Longest common substring (DP)

```
# Time c. O(nm), space c. O(nm)
def longest_common_substring(s1, s2):
    m = [[0] * (1+len(s2)) for i in\
                    range(1+len(s1))]
    max_len, end_idx = 0, 0
    for i in range(1, 1 + len(s1)):
        for j in range(1, 1 + len(s2)):
            if s1[i - 1] == s2[j - 1]:
                m[i][j] = m[i-1][j-1] + 1
                if m[i][j] > max_len:
                    max_len = m[i][j]
                    end_idx = i
            else:
                m[i][j] = 0
    return s1[end_idx-max_len : end_idx]
```

Make sentences with dictionary

```
def make_sent(string, dictionaries):
    global count
    if len(string) == 0:
        return True
    for i in range(0, len(string) + 1):
        prefix,suffix=string[:i],string[i:]
        if prefix in dictionaries:
            if suffix in dictionaries or\
               make_sent(suffix, dictionaries):
                count += 1
    return True

count = 0
string1 = "applet"
dictionary1 = {'app','let','apple','t','applet'}
print( make_sent(string1, dictionary1) )
print( count )
```

```
True
3
```

Longest common subsequence (DP)

```
# time c. O(nm), space c.
def lcs(s1, s2):
    matrix = [ ['' for x in range(len(s2))]\
               for y in range(len(s1)) ]
    for i in range(len(s1)):
        for j in range(len(s2)):
            if s1[i] == s2[j]:
                if i == 0 or j == 0:
                    matrix[i][j] = s1[i]
                else:
                    matrix[i][j] = matrix[i-1][j-1] +\
                                   s1[i]
            else:
                matrix[i][j] = max( matrix[i-1][j],
                                    matrix[i][j-1],
                                    key=len)
    return matrix[-1][-1]
```

Longest common substring in array

```
# time c. O(n*n*(n + n-1 + n-2 ... etc.))?
def longest_substr(arr):

    if len(arr) <= 1:
        return ''
    res       = ''
    reference = arr[0]
    for i in range(len(reference)):
        for j in range(len(reference)-i+1):
            candidate = reference[i:i+j]
            if j > len(res) and\
               all(candidate in x for x in arr):
                res = candidate
    return res
```

Max subarray sum (sum + indices)

```
def max_subarray_sum(arr):
    if len(arr)==0: return 0
    max_sum = curr_sum = arr[0]  # if neg
    start, tstart, end = 0, 0, 0
    for i in range(1, len(arr)):
        if arr[i] > curr_sum + arr[i]:
            curr_sum = arr[i]
            tstart = i
        else:
            curr_sum += arr[i]
        if curr_sum > max_sum:
            max_sum = curr_sum
            start = tstart
            end = i
    return max_sum, arr[ start:end+1 ]
```

All contig. subarrays w/sum k (neg or pos)

```
# Time c. O(n), space c. O(n)
from collections import defaultdict
def findSubarraysWithSumK(arr, k):
    hash_map = defaultdict(list)  # key=sum, value=end indices
    res      = []                 # subarrays
    curr_sum = 0
    for i in range( len(arr) ):
        curr_sum += arr[i]
        if curr_sum == k:
            res.append(arr[:i+1]) # or res.append((0, i+1))
        if curr_sum-k in hash_map:
            for value in hash_map[ curr_sum-k ]:
                res.append( arr[ value+1:i+1 ] )
                # or res.append((value+1, i+1))
        hash_map[ curr_sum ].append(i)
    return res
```

All non-contig. pairs sum up to k (unsorted)

```
def pair_sum(arr, k):
    if len(arr) < 2: return
    seen, output = set(), set()
    for num in arr:
        if k-num not in seen:
            seen.add(num)
        else:
            output.add( (min(num,k-num),  max(num,k-num)) )
    return '\n'.join(map(str,list(output)))
```

Non-contig pair, sum closest to k (sorted)

```
# time c. O(n)
MAX_VAL = 1000000000
def closest_sum(arr, n, k):
    res = 0, 0
    l, r, diff = 0, n-1, MAX_VAL
    while l < r:
        if abs(arr[l] + arr[r] - k) < diff:
            res = l, r
            diff = abs(arr[l] + arr[r] - k)
        if arr[l] + arr[r] > k:
            r -= 1
        else:
            l += 1
    return arr[res[0]], arr[res[1]]
```

