.append('FizzBuzz')
   elif (i%3 == 0):
     answer.append('Fizz')
   elif (i%5 == 0):
     answer.append('Buzz')
   else:
     answer.append(str(i))
 return answer
```

Tips & Tricks

1. [**Annotate Dataset**] How to quickly annotate datasets.
   - Ensure to first install the *pigeon* module. `!pip install pigeon-jupyter`

```
from pigeon import annotate

data = annotate(
```

```
      ["The food taste good.", "Very awesome resturant!", "Terrible
place."],
      options=["positive", "negative"]
  )
```

2. [**Binary Search**] How to execute binary search on sorted array.
    - Ensure to the list/ sequence is sorted in ascending order.

```
def search(nums, target):
  l, mid, r = 0, 0, len(nums)
  step = 0
  while (l <= r):
    print(f"Step{step}: {nums[l:r+1]}")
    step += 1
    mid = (l+r) // 2
    if target == nums[mid]:
      return mid
    elif target < nums[mid]:
      r = mid - 1
    else:
      l = mid + 1
  return f'{target} not found'

# Driver Code
nums, target = [1,2,3,4,5,6,7,8,9], 0
search(nums, target)
```

3. [**Email Sender**] How to send email with smtp.
    - Ensure to first setup 2FA with your email provider (Gmail)

```
def sendEmail(sender, password, reciever, subject, message):
  '''Sends emails using the smtp protocol'''
  import email, ssl, smtplib
  message = email.message.EmailMessage()
  message['To'] = reciever
  message['From'] = sender
  message['Subject'] = subject
  message['Date'] = email.utils.formatdate(localtime=True)
  message['Message-ID'] = email.utils.make_msgid()
  message.set_content(body)

  context = ssl.create_default_context()
  with smtplib.SMTP_SSL('smtp.gmail.com', 465, context=context) as
smtp:
    smtp.login(sender, password)
    smtp.sendmail(sender, reciever, message.as_string())

# Driver Code
```

```
sender, password = 'sender@gmail.com', 'sender_password'
reciever = 'reciever@gmail.com'

subject = 'ANNONCEMENT'
body = '''"With great power, comes great responsibility." - Uncle
Ben.'''
sendEmail(sender, password, reciever, subject, body)
```

4. [**English Dictionary**] How to implement a language dictionary.
    - Ensure to first install the *PyDictionary* module. `!pip install PyDictionary`

```
def dictionary(word):
  '''Returns the meaning of the word'''
  from PyDictionary import PyDictionary
  diction = PyDictionary()
  result = diction.meaning(word)
  for key, value in result.items():
    for idx,item in enumerate(value):
      print(f"[{idx+1}] [{key}]: {item}")

# Driver Code
word = input("Enter a word: ")
print(dictionary(word))
```

5. [**Face Detection**] How to detect faces in an image.
    - Ensure to first install the *opencv* module. `!pip install opencv-python`

```
import cv2

face_cascade_file =
cv2.CascadeClassifier('/content/face_detection/haarcascade_frontalface
_default.xml')
img = cv2.imread('/content/face_detection/chris.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade_file.detectMultiScale(gray, 1.1, 4)
# Drawing rectangle around face
for (x, y, w, h) in faces:
  cv2.rectangle(img, (x,y), (x+w, y+h), (0, 225, 0), 2)

cv2.imwrite("face_detected.jpg", img)
from IPython.display import Image
Image(filename='face_detected.jpg')
```

6. [**Files and Folders (using pathlib)**] Creating files and folders.
    - Both the *pathlib* and the *calendar* modules come with the default python installation.

```
from pathlib import Path
import calendar

months = list(calendar.month_name[1:])
weeks = ['Week 1', 'Week 2', 'Week 3', 'Week 4']

for i, month in enumerate(months, start=1):
  for week in weeks:
    Path(f"2022/{i}.{month}/{week}").mkdir(parents=True,exist_ok=True)
```

7. [**Files and Folders (Get Specific Files)**] How to search for specific files and folders.
    - Both the *pathlib* modules come with the default python installation.

```
from pathlib import Path

folder = Path('.')
paths = folder.glob('**/*.csv')
for path in paths:
  if path.is_file():
    print(path)
```

8. [**Files and Folders (Unzipping Files)**] How to zip/ unzip files.
    - Both the *pathlib* and the *zipfile* modules come with the default python installation.

