- [x] Array
- [x] Backtracking
- [x] Binary Trees
- [x] Bit Manipulation
- [x] Binary Search Trees
- [x] Dynamic Programming
- [x] Graph
- [x] Greedy
- [ ] Heap
- [ ] LinkedList
- [ ] Matrix
- [x] Searching & Sorting
- [ ] Stacks & Queues
- [ ] String
- [x] Trie

Array

Reverse the array
Find the maximum and minimum element in an array
Find the "Kth" max and min element of an array
Given an array which consists of only 0, 1 and 2. Sort the array without using any sorting algo
Move all the negative elements to one side of the array
Find the Union and Intersection of the two sorted arrays.
Write a program to cyclically rotate an array by one.
Find Largest sum contiguous Subarray [V. IMP] / Kadne's Algorithm

Minimise the maximum difference between heights [V.IMP]
Minimum no. of Jumps to reach end of an array
Find duplicate in an array of N+1 Integers
Merge 2 sorted arrays without using Extra space.
Merge Intervals
Next Permutation
Count Inversion
Best time to buy and Sell stock
Find all pairs on integer array whose sum is equal to given number
Find common elements In 3 sorted arrays
Rearrange the array in alternating positive and negative items with O(1) extra space
Find if there is any subarray with sum equal to 0
Find factorial of a large number
Find maximum product subarray
Find longest consecutive subsequence
Given an array of size n and a number k, find all elements that appear more than " n/k " times.
Maximum profit by buying and selling a share atmost twice
Find whether an array is a subset of another array
Find the triplet that sum to a given value
Trapping Rain water problem
Chocolate Distribution problem
Smallest Subarray with sum greater than a given value
Three way partitioning of an array around a given value
Minimum swaps required bring elements less equal K together

- Minimum no. of operations required to make an array palindrome
- Median of 2 sorted arrays of different size

BackTracking

- Rat in a maze Problem
- Printing all solutions in N-Queen Problem
- Word Break Problem using Backtracking
- Remove Invalid Parentheses
- Sudoku Solver
- m-Colouring Problem
- Print all palindromic partitions of a string
- Subset Sum Problem
- The Knight's tour problem
- Tug of War
- Find shortest safe route in a path with landmines
- Combinational Sum
- Find Maximum number possible by doing at-most K swaps
- Print all permutations of a string
- Find if there is a path of more than k length from a source
- Longest Possible Route in a Matrix with Hurdles
- Print all possible paths from top left to bottom right of a mXn matrix
- Partition of a set intoK subsets with equal sum
- Find the K-th Permutation Sequence of first N natural numbers

Binary Trees

- level order traversal AKA BFS
- Reverse Level Order traversal
- Height of a tree
- Diameter of a tree
- Mirror of a tree / Invert Binary Tree
- Inorder, Preorder and Postorder Tree Traversal (Recursive Method)
- Left View of a tree
- Right View of Tree
- Top View of a tree
- Bottom View of a tree
- Zig-Zag traversal of a binary tree
- Check if a tree is balanced or not
- Diagonal Traversal of a Binary tree
- Boundary traversal of a Binary tree
- Construct Binary Tree from String with Bracket Representation
- Convert Binary tree into Doubly Linked List
- Convert Binary tree into Sum tree
- Construct Binary tree from Inorder and preorder traversal
- Find minimum swaps required to convert a Binary tree into BST
- Check if Binary tree is Sum tree or not
- Check if all leaf nodes are at same level or not
- Check if a Binary Tree contains duplicate subtrees of size 2 or more [IMP]
- Check if 2 trees are mirror or not
- Sum of Nodes on the Longest path from root to leaf node
- Check if given graph is tree or not. [IMP]
- Find Largest subtree sum in a tree
- Maximum Sum of nodes in Binary tree such that no two are adjacent
- Print all "K" Sum paths in a Binary tree
- Find Least Common Ancestor in a Binary tree
- Find distance between 2 nodes in a Binary tree
- Kth Ancestor of node in a Binary tree
- Find all Duplicate subtrees in a Binary tree [IMP]
- Tree Isomorphism Problem

Bit Manipulation

- Count set bits in an integer
- Find the two non-repeating elements in an array of repeating elements
- Count number of bits to be flipped to convert A to B

- Count total set bits in all numbers from 1 to n
- Program to find whether a no is power of two
- Find position of the only set bit
- Copy set bits in a range
- Divide two integers without using multiplication, division and mod operator
- Calculate square of a number without using *, / and pow()
- Power Set

Binary Search Trees

- Find a value in a BST
- Deletion of a node in a BST
- Find min and max value in a BST
- Find inorder successor and inorder predecessor in a BST
- Check if a tree is a BST or not
- Populate Inorder successor of all nodes
- Find LCA of 2 nodes in a BST
- Construct BST from preorder traversal
- Convert Binary tree into BST
- Convert a normal BST into a Balanced BST
- Merge two BST
- Find Kth largest element and Kth smallest element in a BST
- Count pairs from 2 BST whose sum is equal to given value "X"
- Find the median of BST in O(n) time and O(1) space
- Count BST nodes that lie in a given range
- Replace every element with the least greater element on its right
- Given "n" appointments, find the conflicting appointments
- Preorder to Postorder
- Check whether BST contains Dead end
- Largest BST in a Binary Tree [V.V.V.V.V IMP]
- Flatten BST to sorted list

Dynamic Programming

- Coin ChangeProblem
- Knapsack Problem
- Binomial CoefficientProblem
- Permutation CoefficientProblem
- Program for nth Catalan Number
- Matrix Chain Multiplication
- Edit Distance
- Subset Sum Problem
- Friends Pairing Problem
- Gold Mine Problem
- Assembly Line SchedulingProblem
- Painting the Fence Problem
- Rod Cutting Problem
- Longest Common Subsequence
- Longest Repeated Subsequence
- Longest Increasing Subsequence
- Space Optimized Solution of LCS (Print only length)
- LCS (Longest Common Subsequence) of three strings
- Maximum Sum Increasing Subsequence
- Count all subsequences having product less than K
- Longest subsequence such that difference between adjacent is one
- Maximum subsequence sum such that no three are consecutive
- Egg Dropping Problem
- Maximum Length Chain of Pairs
- Maximum size square sub-matrix with all 1s
- Maximum sum of pairs with specific difference
- Min Cost PathProblem
- Maximum difference of zeros and ones in binary string
- Minimum number of jumps to reach end
- Minimum cost to fill given weight in a bag

- Minimum removals from array to make max –min <= K
- Longest Common Substring
- Count number of ways to reacha given score in a game
- Count Balanced Binary Trees of Height h
- Smallest sum contiguous subarray
- Unbounded Knapsack (Repetition of items allowed)
- Largest Independent Set Problem
- Partition problem
- Longest Palindromic Subsequence
- Count All Palindromic Subsequence in a given String
- Longest Palindromic Substring
- Longest alternating subsequence
- Weighted Job Scheduling
- Coin game winner where every player has three choices
- Count Derangements (Permutation such that no element appears in its original position) [IMPORTANT]
- Maximum profit by buying and selling a share at most twice [IMP]
- Optimal Strategy for a Game
- Optimal Binary Search Tree
- Palindrome PartitioningProblem
- Word Wrap Problem
- Mobile Numeric Keypad Problem [IMP]
- Boolean Parenthesization Problem
- Largest rectangular sub-matrix whose sum is 0
- Maximum sum rectangle in a 2D matrix
- Maximum profit by buying and selling a share at most k times
- Find if a string is interleaved of two other strings