Partition - two subarrays equal sum

Optional

```
# Recursive, t. complexity = 2**n
def isSubsetSum(arr, n, sum_):
    if sum_ == 0:
        return True
    if n == 0 and sum_ != 0:
        return False
    if arr[n-1] > sum_:
        return isSubsetSum(arr, n-1, sum_)
    return isSubsetSum(arr, n-1, sum_) or\
           isSubsetSum(arr, n-1, sum_-arr[n-1])
```

```
# t.c.= O(n*sum)
def isSubsetSum_dp(arr, n):
    sum_ = 0
    for i in range(n): sum_ += arr[i]
    if sum_ % 2 != 0:  return 0
    part = [0] * ((sum_ // 2) + 1)  # 0 (False)
    # part[j]=true if exists subset w/sum=j
    for i in range(n):
        for j in range(sum_ // 2, arr[i]-1, -1) :
            if (part[j - arr[i]] == 1 or j == arr[i]):
                part[j] = 1
    return part[sum_ // 2]
```

Partition - two subarrayys w/min diff. sums

Finds arrays w/min. possible difference


```
# t.c.= O(2**n)
def findMinRec(arr, i, sumCalculated, sumTotal):
    # If last elem, sum of one subset = sumCalculated,
    # sum of other subset = (sumTotal-sumCalculated). Return abs diff
    if i == 0:
        return abs( (sumTotal-sumCalculated) - sumCalculated )
    # For each arr[i], (1) we can exclude it from first set,
    # (2) we can include it. return min of two choices
    return min( findMinRec( arr,
                            i-1,
                            sumCalculated + arr[i-1],
                            sumTotal),
                findMinRec( arr,
                            i-1,
                            sumCalculated,
                            sumTotal ) )

def findmin_recursive(arr,  n):
    # total sum of arr
    sumTotal = 0
    for i in range(n):
        sumTotal += arr[i]
    return findMinRec(arr, n, 0, sumTotal)
```

Smallest +int != sum of subarray

- If arr[i] > res => found a gap [1, 2, 5] => 4
- Elems after arr[i] > res
- res = solution

```
# time c. O(n) - sorted array
def findSmallest(arr, n):
    res = 1
    for i in range(n):
        if arr[i] <= res:
            res = res + arr[i]
        else:
            break
    return res
```

Balanced parenth. check

```
def balance_check(s):
    if len(s)%2 != 0: return False
    opening = set('([{')
    # matching pairs
    matches = set([ ('(',')'),
                    ('[',']'),
                    ('{','}'), ])
    stack = []
    for paren in s:
        if paren in opening:
            stack.append(paren)
        else:
            if len(stack) == 0:
                return False
            last_open = stack.pop()
            if (last_open,paren) not in matches:
                return False
    return len(stack) == 0
```

All non-contig. triplets w/ sum < k

```
# Time c. O(n^2); sort() + smart iteration
def count_triplets(arr, k):
    arr.sort()
    n     = len(arr)
    count = 0
    for i in range(n-2):
        start = i + 1
        end   = n - 1
        while start < end:
            if (arr[i] + arr[start] + arr[end]) >= k:
                end -= 1
            else:
                count += (end-start)    # array sorted
                start += 1
    return count
```

All non-contig. triplets w/sum=0

```
# Time c. O(n^2), space c. O(1)
def findTriplets(arr):
    arr.sort()
    n     = len(arr)
    count = 0
    for i in range(n-2):
        start = i + 1
        end   = n - 1
        while start < end:
            if arr[i] + arr[start] + arr[end] == 0:
                count += 1
                start += 1
                end   -= 1
            elif (arr[i] + arr[start] + arr[end] < 0):
                start += 1
            else:
                end -= 1
    return count
```

Count paths

```
# Time c. = O(c**n) exponential
def count_paths_rec(m, n):
    if(m == 1 or n == 1): return 1
    return count_paths_rec(m-1, n) + count_paths_rec(m, n-1)

# Time & space = O(mn)
def count_paths(m, n):
    if m < 1 or n < 1: return -1
    count = [[None for j in range(n)] for i in range(m)]
    for i in range(n): count[0][i] = 1
    for j in range(m): count[j][0] = 1
    for i in range(1, m):
        for j in range(1, n):
            count[i][j] = count[i - 1][j] + count[i][j - 1]
    return count[m - 1][n - 1]

# Time O(mn), space O(n)
def count_paths_dp(m, n):
    dp = [1 for i in range(n)]
    for i in range(1, m):
        for j in range(1, n):
            dp[j] = dp[j] + dp[j - 1]
    return dp[n - 1]
```

Get all permutations of string

```
def permute(s):
    out = []
    if len(s) == 1:
        out = [s]
    else:
        for i, char in enumerate(s):
            for perm in permute(s[:i] + s[i+1:]):
                out += [char + perm]
    return out

permute('abc')
```

```
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

Merge Sort (mid)

Usage: sorting **linked lists**, inversion count in nearly sorted arr
(i < j, but A[i] > A[j]), external sort (data too big for memory)