```
from pathlib import Path
import zipfile

current_directory = Path('.')
target_directory = Path('temp')

for path in current_directory.glob('*.zip'):
    with zipfile.ZipFile(path, 'r') as zip_file:
        zip_file.extractall(path=target_directory)
```

9. [**Image Manipulation (Resizing)**] How to resize images.
    - The *PIL* module should come by default. If not, use this command. `!pip install pillow`

```
from PIL import Image
img = Image.open('/content/me.jpg')
resized_img = img.resize((150, 150))
resized_img.save('me_150.jpg')

display(resized_img)
```

10. [**Image Manipulation (Remove Background)**] How to remove background from images.
    - The *PIL* module should come by default. However, you need to install the *rembg* module. `!pip install pillow` and `!pip install rembg`

```
from PIL import Image
from rembg import remove
img = Image.open('/content/me_150.jpg')
rem_back_img = remove(img)

display(rem_back_img)
```

11. [**Math to Latex Description**] How to convert equations to latex descriptions
    - The *math* module comes with default python but the *latexify-py* module would need to be installed. `!pip install latexify-py`.

```
import math
import latexify

@latexify.with_latex
def solve(a,b,c):
  return (-b + math.sqrt(b**2 - 4*a*c)) / (2*a)
solve
```

12. [**QR Code Generator (texts & URLs)**] How to generate QR codes for texts and URLs.
    - Ensure to install the qrcode and image modules `!pip install qrcode image`

```
def generateQRcode(text):
  '''returns image of a QR code'''
  qr = qrcode.QRCode(
      version=1,
      error_correction=qrcode.constants.ERROR_CORRECT_L,
      box_size=10, border=4,
  )
  qr.add_data(text)
  qr.make(fit=True)
  img = qr.make_image(fill_color="black", back_color="white")
  img.save('test.png')

# Driver Code
generateQRcode('https://www.youtube.com/watch?v=dQw4w9WgXcQ')
Image(filename='test.png')
```

13. [**QR Code Generator (WIFI)**] How to generate QR codes for WIFI.

- Ensure to install the wifi-qrcode-generator module `!pip install wifi-qrcode-generator`

```
import wifi_qrcode_generator as qr
from IPython.display import Image
qr.wifi_qrcode('WIFI name', False, 'WPA', 'WIFI password')
```

14. [**Schedule Functions**] How to schedule functions to run at specific time.
    - Ensure to install the schedule module `!pip install schedule`

```
def sayHi():
  print("Hi")
  return schedule.CancelJob

import schedule, time
schedule.every(3).seconds.do(sayHi)

while True:
  schedule.run_pending()
```

15. [**Send Text Messages**] How to texts to be sent.
    - Ensure to use the textbelt API.

```
def sendTextMessage(msg):
  '''Sends scheduled text messages'''
  import requests
  resp = requests.post('https://textbelt.com/text', {
    'phone': '1234567890',
    'message': f'{msg}',
    'key': 'textbelt',
  })
  print(resp.json)

# Driver Code
msg = 'Hello world'
sendTextMessage(msg)
```

16. [**Site Connectivity Checker**] How to check connectivity status of a site.
    - The *urllib* package should come installed by default. if not, use the command `!pip install urllib`

```
def checkSiteConnectivity(url):
  '''Returns the status code of any url'''
```

```
    import urllib.request as request
    response = request.urlopen(url)
    return f"The connection status code is: {response.getcode()}"

# Driver Code
url = 'https://www.youtube.com/watch?
v=dQw4w9WgXcQ&list=RDdQw4w9WgXcQ&start_radio=1'
print(checkSiteConnectivity(url))
```

17. [**Timing Your Code**] How to time code execution accurately.
    - The time module should come by default. Ensure to import it.

```
import time

# Method 1
start = time.time()
time.sleep(5)
end = time.time()
print(end - start)

# method 2
start = time.perf_counter()
time.sleep(5)
end = time.perf_counter()
print(end - start)
```

18. [**Using Pipes**] How to use pipes for cleaner code.
    - Ensure to install the *pipe* package using the command. `!pip install pipe`

```
from pipe import select, where

nums = [1,2,3,4,5,6,7,8,9]
# Without Pipes
without_pipes = list(
    filter(lambda x: x % 2 == 0,
            map(lambda x: x ** 2, nums))
)

# With Pipes
with_pipes =list(
    nums | select(lambda x: x ** 2) | where(lambda x: x % 2 == 0)
)

print(without_pipes)
print(with_pipes)
```