Graph

- Implement Graph
- Implement Weighted Graph
- Implement BFS algorithm
- Implement DFS Algo
- Detect Cycle in Directed Graph using BFS/DFS Algo
- Detect Cycle in UnDirected Graph using BFS/DFS Algo
- Minimum Step by Knight
- flood fill algo
- Clone a graph
- Making wired Connections
- word Ladder
- Dijkstra algo
- Implement Topological Sort
- Minimum time taken by each job to be completed given by a Directed Acyclic Graph
- Find whether it is possible to finish all tasks or not from given dependencies
- Find the no. of Islands
- Given a sorted Dictionary of an Alien Language, find order of characters
- Implement Kruksal'sAlgorithm
- Implement Prim's Algorithm
- Total no. of Spanning tree in a graph
- Implement Bellman Ford Algorithm
- Implement Floyd warshallAlgorithm
- Travelling Salesman Problem
- Graph ColouringProblem
- Snake and Ladders Problem
- Find bridge in a graph
- Count Strongly connected Components(Kosaraju Algo)
- Check whether a graph is Bipartite or Not
- Longest path in a Directed Acyclic Graph
- Journey to the Moon
- Cheapest Flights Within K Stops
- Oliver and the Game
- Water Jug problem using BFS

- Find if there is a path of more thank length from a source
- Minimum edges to reverse o make path from source to destination
- Paths to travel each nodes using each edge(Seven Bridges)
- Vertex Cover Problem
- Chinese Postman or Route Inspection
- Number of Triangles in a Directed and Undirected Graph
- Minimise the cashflow among a given set of friends who have borrowed money from each other
- Two Clique Problem

Greedy

- Activity Selection Problem
- Huffman Coding
- Water Connection Problem
- Fractional Knapsack Problem
- Greedy Algorithm to find Minimum number of Coins
- Maximum trains for which stoppage can be provided
- Minimum Platforms Problem
- Buy Maximum Stocks if i stocks can be bought on i-th day
- Find the minimum and maximum amount to buy all N candies
- Minimum Cost to cut a board into squares
- Check if it is possible to survive on Island
- Maximum product subset of an array
- Maximize array sum after K negations
- Maximize the sum of arr[i]*i
- Maximum sum of absolute difference of an array
- Maximize sum of consecutive differences in a circular array
- Minimum sum of absolute difference of pairs of two arrays
- Program for Shortest Job First (or SJF) CPU Scheduling
- Program for Least Recently Used (LRU) Page Replacement algorithm
- Smallest subset with sum greater than all other elements
- Chocolate Distribution Problem
- DEFKIN -Defense of a Kingdom
- DIEHARD -DIE HARD
- GERGOVIA -Wine trading in Gergovia
- Picking Up Chicks
- CHOCOLA –Chocolate
- ARRANGE -Arranging Amplifiers
- K Centers Problem
- Minimum Cost of ropes
- Find smallest number with given number of digits and sum of digits
- Rearrange characters in a string such that no two adjacent are same
- Find maximum sum possible equal sum of three stacks

Heap

- Implement a Maxheap/MinHeap using arrays and recursion.
- Sort an Array using heap. (HeapSort)
- Maximum of all subarrays of size k.
- “k” largest element in an array
- Kth smallest and largest element in an unsorted array
- Merge “K” sorted arrays. [IMP]
- Merge 2 Binary Max Heaps
- Kth largest sum continuous subarrays
- Leetcode- reorganize strings
- Merge “K” Sorted Linked Lists [V.IMP]
- Smallest range in “K” Lists
- Median in a stream of Integers
- Check if a Binary Tree is Heap
- Connect “n” ropes with minimum cost
- Convert BST to Min Heap
- Convert min heap to max heap
- Rearrange characters in a string such that no two adjacent are same.
- Minimum sum of two numbers formed from digits of an array

Linked List

- Write a Program to reverse the Linked List. (Both Iterative and recursive)
- Reverse a Linked List in group of Given Size. [Very Imp]
- Write a program to Detect loop in a linked list.
- Write a program to Delete loop in a linked list.
- Find the starting point of the loop.
- Remove Duplicates in a sorted Linked List.
- Remove Duplicates in a Un-sorted Linked List.
- Write a Program to Move the last element to Front in a Linked List.
- Add “1” to a number represented as a Linked List.
- Add two numbers represented by linked lists.
- Intersection of two Sorted Linked List.
- Intersection Point of two Linked Lists.
- Merge Sort For Linked lists.[Very Important]
- Quicksort for Linked Lists.[Very Important]
- Find the middle Element of a linked list.
- Check if a linked list is a circular linked list.
- Split a Circular linked list into two halves.
- Write a Program to check whether the Singly Linked list is a palindrome or not.
- Deletion from a Circular Linked List.
- Reverse a Doubly Linked list.
- Find pairs with a given sum in a DLL.
- Count triplets in a sorted DLL whose sum is equal to given value “X”.
- Sort a “k”sorted Doubly Linked list.[Very IMP]
- Rotate DoublyLinked list by N nodes.
- Rotate a Doubly Linked list in group of Given Size.[Very IMP]
- Can we reverse a linked list in less than O(n) ?
- Why Quicksort is preferred for. Arrays and Merge Sort for LinkedLists ?
- Flatten a Linked List
- Sort a LL of 0's, 1's and 2's
- Clone a linked list with next and random pointer
- Merge K sorted Linked list
- Multiply 2 no. represented by LL
- Delete nodes which have a greater value on right side
- Segregate even and odd nodes in a Linked List
- Program for n’th node from the end of a Linked List
- Find the first non-repeating character from a stream of characters

Matrix

- Spiral traversal on a Matrix
- Search an element in a matriix
- Find median in a row wise sorted matrix
- Find row with maximum no. of 1's
- Print elements in sorted order using row-column wise sorted matrix
- Maximum size rectangle
- Find a specific pair in matrix
- Rotate matrix by 90 degrees
- Kth smallest element in a row-column wise sorted matrix
- Common elements in all rows of a given matrix

Searching & Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Counting Sort
- Heap Sort
- Radix Sort
- Linear Search
- Binary Search
- Interpolation Search

- Find first and last positions of an element in a sorted array
- Find a Fixed Point (Value equal to index) in a given array
- Search in a rotated sorted array
- square root of an integer
- Find the repeating and the missing
- Searching in an array where adjacent differ by at most k
- find a pair with a given difference
- find two elements that sum to a given value - TwoSum
- find four elements that sum to a given value - ThreeSum
- find four elements that sum to a given value - - FourSum
- maximum sum such that no 2 elements are adjacent
- Count triplet with sum smaller than a given value
- print all subarrays with 0 sum
- Product array Puzzle
- Sort array according to count of set bits
- minimum no. of swaps required to sort the array
- Find pivot element in a sorted array
- K-th Element of Two Sorted Arrays
- Aggressive cows
- Book Allocation Problem
- EKOSPOJ:
- Missing Number in AP
- Smallest number with atleastn trailing zeroes infactorial
- ROTI-Prata SPOJ
- DoubleHelix SPOJ
- Subset Sums
- Implement Merge-sort in-place