```
# time c. O(nLogn), space c. O(n)
def merge_sort(arr):
    # arr of length 1 - returned as is
    if len(arr) > 1:
        mid = len(arr) // 2
        left =  merge_sort(arr[ :mid ])
        right = merge_sort(arr[ mid: ])
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right)
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
    return arr
```

Binary Search

```
# time O(Logn), space (O1)
def binary_search(arr, value):
    if len(arr) == 0: return None
    min_idx, max_idx= 0, len(arr)
    while min_idx < max_idx:
        mid = (min_idx + max_idx) // 2
        if arr[mid] == value:
            return mid
        elif arr[mid] < value:
            min_idx = mid + 1
        else: max_idx = mid
    return None
```

Quick Sort (pivot)

Pick **pivot** element (*first, last, random, median*),
and **partition** array: all smaller elems before pivot,
greater elements after pivot Usage: sorting arrays

```
# O(nLogn), worst case O(n^2)
# time c. O(logn), qualifies as in-place
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)

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

Edit Levenshtein Distance

```
def levenshtein(s1, s2):
    if len(s1) > len(s2):
        s1, s2 = s2, s1
    distances = range(len(s1) + 1)
    for idx2, char2 in enumerate(s2):
        distances_ = [ idx2+1 ]
        for idx1, char1 in enumerate(s1):
            if char1 == char2:
                distances_.append( distances[ idx1] )
            else:
                distances_.append( 1 + min((distances[idx1],\
                        distances[idx1+1], distances_[-1])))
        distances = distances_
    return distances[-1]
```

Heap Sort

Usage: sort nearly sorted array,
k largest (smallest) elems
Complete binary tree = every level filled,
except possibly last, and nodes are in far left
Binary Heap = cbt where parent >(<) children
Procedure: build max heap; replace max w/ last elem;
reduce heap by 1; heapify root; repeat Array repr:
root = arr[0];
for any i-th node arr[i]:
a) arr[(i-1)/2] = parent node
b) arr[(2*i)+1] = left child
c) arr[(2*i)+2] = right child

```
# time O(nLogn), space in-place
def heap_sort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if (l < n and arr[i] < arr[l]):
        largest = l
    if (r < n and arr[largest] < arr[r]):
        largest = r
    if (largest != i):
        arr[i],arr[largest] = arr[largest
        heapify(arr, n, largest)
```

Graphs

- **vertices / nodes** w/names (keys) and payloads (additional info);
- unordered = **undirected graph** vs.**directed graph** (digraph);
- **weight** on edge - cost from 1 vertex to the other (distance betw. cities)
- **Path** - # vertices connected by edges.
- **Path length** - # edges in unweighted g. (n−1) OR **sum of weights** in weighted g.
- **Cycle** in directed g. - path starts / ends at same vertex.
- If no cycles - **acyclic graph**; if also directed - directed acyclic graph (**DAG**)

Adjacency matrix = [[0, 1, 1], [1, 0, 0,]]

- **time c. O(1)** *see if edge present, add / remove edge*;
- **O(n)** get vertex's *adj. list,* ***O(V^2)*** *_add vertex*
- memory hog: **space c. O(n^2)** => good for **dense graphs**:
- better for **weighted graphs**;
- good for matrix operations (**insights**);
- **Undirected** graphs - adj. matrix **symmetric**

Adjacency list r.: edge_list = [[0,1], [0,2],] or dict[list] (node: verts) or dict[dicts] (if weigh

- **time c. O(1)** *get vertex's adj. list* (array indexing)
- **O(E)** (E=# v-s) *if edge (i,j) exists*
- **Efficient storage** (**space c. = O(2E)**)

Objects/pointer r.

- **time c. O(n)** *access a node*;
- **space c. O(n)** if nodes only, **O(n^2)** w/pointers;
- better for **directed graphs** (pointers);
- **high search time c.**

Incidence matrix
rows = vertices, columns = edges, entries = is vertex incident to edge (bool)

BFS2

```
def bfs2(graph, start):
    visited, queue = {start}, [start]
    while queue:
        vertex = queue.pop(0)
        print(str(vertex) + " ", end="")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
bfs2(graph, 0)
```

```
0 1 2 3
```

DFS

queue.pop(0) for BFS

```
# time c. O(V+E), space c. O(V)
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex, end=' ')
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

graph = { 0: set([1, 2]), 1: set([2]),
          2: set([3]),    3: set([1, 2]) }
dfs(graph, 0)
```

```
0 2 3 1
```

Shortest path

Always returned first by BFS

```
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

graph = {'A': set(['B', 'C']),
     'B': set(['A', 'D', 'E']),
     'C': set(['A', 'F']),
     'D': set(['B']),
     'E': set(['B', 'F']),
     'F': set(['C', 'E'])}

shortest_path(graph, 'A', 'F')
```

```
['A', 'C', 'F']
```

```
def kahn_toposort( graph ):
    topo = []
    in_degree = { k: 0 for k in graph }
    for k in graph:
        for v in graph[ k ]:
            in_degree[v] += 1
    q = [k for k,v in in_degree.items() if v==0]
    while q:
        k = q.pop()
        topo.append( k )
        for v in graph[ k ]:
            in_degree[ v ] -= 1
            if in_degree[ v ] == 0:
                q.append( v )
    return topo if len(topo)==len(graph) else []
```

Articulation point (vertex) or **bridge** (edge) – disconnects graph when removed. Connectedness verified by DFS

All Paths from Vertex A to Vertex B

```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for nxt in graph[vertex] - set(path):
            if nxt == goal:
                yield path + [nxt]
            else:
                queue.append((nxt, path + [nxt]))
```

2a. Depth First Traversals (Binary Tree)

```
# time O(n),space O(h),h=height
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

# Root Left Right
def Preorder(root):
    if root:
        print(root.val, end=' ')
        Preorder(root.left)
        Preorder(root.right)

# Left Right Root
def Postorder(root):
    if root:
        Postorder(root.left)
        Postorder(root.right)
        print(root.val, end=' '),

# Left Root Right
def Inorder(root):
    if root:
        Inorder(root.left)
        print(root.val, end=' '),
        Inorder(root.right)
```

```
root            = Node(1)
root.left       = Node(2)
root.right      = Node(3)
root.left.left  = Node(4)
root.left.right = Node(5)
print("Preorder traversal:")
Preorder(root)
```

BFS (iterative)

```
# time O(n), space O(n)
class Node:
    # create a new node
    def __init__(self, key):
        self.val = key
        self.left = None
        self.right = None

# print height
def LevelOrder(root):
    # base case
    if root is None:
        return
    queue = []
    queue.append(root)
    while(len(queue) > 0):
        node = queue.pop(0)
        print(node.val, end=' ')
        if node.left is not None:
            queue.append(node.left)
        if node.right is not None:
            queue.append(node.right)
```

Array representation of bin. tree:
node i
parent: **(i-1)/2**
left child: **2*i + 1**
right child: **2*i + 2**

TYPES OF TREES:

- **m-ary tree** - rooted tree, each node has no more than m children
- **Binary tree** (BT) - each node has *at most two children*
- **Complete** BT = bin tree, **every level is full**, except possibly last (far **left**).
- **BST** - parent's value greater that values in left subtree & less values in right subtree
- **Binary Heap** (max / min) = cbt, parent >(<) children
- **Full** (proper or plane) BT - every node has *0 or 2 children*
- **Perfect** BT - all interior nodes have two children and all leaves have same depth
- **Infinite complete** BT - every node has two children (infinite levels)
- **Balanced** BT - L and R subtrees of each node differ in height by no more than 1
- **Degenerate (pathological)** tree - each parent has only one child = **linked list**
- **Representation**: list, dict, object

BFS (recursive) incl. height

```
# Recursive - time O(n^2), space O(n)
def LevelOrder_rec(root):
    h = height(root)
    for i in range(1, h+1):
        printGivenLevel(root, i)

# print nodes at a given level
def printGivenLevel(root , level):
    if root is None:
        return
    if level == 1:
        print(root.val, end=" ")
    elif level > 1 :
        printGivenLevel(root.left , level-1)
        printGivenLevel(root.right , level-1)

# compute height - number of nodes from root
def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1
```

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
LevelOrder_rec(root)
```

Check if BT is balanced

```
# time O(n)
def is_balanced(root):
    return -1 != get_depth(root)
# return 0 if unbalanc. else depth+1
def get_depth(root):
    if not root:
        return 0
    left  = get_depth(root.left)
    right = get_depth(root.right)
    if abs(left-right) > 1 or\
       left == -1 or right == -1:
        return -1
    return 1 + max(left, right)
```

Lowest common ancestor (any BT)

```
def lca(root, p, q):
    if not root or root is p or root is q:
        return root
    left = lca(root.left, p, q)
    right = lca(root.right, p, q)
    if left and right:
        return root
    return left if left else right
```

Check if tree is BST

```
# If BST=>inorder traversal = sorted list
def inorder( root ):
    if tree != None:
        inorder(root.left)
        tree_vals.append(root.val)
        inorder(root.right)

def sort_check( tree_vals ):
    return tree_vals == sorted(tree_vals)

tree_vals = []
inorder(tree)
sort_check(tree_vals)
```

Rotate matrix clockwise, in place