Stacks & Queues

- Implement Stack from Scratch
- Implement Queue from Scratch
- Implement 2 stack in an array
- find the middle element of a stack
- Implement "N" stacks in an Array
- Check the expression has valid or Balanced parenthesis or not.
- Reverse a String using Stack
- Design a Stack that supports getMin() in O(1) time and O(1) extra space.
- Find the next Greater element
- The celebrity Problem
- Arithmetic Expression evaluation
- Evaluation of Postfix expression
- Implement a method to insert an element at its bottom without using any other data structure.
- Reverse a stack using recursion
- Sort a Stack using recursion
- Merge Overlapping Intervals
- Largest rectangular Area in Histogram
- Length of the Longest Valid Substring
- Expression contains redundant bracket or not
- Implement Stack using Queue
- Implement Stack using Deque
- Stack Permutations (Check if an array is stack permutation of other)
- Implement Queue using Stack
- Implement "n" queue in an array
- Implement a Circular queue
- LRU Cache Implementationa
- Reverse a Queue using recursion
- Reverse the first “K” elements of a queue
- Interleave the first half of the queue with second half
- Find the first circular tour that visits all Petrol Pumps
- Minimum time required to rot all oranges
- Distance of nearest cell having 1 in a binary matrix

Array

Reverse the array

```
def reverseArray(A: list):
    start, end= 0, len(A)-1
    while start<end:
        A[start], A[end]= A[end], A[start]
        start+=1
        end-=1


A=[1,54,21,51,2,353,2,1,99,121,5,5]
reverseArray(A)
print("After reversing:", A)
```

Find the maximum and minimum element in an array

```
def getMinMax(arr: list, n: int):
    min = 0
    max = 0

    # If there is only one element then return it as min and max both
    if n == 1:
        max = arr[0]
        min = arr[0]
        return min, max

    # If there are more than one elements, then initialize min
    # and max
    if arr[0] > arr[1]:
        max = arr[0]
        min = arr[1]
    else:
        max = arr[1]
        min = arr[0]

    for i in range(2, n):
        if arr[i] > max:
            max = arr[i]
        elif arr[i] < min:
            min = arr[i]

    return min, max


arr = [1000, 11, 445, 1, 330, 3000]
arr_size = 6
min, max = getMinMax(arr, arr_size)
print("Minimum element is", min)
print("Maximum element is", max)
```

Find the "Kth" max and min element of an array

```
import sys

# function to calculate number of elements less than equal to mid
def count(nums, mid):
```

```
    cnt = 0
    for i in range(len(nums)):
        if nums[i] <= mid:
            cnt += 1
    return cnt

def kthSmallest(nums, k):
    low = sys.maxsize
    high = -sys.maxsize

    # calculate minimum and maximum the array.
    for i in range(len(nums)):
        low = min(low, nums[i])
        high = max(high, nums[i])

        # Our answer range lies between minimum and maximum element
        # of the array on which Binary Search is Applied
    while low < high:
        mid = low + (high - low) // 2
        # if the count of number of elements in the array less than equal
        # to mid is less than k then increase the number. Otherwise decrement
        # the number and try to find a better answer.
        if count(nums, mid) < k:
            low = mid + 1
        else:
            high = mid
    return low

nums = [1, 4, 5, 3, 19, 3]
k = 3
print("K'th smallest element is", kthSmallest(nums, k))
```

Given an array which consists of only 0, 1 and 2. Sort the array without using any sorting algo

```
def sort012(arr):
        n=len(arr)
        low=0
        high=n-1
        mid=0
        while mid<=high:
            if arr[mid]==0:
                arr[mid] , arr[low] = arr[low] , arr[mid]
                mid+=1
                low+=1

            elif arr[mid]==1:
                mid+=1

            else:
                arr[mid] , arr[high] = arr[high] , arr[mid]
                high-=1

A=[0,0,0,2,2,2,1,1,1,0,2,1,1,2,0]
sort012(A)
print("After sorting:", A)
```

Move all the negative elements to one side of the array

```
def RearrangePosNeg(arr):
 n = len(arr)
 for i in range(1, n):
  key = arr[i]

  # if current element is positive do nothing
  if (key > 0):
   continue

  # if current element is negative, shift positive elements of arr[0..i-1], to one position to
their right
  j = i - 1
  while (j >= 0 and arr[j] > 0):
   arr[j + 1] = arr[j]
   j = j - 1

  # Put negative element at its
  # right position
  arr[j + 1] = key


# Driver Code
if __name__ == "__main__":
 arr = [-12, 11, -13, -5,
  6, -7, 5, -3, -6]
 RearrangePosNeg(arr)
 print(arr)
```

Find the Union and Intersection of the two sorted arrays.

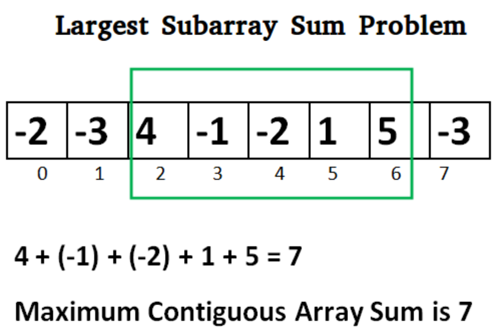
```
def printUnion(arr1, arr2, n1, n2):
    hs = set()
    # Insert the elements of arr1[] to set hs
    for i in range(0, n1):
     hs.add(arr1[i])
    # Insert the elements of arr2[] to set hs
    for i in range(0, n2):
     hs.add(arr2[i])
    for i in hs:
     print(i, end=" ")
    print("Union Count", len(hs))


def printIntersection(arr1, arr2, n1, n2):
    hs = set()
    # Insert the elements of arr1[] to set S
    for i in range(0, n1):
     hs.add(arr1[i])
    intersectCount=0
    for i in range(0, n2):
     # If element is present in set then
     # push it to vector V
     if arr2[i] in hs:
            print(arr2[i], end=" ")
            intersectCount+=1
    print("Intersection Count", intersectCount)
```

```
# Driver Program
arr1 = [7, 1, 5, 2, 3, 6]
arr2 = [3, 8, 6, 20, 7]
n1 = len(arr1)
n2 = len(arr2)

# Function call
printUnion(arr1, arr2, n1, n2)
printIntersection(arr1, arr2, n1, n2)
```

Write a program to cyclically rotate an array by one.

```
def rotate(arr):
    n = len(arr)
    i = 0
    j = n - 1
    while i != j:
        arr[i], arr[j] = arr[j], arr[i]
        i = i + 1

# Driver function
arr= [1, 2, 3, 4, 5]
rotate(arr)
print(arr)
```

Find Largest sum contiguous Subarray [V. IMP] / Kadne's Algorithm




```
def maxSubArraySum(a):
    size=len(a)
    max_so_far =a[0]
    curr_max = a[0]

    for i in range(1,size):
        curr_max = max(a[i], curr_max + a[i])
        max_so_far = max(max_so_far,curr_max)
    return max_so_far

a = [-2, -3, 4, -1, -2, 1, 5, -3]
print ("Maximum contiguous sum is" , maxSubArraySum(a))
```