```
# time O(2M) -> O(M), space O(1)
class Solution(object):
    def rotate(self, matrix):
        self.transpose(matrix)
        self.reflect(matrix)

    def transpose(self, matrix):
        n = len(matrix)
        for i in range(n):
            for j in range(i+1, n):
                matrix[i][j], matrix[j][i] =\
                matrix[j][i], matrix[i][j]

    def reflect(self, matrix):
        n = len(matrix)
        for i in range(n):
            for j in range(n//2):
                matrix[i][j], matrix[i][-j-1] =\
                matrix[i][-j-1], matrix[i][j]
```

Invert a binary tree

```
def reverse(root):
    if not root: return
    root.left,root.right=root.right,root.left
    if root.left:
        reverse(root.left)
    if root.right:
        reverse(root.right)
```

Check if 2 BTs equal

```
# time O(min(N,M)), N,M=# nodes
# space O(min(h1, h2))
def is_same_tree(p, q):
    if not p and not q:
        return True
    if p and q and p.val == q.val:
        return is_same_tree(p.left, q.left) and\
               is_same_tree(p.right, q.right)
    return False
```

Maximum path sum

```
def max_path_sum(root):
    maximum = float("-inf")
    maximum = helper_max(root, maximum)
    return maximum

def helper_max(root, maximum):
    if not root: return 0
    left = helper_max(root.left, maximum)
    right = helper_max(root.right, maximum)
    maximum = max(maximum, left+right+root.val)
    return root.val + maximum

max_path_sum(tree)
```

Min / Max height

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

def min_height(root):
    if not root: return 0
    if not root.left or not root.right:
        return max(min_height(root.left),\
                   min_height(root.right))+1
    return min(min_height(root.left),\
               min_height(root.right)) + 1
```

Trim BST

```
def trimBST(tree, minVal, maxVal):
    if not tree: return
    tree.left=trimBST(tree.left, minVal, maxVal)
    tree.right=trimBST(tree.right, minVal, maxVal)
    if minVal<=tree.val<=maxVal:
        return tree
    if tree.val < minVal:
        return tree.right
    if tree.val > maxVal:
        return tree.left
```

BST nodes in given range

```
#Time O(h + k), k=num nodes
def getCount(root, low, high):
    if root == None: return 0
    if root.data==high and root.data==low:
        return 1
    if root.data <= high and root.data >= low:
        return (1 + getCount(root.left, low, high) +\
                getCount(root.right, low, high))
    elif root.data < low:
        return getCount(root.right, low, high)
    else:
        return getCount(root.left, low, high)
```

BST node w/value closest to k

```
def closest_value(root, k):
    a = root.val
    child = root.left if k < a else root.right
    if not child:
        return a
    b = closest_value(child, k)
    return min(k-a, k-b)
```

Queue with two stacks

```
class Queue2Stacks(object):
    def __init__(self):
        self.instack  = []
        self.outstack = []

    def enqueue(self,element):
        self.instack.append(element)

    def dequeue(self):
        if not self.outstack:
            while self.instack:
                self.outstack.append(
                    self.instack.pop()
                )
        return self.outstack.pop()
```

Queue - FIFO, stack - LIFO
Dequeue – double-ended queue

Compare 2 strings as linked lists

```
# 0=same, 1=1st str lexicographically >,
# -1 - 2nd
class Node:
    def __init__(self, key):
        self.c = key ;
        self.next = None

def compare(list1, list2):
    while (list1 and list2 and\
           list1.c == list2.c):
        list1 = list1.next
        list2 = list2.next
    if (list1 and list2):
        return 1 if list1.c > list2.c else -1
    if (list1 and not list2):
        return 1
    if (list2 and not list1):
        return -1
    return 0

list1 = Node('g')
list1.next = Node('o')
...
print(compare(list1, list2))
```

Stack using two queues

```
from queue import Queue
class Stack:
    def __init__(self):
        self.q1 = []
        self.q2 = []
        self.curr_size = 0

    def push(self, x):
        self.curr_size += 1
        self.q2.append(x)
        while self.q1:
            self.q2.append(self.q1.pop(0))
        self.q = self.q1
        self.q1 = self.q2
        self.q2 = self.q

    def pop(self):
        if not self.q1:
            return
        self.curr_size -= 1
        return self.q1.pop(0)

    def peek(self):
        if not self.q1:
            return -1
        return self.q1[0]

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

Next greater elem w/stack

[11, 14, 21] => 11:14, 14:21, 21:-1