Minimise the maximum difference between heights [V.IMP]

```
"""
```

```
Given heights of n towers and a value k. We need to either increase or decrease the height of
every tower by k (only once) where k > 0. The task is to minimize the difference between the
heights of the longest and the shortest tower after modifications and output this difference.
Input : arr[] = {1, 5, 15, 10} k = 3
Output : Maximum difference is 8 arr[] = {4, 8, 12, 7}

"""
def getMinDiff(arr, k):
    arr.sort()
    n=len(arr)
    ans = arr[n - 1] - arr[0]  # Maximum possible height difference

    tempmin = arr[0]
    tempmax = arr[n - 1]

    for i in range(1, n):
        tempmin = min(arr[0] + k, arr[i] - k)

        # Minimum element when we add k to whole array Maximum element when we
        tempmax = max(arr[i - 1] + k, arr[n - 1] - k)

        # subtract k from whole array
        ans = min(ans, tempmax - tempmin)

    return ans

# Driver Code Starts
k = 6 # total towers
arr = [7, 4, 8, 8, 8, 9] # height of each array
print("Maximum difference of height between all towers (minimized as much as possible) is",
getMinDiff(arr, k))
```

Minimum no. of Jumps to reach end of an array

```
"""
Given an array of integers where each element represents the max number of steps that can be made
forward from that element. Write a function to return the minimum number of jumps to reach the end
of the array (starting from the first element). If an element is 0, then we cannot move through
that element. If we can't reach the end, return -1.

Input:  arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 -> 9)
"""
# Returns minimum number of jumps to reach arr[n-1] from arr[0]
def minJumps(arr, n):
    # The number of jumps needed to reach the starting index is 0
    if (n <= 1):
      return 0

    # Return -1 if not possible to jump
    if (arr[0] == 0):
      return -1

    # initialization
    # stores all time the maximal reachable index in the array
    maxReach = arr[0]
    # stores the amount of steps we can still take
    step = arr[0]
```

```
    # stores the amount of jumps necessary to reach that maximal reachable position
    jump = 1

    # Start traversing array

    for i in range(1, n):
        # Check if we have reached the end of the array
        if (i == n-1):
          return jump

        # updating maxReach
        maxReach = max(maxReach, i + arr[i])

        # we use a step to get to the current index
        step -= 1;

        # If no further steps left
        if (step == 0):
          # we must have used a jump
            jump += 1

          # Check if the current index / position or lesser index
          # is the maximum reach point from the previous indexes
            if(i >= maxReach):
                return -1

          # re-initialize the steps to the amount
          # of steps to reach maxReach from position i.
            step = maxReach - i;
    return -1


arr = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]
size = len(arr)
print("Minimum number of jumps to reach end is: ", minJumps(arr, size))
```

Find duplicate in an array of N+1 Integers

```
"""
Given a limited range array of size n containing elements between 1 and n-1 with one element
repeating, find the duplicate number in it without using any extra space. NOTE : ARRAY IS LIMITED
RANGE
"""
def findDuplicate(nums):
    actual_sum = sum(nums)
    expected_sum = len(nums) * (len(nums) - 1) // 2
    return actual_sum - expected_sum

A=[3,1,2,4,2]
print(findDuplicate(A))
```

Merge 2 sorted arrays without using Extra space.

```
def merge(X, Y):
    m = len(X)
    n = len(Y)

    # Consider each element `X[i]` of list `X[]` and ignore the element if it is
```

```
    # already in the correct order; otherwise, swap it with the next smaller
    # element, which happens to be the first element of `Y[]`.
    for i in range(m):

        # compare the current element of `X[]` with the first element of `Y[]`
        if X[i] > Y[0]:

            # swap `X[i] with `Y[0]`
            X[i],Y[0]=Y[0], X[i]

            first = Y[0]

            # move `Y[0]` to its correct position to maintain the sorted
            # order of `Y[]`. Note: `Y[1…n-1]` is already sorted
            k = 1
            while k < n and Y[k] < first:
                Y[k - 1] = Y[k]
                k = k + 1

            Y[k - 1] = first


X = [1, 4, 7, 8, 10]
Y = [2, 3, 9]
merge(X, Y)
print("X:", X)
print("Y:", Y)
```

Merge Intervals

```
def mergeIntervals(arr):

    # Sorting based on the increasing order
    # of the start intervals
    arr.sort(key=lambda x: x[0])

    # Stores index of last element in output array (modified arr[])
    index = 0

    # Traverse all input Intervals starting from second interval
    for i in range(1, len(arr)):

        # If this is not first Interval and overlaps with the previous one, Merge previous and
current Intervals
        if (arr[index][1] >= arr[i][0]):
            arr[index][1] = max(arr[index][1], arr[i][1])
        else:
            index = index + 1
            arr[index] = arr[i]

    print("The Merged Intervals are :", end=" ")
    for i in range(index+1):
        print(arr[i], end=" ")



arr = [[6, 8], [1, 3], [2, 4], [4, 7]]
mergeIntervals(arr)
```

Next Permutation

```
"""
If all digits sorted in descending order, then output is always "Not Possible". For example, 4321.
If all digits are sorted in ascending order, then we need to swap last two digits. For example,
1234.
For other cases, we need to process the number from rightmost side (why? because we need to find
the smallest of all greater numbers)
"""
def nextPermutation(arr):
    N=len(arr)
    ind = 0
    l = []
    l += arr
    for i in range(N-2, -1, -1):
        if l[i]<l[i+1]:
            ind = i
            break
    for i in range(N-1, ind, -1):
        if l[i]>l[ind]:
            l[i], l[ind] = l[ind], l[i]
            ind += 1
            break
    for i in range((N-ind)//2):
        l[i+ind], l[N-i-1] = l[N-i-1], l[i+ind]
    return "".join(l)

print(nextPermutation("218765"))
```

Count Inversion

```
def mergeSort(arr, n):
    # A temp_arr is created to store sorted array in merge function
    temp_arr = [0]*n
    return _mergeSort(arr, temp_arr, 0, n-1)

# This Function will use MergeSort to count inversions
def _mergeSort(arr, temp_arr, left, right):
    inv_count = 0

    # We will make a recursive call if and only if we have more than one elements
    if left < right:

        # mid is calculated to divide the array into two subarrays Floor division is must in case
of python
        mid = (left + right)//2

        # It will calculate inversion counts in the left subarray
        inv_count += _mergeSort(arr, temp_arr, left, mid)

        # It will calculate inversion counts in right subarray
        inv_count += _mergeSort(arr, temp_arr,mid + 1, right)

        # It will merge two subarrays in a sorted subarray
        inv_count += merge(arr, temp_arr, left, mid, right)
    return inv_count

# This function will merge two subarrays
```

```
# in a single sorted subarray


def merge(arr, temp_arr, left, mid, right):
    i = left     # Starting index of left subarray
    j = mid + 1  # Starting index of right subarray
    k = left     # Starting index of to be sorted subarray
    inv_count = 0

    # Conditions are checked to make sure that i and j don't exceed their subarray limits.
    while i <= mid and j <= right:

        # There will be no inversion if arr[i] <= arr[j]
        if arr[i] <= arr[j]:
            temp_arr[k] = arr[i]
            k += 1
            i += 1
        else:
            # Inversion will occur.
            temp_arr[k] = arr[j]
            inv_count += (mid-i + 1)
            k += 1
            j += 1

    # Copy the remaining elements of left subarray into temporary array
    while i <= mid:
        temp_arr[k] = arr[i]
        k += 1
        i += 1

    # Copy the remaining elements of right subarray into temporary array
    while j <= right:
        temp_arr[k] = arr[j]
        k += 1
        j += 1

    # Copy the sorted subarray into Original array
    for loop_var in range(left, right + 1):
        arr[loop_var] = temp_arr[loop_var]

    return inv_count

arr = [1, 20, 6, 4, 5]
n = len(arr)
result = mergeSort(arr, n)
print("Number of inversions are", result)
```

Best time to buy and Sell stock

```
"""
Input: [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
             Not 7-1 = 6, as selling price needs to be larger than buying price.
"""
def maxProfit(prices):
        max_profit = 0
        min_buy = float('inf')
        for price in prices:
```

```
            min_buy=min(min_buy, price)
            max_profit=max(max_profit, price-min_buy)
        return max_profit
print(maxProfit([7,1,5,3,6,4]))
```

Find all pairs on integer array whose sum is equal to given number

```
"""
An extended version of the two sum problem
"""
# Returns number of pairs in arr[0..n-1] with sum equal to 'sum'
def getPairsCount(arr, n, sum):
  unordered_map = {}
  count = 0
  for i in range(n):
    if sum - arr[i] in unordered_map:
      count += unordered_map[sum - arr[i]]
    if arr[i] in unordered_map:
      unordered_map[arr[i]] += 1
    else:
      unordered_map[arr[i]] = 1
  return count

# Driver code
arr = [1, 5, 7, -1, 5]
n = len(arr)
sum = 6
print('Count of pairs is', getPairsCount(arr, n, sum))
```

Find common elements In 3 sorted arrays

```
"""
Given three arrays sorted in non-decreasing order, print all common elements in these arrays.
"""
# Python function to print common elements in three sorted arrays
def findCommon(ar1, ar2, ar3, n1, n2, n3):

  # Initialize starting indexes for ar1[], ar2[] and ar3[]
  i, j, k = 0, 0, 0

  # Iterate through three arrays while all arrays have elements
  while (i < n1 and j < n2 and k < n3):

    # If x = y and y = z, print any of them and move ahead
    # in all arrays
    if (ar1[i] == ar2[j] and ar2[j] == ar3[k]):
      print (ar1[i])
      i += 1
      j += 1
      k += 1

    # x < y
    elif ar1[i] < ar2[j]:
      i += 1

    # y < z
```

```
    elif ar2[j] < ar3[k]:
      j += 1

    # We reach here when x > y and z < y, i.e., z is smallest
    else:
      k += 1


# Driver program to check above function
ar1 = [1, 5, 10, 20, 40, 80]
ar2 = [6, 7, 20, 80, 100]
ar3 = [3, 4, 15, 20, 30, 70, 80, 120]
n1 = len(ar1)
n2 = len(ar2)
n3 = len(ar3)

print ("Common elements are")
findCommon(ar1, ar2, ar3, n1, n2, n3)
```

Rearrange the array in alternating positive and negative items with O(1) extra space

```
def rearrange(arr, n):
  i = 0
  j = n - 1

  # shift all negative values to the end
  while (i < j):

    while (i <= n - 1 and arr[i] > 0):
      i += 1
    while (j >= 0 and arr[j] < 0):
      j -= 1

    if (i < j):
      temp = arr[i]
      arr[i] = arr[j]
      arr[j] = temp

  # i has index of leftmost
  # negativ ement
  if (i == 0 or i == n):
    return 0

  # start with first positive element at index 0  array in alternating positive & negative items
  k = 0
  while (k < n and i < n):

    # swap next positive element at even position from next negative element.
    arr[k], arr[i]= arr[i], arr[k]
    i = i + 1
    k = k + 2

arr = [2, 3,-4, -1, 6, -9 ]
n = len(arr)
rearrange(arr, n)
```

```
print("Rearranged array is", arr)
```

Find if there is any subarray with sum equal to 0

```
"""
Given an array of positive and negative numbers, find if there is a subarray (of size at-least
one) with 0 sum.

"""
#Function to check whether there is a subarray present with 0-sum or not.

def subArrayExists(arr):
    n=len(arr)
    #using set to store the prefix sum which has appeared already.
    s = set()

    sum = 0
    #iterating over the array.
    for i in range(n):
        #storing prefix sum.
        sum += arr[i]

        #if prefix sum is 0 or if it is already present in set then it is
  #repeated which means there is a subarray whose summation was 0, so we return true.
        if sum == 0 or sum in s:
            return True

        #storing every prefix sum obtained in set.
        s.add(sum)

    #returning false if we don't get any subarray with 0 sum.
    return False

print(subArrayExists([4, 2, -3, 1, 6]))
```

Find factorial of a large number

```
def range_prod(low,high):
    if low+1 < high:
        mid = (high+low)//2
        return range_prod(low,mid) * range_prod(mid+1,high)
    if low == high:
        return low
    return low*high

def factorial(n):
    if n < 2:
        return 1
    return range_prod(1,n)

print(factorial(12))
```

Find maximum product subarray

```
def maxProduct(arr):
    n=len(arr)
    # Variables to store maximum and minimum product till ith index.
```

```
    minVal = arr[0]
    maxVal = arr[0]
    maxProduct = arr[0]
    for i in range(1, n):
        # When multiplied by -ve number, maxVal becomes minVal and minVal becomes maxVal.
        if (arr[i] < 0):
            minVal, maxVal = maxVal, minVal
        # maxVal and minVal stores the product of subarray ending at arr[i].
        maxVal = max(arr[i], maxVal * arr[i])
        minVal = min(arr[i], minVal * arr[i])
        # Max Product of array.
        maxProduct = max(maxProduct, maxVal)
    return maxProduct

print(maxProduct([6, -3, -10, 0, 2]))
```

Find longest consecutive subsequence

```
"""
Given an array of integers, find the length of the longest sub-sequence such that elements in the
subsequence are consecutive integers, the consecutive numbers can be in any order.
"""
def findLongestConseqSubseq(arr):
    n=len(arr)
    #using a Set to store elements.
    s = set()
    ans=0

    #inserting all the array elements in Set.
    for ele in arr:
        s.add(ele)

    #checking each possible sequence from the start.
    for i in range(n):

        #if current element is starting element of a sequence then only we try to find out the
length of sequence.
        if (arr[i]-1) not in s:

            j=arr[i]
            #then we keep checking whether the next consecutive elements are present in Set and
         #we keep incrementing the counter.
            while(j in s):
                j+=1

            #storing the maximum count.
            ans=max(ans, j-arr[i])

    #returning the length of longest subsequence.
    return ans

print(findLongestConseqSubseq([1, 9, 3, 10, 4, 20, 2]))
```

Given an array of size n and a number k, find all elements that appear more than " n/k " times.

```python
"""
Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.
"""
def majorityElement(nums):
    if not nums:
        return []
    count1, count2, candidate1, candidate2 = 0, 0, None, None
    for x in nums:
        if candidate1 == x:
            count1 += 1
        elif candidate2 == x:
            count2 += 1
        elif count1 == 0:
            candidate1 = x
            count1 = 1
        elif count2 == 0:
            candidate2 = x
            count2 = 1
        else:
            count1 -= 1
            count2 -= 1
    res = []
    for c in [candidate1, candidate2]:
        if nums.count(c) > len(nums) // 3:
            res.append(c)
    return res
print(majorityElement([3,2,3]))
```