```
# time O(n)
def printNGE(arr):
    stack = []
    stack.append(arr[0])
    for i in range(1, len(arr)):
        next_ = arr[i]
        if stack:
            elem = stack.pop()
            while elem < next_ :
                print(str(elem)+ " :  " + str(next_))
                if not stack:
                    break
                elem = stack.pop()
            if  elem > next_:
                stack.append(elem)
        stack.append(next_)
    while stack:
            elem = stack.pop()
            next_ = -1
            print(str(elem) + " : " + str(next_))

arr = [11, 14, 21, 3]
printNGE(arr)
```

Singly Linked List (SLL)

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

Doubly Linked List (DLL)

```
class DLL_Node(object):
    def __init__(self,value):
        self.value = value
        self.next = None
        self.prev = None
```

(SLL) nth to the last element

```
def nth_to_last_node(head, n):
    left_pointer  = head
    right_pointer = head
    for i in range(n-1):
        if not right_pointer.next:
            print('n > length list')
            return
        right_pointer = right_pointer.next
    while right_pointer.next:
        left_pointer  = left_pointer.next
        right_pointer = right_pointer.next
    return left_pointer.value
```

Check if cycle (SLL)

```
def cycle_check(node):
    marker1 = node
    marker2 = node
    while marker2 and marker2.next:
        marker1 = marker1.next
        marker2 = marker2.next.next
        if marker2 == marker1:
            return True
    return False
```

Length of linked list (SLL)

```
def length_LL(LL, head):
    if (not head):
        return 0
    else:
        return 1 +\
    length_LL(LL, head.next)
```

Reverse SLL

```
# time O(n), space O(1)
def reverse_sll(head):
    current = head
    previous = None
    next_node = None
    while current:
        next_node = current.next
        current.next = previous
        previous = current
        current = next_node
    return previous
```

Break cycle (SLL)

```
def removeLoop(head, loop_node):
    pointer1 = head
    while(1):
        pointer2 = loop_node
        while (pointer2.next != loop_node and\
               pointer2.next != pointer1):
            pointer2 = pointer2.next
        if pointer2.next == pointer1:
            break
        pointer1 = pointer1.next
    pointer2.next = None
```

Knapsack DP

```
# max sum of values for subset of weights <= W
# O(N*W) time and space
def knapSack(W, weights, values, n):
    DP = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                DP[i][w] = 0
            elif weights[i-1] <= w:
                DP[i][w] = max(values[i-1] + DP[i-1]\
                    [w - weights[i-1]],  DP[i-1][w])
            else:
                DP[i][w] = DP[i-1][w]
    return DP[n][W]

values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
print(knapSack(W, weights, values, len(values)))
```

Coin change (knapsack variant)

```
# recursion + memoization
# arr coins + target amnt => fewest coins as change
def coins_dp(target, coins, known_results):
    min_coins = target
    if target in coins:
        known_results[target] = 1
        return 1
    elif known_results[target] > 0:
        return known_results[target]
    else:
        for i in [c for c in coins if c <= target]:
            num_coins = 1 + coins_dp( target-i,
                                      coins,
                                      known_results, )
            if num_coins < min_coins:
                min_coins = num_coins
                known_results[target] = min_coins
    return min_coins
```

```
target = 74
coins = [1,5,10,25]
known_results = [0]*(target+1)
coins_dp(target, coins, known_results)
```

```
8
```

Num ways to cover distance

All ways to cover distance it w/1, 2, 3 steps

```
# time O(n)., space O(n)
def count_dp(dist):
    count = [0] * (dist + 1)
    count[0] = 1
    count[1] = 1
    count[2] = 2
    for i in range(3, dist + 1):
        count[i] = ( count[i-1] +\
                     count[i-2] +\
                     count[i-3] )
    return count[dist];

dist = 20
print( count_dp(dist))
```

```
121415
```

Fibonnaci dynamic

```
# memoization (dynamic)
# time O(n) - counting sequentially
def fib_dyn(n):
    if n == 0 or n == 1:
        return n
    if not n in memo:
        memo[n] = fib_dyn(n-1) +\
                  fib_dyn(n-2)
    return memo[n]

for n in range(20):
    memo = [None] * (n + 1)
    print(fib_dyn(n), end=', ')
```

Fibonnaci recursive

```
# time O(2^n)
def fib_rec(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)