Maximum profit by buying and selling a share atmost twice

```python
"""
Input: prices = [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
"""
def maxProfit(prices):
    b1, b2= -float('inf'), -float('inf')
    s1, s2 = 0, 0
    for price in prices:
        s2 = max(s2, b2 + price)
        b2 = max(b2, s1 - price)
        s1 = max(s1, b1 + price)
        b1 = max(b1, -price)
    return s2

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

Find whether an array is a subset of another array

```python
def isSubset( a1, a2):
    n,m =len(a1), len(a2)
    s = set()
    for i in range(n) :
        s.add(a1[i])

    p = len(s)
    for i in range(m) :
```

```
        s.add(a2[i])
    if (len(s) == p) :
        return "Yes"
    return "No"

a=[11, 1, 13, 21, 3, 7]
b=[11, 3, 7, 1]
print(isSubset(a, b))
```

Find the triplet that sum to a given value

```
def findTriplets(arr, X):
    n=len(arr)
    found = False
    for i in range(0, n-2):
        for j in range(i+1, n-1):
            for k in range(j+1, n):
                if (arr[i] + arr[j] + arr[k] == X):
                    print(arr[i], arr[j], arr[k])
                    found = True

    # If no triplet with 0 sum found in array
    if (found == False):
        print("Three Sum not exist ")

arr = [0, -1, 2, -3, 1]
sum= 3
findTriplets(arr, sum)
```

Trapping Rain water problem

```
def trap(heights):

    # maintain two pointers left and right, pointing to the leftmost and
    # rightmost index of the input list
    (left, right) = (0, len(heights) - 1)
    water = 0

    maxLeft = heights[left]
    maxRight = heights[right]

    while left < right:
        if heights[left] <= heights[right]:
            left = left + 1
            maxLeft = max(maxLeft, heights[left])
            water += (maxLeft - heights[left])
        else:
            right = right - 1
            maxRight = max(maxRight, heights[right])
            water += (maxRight - heights[right])

    return water


heights = [7, 0, 4, 2, 5, 0, 6, 4, 0, 5]
print("The maximum amount of water that can be trapped is", trap(heights))
```

Chocolate Distribution problem

```
"""
Input : arr[] = {7, 3, 2, 4, 9, 12, 56} , m = 3
Output: Minimum Difference is 2
Explanation: We have seven packets of chocolates and we need to pick three packets for 3 students.
If we pick 2, 3 and 4, we get the minimum  difference between maximum and minimum packet sizes.
"""
# arr[0..n-1] represents sizes of packets
# m is number of students.
# Returns minimum difference between maximum
# and minimum values of distribution.
def findMinDiff(arr, m):
    n=len(arr)
    if (m==0 or n==0):
        return 0
    arr.sort()

    # Number of students cannot be more than number of packets
    if (n < m):
     return -1

    # Largest number of chocolates
    min_diff = arr[n-1] - arr[0]

    # Find the subarray of size m such that difference between last (maximum in case of sorted)
and
    #first (minimum in case of sorted) elements of subarray is minimum.
    for i in range(len(arr) - m + 1):
     min_diff = min(min_diff , arr[i + m - 1] - arr[i])

    return min_diff


arr = [12, 4, 7, 9, 2, 23, 25, 41, 30, 40, 28, 42, 30, 44, 48, 43, 50]
m = 7 # Number of students
print("Minimum difference is", findMinDiff(arr, m))
```

Smallest Subarray with sum greater than a given value

```
"""
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
"""
def minSubArrayLen(nums, target):
    i, j, pres, res = 0 , 0 , 0, len(nums) + 1
    while j < len(nums):
        pres += nums[j]; j += 1
        while pres >= target:
            res = min(res, j - i)
            pres -= nums[i]
            i+= 1
    return res if res != len(nums) + 1 else 0

sum=7
```

```
print(minSubArrayLen([2,3,1,2,4,3], sum))
```

Three way partitioning of an array around a given value

```
def threeWayPartition(arr, lowVal, highVal):
    n = len(arr)
    # Initialize ext available positions for smaller (than range) and greater elements
    start = 0
    end = n - 1
    i = 0

    # Traverse elements from left
    while i <= end:

        # If current element is smaller than range, put it on next available smaller position.
        if arr[i] < lowVal:
            arr[i], arr[start] = arr[start], arr[i]
            i += 1
            start += 1

        # If current element is greater than range, put it on next available greater position.
        elif arr[i] > highVal:
             arr[i], arr[end] = arr[end], arr[i]
             end -= 1
        else:
            i += 1

arr = [1, 14, 5, 20, 4, 2, 54, 20, 87, 98, 3, 1, 32]
"""
1) All elements smaller than lowVal come first.
2) All elements in range lowVal to highVal come next.
3) All elements greater than highVal appear in the end.
"""
threeWayPartition(arr, 10, 20)
print(arr)
```

Minimum swaps required bring elements less equal K together

```
"""
Find the minimum number of swaps required to bring all the numbers less than or equal to k
together, i.e. make them a contiguous subarray.
"""
def minSwap(arr, k) :
    n=len(arr)
    # Find count of elements which are less than equals to k
    count = 0
    for i in range(0, n) :
        if (arr[i] <= k) :
            count = count + 1

    # Find unwanted elements in current window of size 'count'
    bad = 0
    for i in range(0, count) :
        if (arr[i] > k) :
            bad = bad + 1

    # Initialize answer with 'bad' value of current window
```

```
    ans = bad
    j = count
    for i in range(0, n) :

        if(j == n) :
            break

        # Decrement count of previous window
        if (arr[i] > k) :
            bad = bad - 1

        # Increment count of current window
        if (arr[j] > k) :
            bad = bad + 1

        # Update ans if count of 'bad' is less in current window
        ans = min(ans, bad)
        j = j + 1

    return ans


arr = [2, 1, 5, 6, 3]
k = 3
print (minSwap(arr, k))

arr1 = [2, 7, 9, 5, 8, 7, 4]
k = 5
print (minSwap(arr1, k))
```

Minimum no. of operations required to make an array palindrome

```
"""
Input:  [6, 1, 3, 7]
Output: 1
Explanation: [6, 1, 3, 7] -> Merge 6 and 1 -> [7, 3, 7]
"""
def findMin(arr):

    # stores the minimum number of merge operations needed
    count = 0

    # `i` and `j` initially points to endpoints of the array
    i = 0
    j = len(arr) - 1

    # loop till the search space is exhausted
    while i < j:
        if arr[i] < arr[j]:
            # merge item at i'th index with the item at (i+1)'th index
            arr[i + 1] += arr[i]
            i = i + 1
            count = count + 1
        elif arr[i] > arr[j]:
            # merge item at (j-1)'th index with the item at j'th index
            arr[j - 1] += arr[j]
            j = j - 1
```

```
            count = count + 1
        # otherwise, ignore both the elements
        else:
            i = i + 1
            j = j - 1

    return count


arr = [6, 1, 4, 3, 1, 7]
min = findMin(arr)
print("The minimum number of operations required:", min)
```

Median of 2 sorted arrays of different size

```
def Solution(arr1, arr2):
    arr = arr1 + arr2
    arr.sort()
    n = len(arr)

    # If length of array is even
    if n % 2 == 0:
        return (arr[n // 2] + arr[n // 2 - 1]) / 2

    # If length of array is odd
    else:
        return arr[n//2]


arr1 = [ -5, 3, 6, 12, 15]
arr2 = [ -12, -10, -6, -3, 4, 10 ]
print("Median = ", Solution(arr1, arr2))
```

BackTracking

Rat in a maze Problem

```
"""
N = 4
m[][] = {{1, 0, 0, 0},
         {1, 1, 0, 1},
         {1, 1, 0, 0},
         {0, 1, 1, 1}}
Output:
DDRDRR DRDDRR
Explanation:
The rat can reach the destination at
(3, 3) from (0, 0) by two paths - DRDDRR
and DDRDRR, when printed in sorted order
we get DDRDRR DRDDRR.
"""
def setup():
    global v
    v = [[0 for i in range(100)] for j in range(100)]
    global ans
    ans = []
```

```
def path(arr, x, y, pth, n):
    if x==n-1 and y==n-1:
        global ans
        ans.append(pth)
        return
    global v
    if arr[x][y]==0 or v[x][y]==1:
        return
    v[x][y]=1
    if x>0:
        path(arr, x-1, y, pth+'U', n)
    if y>0:
        path(arr, x, y-1, pth+'L', n)
    if x<n-1:
        path(arr, x+1, y, pth+'D', n)
    if y<n-1:
        path(arr, x, y+1, pth+'R', n)
    v[x][y]=0

def findPath(m, n):
    global ans
    ans= []
    if m[0][0] == 0 or m[n-1][n-1]==0 :
        return ans
    setup()
    path(m, 0, 0, "", n)
    ans.sort()
    return ans

m = [ [ 1, 0, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 0, 1 ], [ 0, 0, 0, 0, 1 ],[ 0, 0, 0, 0, 1 ]
]
n = len(m)
print(findPath(m, n))
```

Printing all solutions in N-Queen Problem

```
def isSafe(mat, r, c):

    # return false if two queens share the same column
    for i in range(r):
        if mat[i][c] == 'Q':
            return False

    # return false if two queens share the same `` diagonal
    (i, j) = (r, c)
    while i >= 0 and j >= 0:
        if mat[i][j] == 'Q':
            return False
        i = i - 1
        j = j - 1

    # return false if two queens share the same `/` diagonal
    (i, j) = (r, c)
    while i >= 0 and j < len(mat):
        if mat[i][j] == 'Q':
            return False
        i = i - 1
        j = j + 1
```

```
    return True


def printSolution(mat):
    for r in mat:
        print(str(r).replace(',', '').replace('\'', ''))
    print()


def nQueen(mat, r):

    # if `N` queens are placed successfully, print the solution
    if r == len(mat):
        printSolution(mat)
        return

    # place queen at every square in the current row `r`
    # and recur for each valid movement
    for i in range(len(mat)):

        # if no two queens threaten each other
        if isSafe(mat, r, i):
            # place queen on the current square
            mat[r][i] = 'Q'

            # recur for the next row
            nQueen(mat, r + 1)

            # backtrack and remove the queen from the current square
            mat[r][i] = '-'


# `N × N` chessboard
N = 8

# `mat[][]` keeps track of the position of queens in
# the current configuration
mat = [['-' for x in range(N)] for y in range(N)]

nQueen(mat, 0)
```

Word Break Problem using Backtracking

```
# A recursive program to print all possible partitions of a given string into dictionary words

# A utility function to check whether a word is present in dictionary or not.  An array of strings
is used for dictionary.  Using array of strings for dictionary is definitely not a good idea. We
have used for simplicity of the program
def dictionaryContains(word):
    dictionary = {"mobile", "samsung", "sam", "sung", "man",
                  "mango", "icecream", "and", "go", "i", "love", "ice", "cream"}
    return word in dictionary

# Prints all possible word breaks of given string
def wordBreak(string):

    # Last argument is prefix
    wordBreakUtil(string, len(string), "")
```

```
# Result store the current prefix with spaces
# between words
def wordBreakUtil(string, n, result):

    # Process all prefixes one by one
    for i in range(1, n + 1):

        # Extract substring from 0 to i in prefix
        prefix = string[:i]

        # If dictionary contains this prefix, then
        # we check for remaining string. Otherwise
        # we ignore this prefix (there is no else for
        # this if) and try next
        if dictionaryContains(prefix):

            # If no more elements are there, print it
            if i == n:

                # Add this element to previous prefix
                result += prefix
                print(result)
                return
            wordBreakUtil(string[i:], n - i, result+prefix+" ")


print("First Test:")
wordBreak("iloveicecreamandmango")

print("\nSecond Test:")
wordBreak("ilovesamsungmobile")
```

Remove Invalid Parentheses

```
from collections import deque

def isValidString(string):
    left = 0
    right = 0
    index = 0

    while index < len(string):
        if string[index] == '(':
            left += 1

        elif string[index] == ')':
            if left > 0:
                left -= 1

            else:
                right += 1

            if right > left:
                return False

        index += 1
```

```
    return left == right

def removeInvalidParentheses(string):

    visited = set()
    result = []
    q = deque()
    valid = False

    visited.add(string)
    q.append(string)
    # BFS.
    while len(q) > 0:
        possibleAnswer = q.popleft()

        # Check whether 'possibleAnswer' is valid or not.
        if isValidString(possibleAnswer):
            result.append(possibleAnswer)
            valid = True

        # If true, then the solution exists at current level. No need to move at next state.
        if valid == True:
            continue

        # Generate all possible next state of Strings from current String.
        for i in range(len(possibleAnswer)):
            if possibleAnswer[i] == '(' or possibleAnswer[i] == ')':
                temp = possibleAnswer[0 : i] + possibleAnswer[i + 1 : len(possibleAnswer)]

                if temp not in visited:
                    q.append(temp)
                    visited.add(temp)

    return sorted(result)

print(removeInvalidParentheses(')(()))'))
print(removeInvalidParentheses('(((a)) ((a))()'))
```

Sudoku Solver

```
# N is the size of the 2D matrix N*N
N = 9

# A utility function to print grid
def printing(arr):
    for i in range(N):
        for j in range(N):
            print(arr[i][j], end = " ")
        print()

# Checks whether it will be legal to assign num to the given row, col
def isSafe(grid, row, col, num):

    # Check if we find the same num in the similar row , we return false
    for x in range(9):
        if grid[row][x] == num:
```

```
            return False

    # Check if we find the same num in the similar column , we return false
    for x in range(9):
        if grid[x][col] == num:
            return False

    # Check if we find the same num in the particular 3*3 matrix, we return false
    startRow = row - row % 3
    startCol = col - col % 3
    for i in range(3):
        for j in range(3):
            if grid[i + startRow][j + startCol] == num:
                return False
    return True

# Takes a partially filled-in grid and attempts to assign values to all unassigned locations in
# such a way to meet the requirements for Sudoku solution (non-duplication across rows, columns,
and boxes)
def solveSudoku(grid, row, col):

    # Check if we have reached the 8th row and 9th column (0 indexed matrix) , we are
    # returning true to avoid further backtracking
    if (row == N - 1 and col == N):
        return True

    # Check if column value becomes 9 , we move to next row and column start from 0
    if col == N:
        row += 1
        col = 0

    # Check if the current position of the grid already contains value >0, we iterate for next
column
    if grid[row][col] > 0:
        return solveSudoku(grid, row, col + 1)
    for num in range(1, N + 1, 1):

        # Check if it is safe to place the num (1-9) in the given row ,col ->we
        # move to next column
        if isSafe(grid, row, col, num):