# print 1st 20 Fib. nums
for i in range(20):
    print(fib_rec(i), end=', ')
```

Fibonnaci iterative

```
# iterative - tuple unpacking
# time O(n) - counting sequentially
def fib_iter(n):
    a = 0
    b = 1
    for i in range(n):
        a, b = b, a + b
    return a

for i in range(20):
    print(fib_iter(i), end=', ')
```

Memoization

```
def factorial(n):
    if n < 2:
        return 1
    if not n in memo:
        memo[n] = n*factorial(n-1)
    return memo[n]

memo = {}
factorial(1)
```

Recursion

```
def fact(n):
    if n < 2:
        return 1
    else:
        return n*fact(n-1)

fact(5)
```

Longest Substr. w/K Chars

```
# Time c. O(n), space c. O(1)
# At most k distinct characters
def LongestSubstring(s, k):
    n = len(s)
    if n*k==0:
        return 0
    l, r = 0, 0
    mapp = dict()
    max_len = 2
    for c in s:
        mapp[ s[r] ] = r
        r += 1
        if len(mapp) == k+1:
            del_idx = min(mapp.values())
            del mapp[ s[del_idx] ]
            l = del_idx + 1
        max_len = max(max_len, r - l)
    return max_len
```

Longest Substr. w/Unique Ch

```
def lengthOfLongestSubstring2(s):
    mapp = {c:0 for c in s}
    left = 0
    max_len = 0
    for right, c in enumerate(s):
        mapp[c] += 1
        while mapp[c] > 1:
            mapp[ s[left] ] -= 1
            left += 1
        max_len = max(max_len,
                      right-left+1)
    return max_len
```

Minimum Window Substring

```
# time/space c. O(n+m)
# min. contig. substr. of s w/all chars from t
from collections import Counter
def minWindow(s, t):
    if not t or not s:
        return ''
    # unique chars in t and curr window
    dict_t = Counter(t)
    curr = {}
    len_t  = len(dict_t)
    len_curr  = 0
    l,r = 0,0
    # length, l, r
    res = float('inf'), None, None
    while r < len(s):
        char = s[r]
        curr[ char ] = curr.get(char, 0) + 1
        if char in dict_t and\
        curr[char] == dict_t[char]:
            len_curr += 1
        while l <= r and len_curr == len_t:
            char = s[l]
            if r - l + 1 < res[0]:
                res = (r - l + 1, l, r)
            curr[ char ] -= 1
            if char in dict_t and\
            curr[char] < dict_t[char]:
                len_curr -= 1
            l += 1
        r += 1
    return '' if res[0]==float('inf') else\
           s[res[1]:res[2]+1]
```

Median of Two Sorted Arrays

```python
# time O(log(min(N,M)), space O(1). BIN SEARCH
def median(nums1, nums2):
    # capture edge cases
    if len(nums2) < len(nums1):
        nums1, nums2 = nums2, nums1
    total = len(nums1) + len(nums2)
    half = total // 2
    l, r = 0, len(nums1)-1
    # median is guaranteed
    while True:
        i = (l + r) // 2    # for nums1
        # subtr. 2 - j starts at 0, i starts at 0
        j = half - i - 2    # for nums2
        # overflow of indices
        nums1_left  = nums1[i] if i >= 0 else float("-inf")
        nums1_right = nums1[i+1] if (i+1) < len(nums1) else float("inf")
        nums2_left  = nums2[j] if j >= 0 else float("-inf")
        nums2_right = nums2[j+1] if (j+1) < len(nums2) else float("inf")
        # if correct partition is found
        if nums1_left <= nums2_right and nums2_left <= nums1_right:
            if total % 2:
                return min(nums1_right, nums2_right)
            else:
                return ( max(nums1_left, nums2_left) +\
                         min(nums1_right, nums2_right) ) / 2
        # if no correct partition - arrays are in ascending order
        elif nums1_left > nums2_right:
            r = i - 1
        else:
            l = i + 1
```

Merge k Sorted Linked Lists

```python
# time c. O(Nlogk), k = # lists. Space c. O(1)
def mergeKLists(lists):
    n = len(lists)
    interval = 1
    while interval < n:
        for i in range( 0, n-interval,
                        interval*2 ):
            lists[i] = merge2Lists( lists[i],
                                    lists[i+interval] )
        interval *= 2
    return lists[0] if n > 0 else None

def merge2Lists(l1, l2):
    head = point = Node(0)
    while l1 and l2:
        if l1.val <= l2.val:
            point.next = l1
            l1 = l1.next
        else:
            point.next = l2
            l2 = l1
            l1 = point.next.next
        point = point.next
    if not l1: point.next=l2
    else: point.next=l1
    return head.next
```

Multiply strings

```python
# GET NUMBER FROM CHAR: ord(char) - ord('0')
def multiply_strings(num1, num2):
    res = 0
    for i, c1 in enumerate(num1[::-1]):
        for j, c2 in enumerate(num2[::-1]):
            res += (ord(c1)-ord('0')) *\
                   (ord(c2)-ord('0')) *\
                   (10**(i+j))
    return str(res)
num1, num2 = '123', '456'
multiply_strings(num1, num2)        # '56088'
```

Merge k Sorted Arrays

```python
# time c. O(kN*Logk) since
# using heap (N*Logk) k times;
# space c. O(N) - output array
from heapq import merge
def mergeK(arr, k):
    l = arr[0]
    for i in range(k-1):
        l = list(merge(l, arr[i + 1]))
    return l
```

Kth Largest Element in Array


```
# time = O(n), space = O(1)
def findKthLargest(nums, k):
    def partition(left, right, pivot_idx):
        pivot = nums[pivot_idx]
        # 1. move pivot to end
        nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
        # 2. move all smaller elements to the left
        store_idx = left
        for i in range(left, right):
            if nums[i] < pivot:
                nums[store_idx], nums[i] = nums[i], nums[store_idx]
                store_idx += 1
        # 3. move pivot to its final place
        nums[right], nums[store_idx] = nums[store_idx], nums[right]
        return store_idx

    def select_rec(left, right, k_smallest):
        if left == right:    # base case - 1 elem
            return nums[left]
        pivot_idx = random.randint(left, right)
        # find pivot pos in sorted list
        pivot_idx = partition(left, right, pivot_idx)
        # if pivot in final sorted position
        if k_smallest == pivot_idx:
             return nums[k_smallest]
        elif k_smallest < pivot_idx:    # go left
            return select_rec(left, pivot_idx-1, k_smallest)
        else:                           # go right
            return select_rec(pivot_idx+1, right, k_smallest)
    # kth largest = (n - k)th smallest
    return select_rec(0, len(nums)-1, len(nums)-k)
```

Dijkstra's Algorithm

Shortest path from one vertex to all others

```
# time c. O(V + E*logE), iterative, BFS
import heapq    # min heap of (distance, vertex) pairs
# control iteration order - to pick vertex w/smallest dist
def calculate_distances(graph, start):
    # cost from start to each destination
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    pq = [(0, start)]           # min heap or priority queue
    while len(pq) > 0:
        # pop smallest, maintain heap
        current_distance, current_vertex = heapq.heappop(pq)
        # nodes can be added to pq multiple times; process node
        # first time it's removed from pq
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            # only if new path is better
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                # push onto heap, maintain heap
                heapq.heappush(pq, (distance, neighbor))
    return distances
```

Min. spanning tree - Prim's algo

S.t. for a graph G = (V,E) is **acyclic subset of E connecting all vertices in V** (sum of edge weights minimized)
Most efficient info flow. There may be several spanning trees - we need to find the minimum one.
Using priority queue to select next vertex for growing graph

```
from collections import defaultdict
import heapq
def min_spanning_tree(graph, start):
    ''' Outputs mst - min. spanning tree '''
    mst     = defaultdict(set)
    visited = set([ start ])
    edges   = [ (cost, start, to)
                for to, cost in graph[start].items() ]
    heapq.heapify(edges)
    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst[frm].add(to)
            for to_next, cost in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost, to, to_next))
    return mst
```

Floyd Warshall Algorithm

Shortest path between all pairs of nodes

```
# time c. O(V^3)
def floyd_warshall(graph):
    dist = list(map(lambda i : list(map(lambda j : j , i)) , graph))
    for k in range(V):
        for i in range(V):          # all vertices as source
            for j in range(V):      # all vertices as destination
                # update dist[i][j] if vertex k on shortest path from i to j
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist
```

Traveling Salesman Problem (TSP)

NP hard problem. There is no known polynomial time solution

```python
# Naive approach. Time c. n!
from sys import maxsize
V = 4
def travellingSalesmanProblem(graph, s):
    vertices = []          # all verteces, but source vtx
    for i in range(V):
        if i != s:
            vertices.append(i)
    min_pathweight = maxsize    # min weight Hamiltonian Cycle
    while True:
        current_pathweight = 0 # current Path weight(cost)
        k = s                  # compute current path weight
        for i in range(len(vertices)):
            current_pathweight += graph[k][vertices[i]]
            k = vertices[i]
        current_pathweight += graph[k][s]
        min_pathweight = min(min_pathweight,
                             current_pathweight) #update min
        if not next_permutation(vertices):
            break
    return min_pathweight
```

```python
def next_permutation(L):
    n = len(L)
    i = n - 2
    while i >= 0 and L[i] >= L[i + 1]:
        i -= 1
    if i == -1:
        return False
    j = i + 1
    while j < n and L[j] > L[i]:
        j += 1
    j -= 1
    L[i], L[j] = L[j], L[i]
    left = i + 1
    right = n - 1
    while left < right:
        L[left], L[right] = L[right], L[left]
        left += 1
        right -= 1
    return True


# matrix representation of graph
graph = [[0, 10, 15, 20], [10, 0, 35, 25],
         [15, 35, 0, 30], [20, 25, 30, 0]]
start = 0
print(travellingSalesmanProblem(graph, s))
```