            # Assigning the num in the current (row,col) position of the grid and assuming our
assigned
            # num in the position is correct
            grid[row][col] = num

            # Checking for next possibility with next column
            if solveSudoku(grid, row, col + 1):
                return True

        # Removing the assigned num , since our assumption was wrong , and we go for next
assumption with
      #diff num value
        grid[row][col] = 0
    return False


# 0 means unassigned cells
grid = [[3, 0, 6, 5, 0, 8, 4, 0, 0],
        [5, 2, 0, 0, 0, 0, 0, 0, 0],
```

```
        [0, 8, 7, 0, 0, 0, 0, 3, 1],
        [0, 0, 3, 0, 1, 0, 0, 8, 0],
        [9, 0, 0, 8, 6, 3, 0, 0, 5],
        [0, 5, 0, 0, 9, 0, 6, 0, 0],
        [1, 3, 0, 0, 0, 0, 2, 5, 0],
        [0, 0, 0, 0, 0, 0, 0, 7, 4],
        [0, 0, 5, 2, 0, 6, 3, 0, 0]]

if (solveSudoku(grid, 0, 0)):
    printing(grid)
else:
    print("no solution exists ")
```

m-Colouring Problem

```
"""
An array color[V] that should have numbers from 1 to m. color[i] should represent the color
assigned to the ith vertex.
The code should also return false if the graph cannot be colored with m colors.
"""

from queue import Queue

class node(object):
    color = 1
    edges = set()

def canPaint(nodes, n, m):

    # Create a visited array of n nodes, initialized to zero
    visited = [0 for _ in range(n+1)]

    # maxColors used till now are 1 as all nodes are painted color 1
    maxColors = 1

    # Do a full BFS traversal from all unvisited starting points
    for _ in range(1, n + 1):
        if visited[_]:
            continue

        # If the starting point is unvisited, mark it visited and push it in queue
        visited[_] = 1
        q = Queue()
        q.put(_)

        # BFS Travel starts here
        while not q.empty():
            top = q.get()

            # Checking all adjacent nodes to "top" edge in our queue
            for _ in nodes[top].edges:

                # IMPORTANT: If the color of the adjacent node is same, increase it by 1

                if nodes[top].color == nodes[_].color:
                    nodes[_].color += 1

                # If number of colors used shoots m, return 0
                maxColors = max(maxColors, max(
```

```
                        nodes[top].color, nodes[_].color))

                    if maxColors > m:
                        print(maxColors)
                        return 0

                    # If the adjacent node is not visited, mark it visited and push it in queue
                    if not visited[_]:
                        visited[_] = 1
                        q.put(_)

    return 1


n = 4
graph = [ [ 0, 1, 1, 1 ],
          [ 1, 0, 1, 0 ],
          [ 1, 1, 0, 1 ],
          [ 1, 0, 1, 0 ] ]

# Number of colors
m = 3

# Create a vector of n+1 nodes of type "node" The zeroth position is just dummy (1 to n to be
used)
nodes = []
for _ in range(n+1):
    nodes.append(node())

# Add edges to each node as per given input
for _ in range(n):
    for __ in range(n):
        if graph[_][__]:

            # Connect the undirected graph
            nodes[_].edges.add(_)
            nodes[__].edges.add(__)

# Display final answer
print(canPaint(nodes, n, m))
```

Print all palindromic partitions of a string

```
def isPalindrome(string: str,
                 low: int, high: int):
    while low < high:
        if string[low] != string[high]:
            return False
        low += 1
        high -= 1
    return True

# Recursive function to find all palindromic partitions of str[start..n-1]
# allPart --> A vector of vector of strings.
#             Every vector inside it stores a partition
# currPart --> A vector of strings to store current partition
def allPalPartUtil(allPart: list, currPart: list,
                   start: int, n: int, string: str):
```

```
    # If 'start' has reached len
    if start >= n:

        # In Python list are passed by reference that is why it is needed to copy first and then
append
        x = currPart.copy()

        allPart.append(x)
        return

    # Pick all possible ending points for substrings
    for i in range(start, n):

        # If substring str[start..i] is palindrome
        if isPalindrome(string, start, i):

            # Add the substring to result
            currPart.append(string[start:i + 1])

            # Recur for remaining substring
            allPalPartUtil(allPart, currPart,
                                i + 1, n, string)

            # Remove substring str[start..i] from current partition
            currPart.pop()

# Function to print all possible palindromic partitions of str.
# It mainly creates vectors and calls allPalPartUtil()
def allPalPartitions(string: str):

    n = len(string)

    # To Store all palindromic partitions
    allPart = []

    # To store current palindromic partition
    currPart = []

    # Call recursive function to generate all partitions and store in allPart
    allPalPartUtil(allPart, currPart, 0, n, string)

    # Print all partitions generated by above call
    for i in range(len(allPart)):
        for j in range(len(allPart[i])):
            print(allPart[i][j], end = " ")
        print()


string = "nitin"
allPalPartitions(string)
```

Subset Sum Problem

```
"""
Input : arr[] = {4, 1, 10, 12, 5, 2},
          sum = 9
Output : TRUE
{4, 5} is a subset with sum 9.
```

```
Input : arr[] = {1, 8, 2, 5},
          sum = 4
Output : FALSE
There exists no subset with sum 4.
"""
def isPossible(elements, target):

    dp = [False]*(target+1)

    # initializing with 1 as sum 0 is always possible
    dp[0] = True

    # loop to go through every element of the elements array
    for ele in elements:

        # to change the value o all possible sum values to True
        for j in range(target, ele - 1, -1):
            if dp[j - ele]:
                dp[j] = True

    # If target is possible return True else False
    return dp[target]

# Driver code
arr = [6, 2, 5]
target = 7

if isPossible(arr, target):
    print("YES")
else:
    print("NO")
```

The Knight's tour problem

```
# Chessboard Size
n = 6

def isSafe(x, y, board):
  '''
    A utility function to check if i,j are valid indexes
    for N*N chessboard
  '''
  if(x >= 0 and y >= 0 and x < n and y < n and board[x][y] == -1):
    return True
  return False


def printSolution(n, board):
  '''
    A utility function to print Chessboard matrix
  '''
  for i in range(n):
    for j in range(n):
      print(board[i][j], end=' ')
    print()


def solveKT(n):
  '''
```

```
    This function solves the Knight Tour problem using
    Backtracking. This function mainly uses solveKTUtil()
    to solve the problem. It returns false if no complete
    tour is possible, otherwise return true and prints the
    tour.
    Please note that there may be more than one solutions,
    this function prints one of the feasible solutions.
  '''

  # Initialization of Board matrix
  board = [[-1 for i in range(n)]for i in range(n)]

  # move_x and move_y define next move of Knight.
  # move_x is for next value of x coordinate
  # move_y is for next value of y coordinate
  move_x = [2, 1, -1, -2, -2, -1, 1, 2]
  move_y = [1, 2, 2, 1, -1, -2, -2, -1]

  # Since the Knight is initially at the first block
  board[0][0] = 0

  # Step counter for knight's position
  pos = 1

  # Checking if solution exists or not
  if(not solveKTUtil(n, board, 0, 0, move_x, move_y, pos)):
    print("Solution does not exist")
  else:
    printSolution(n, board)


def solveKTUtil(n, board, curr_x, curr_y, move_x, move_y, pos):
  '''
    A recursive utility function to solve Knight Tour
    problem
  '''

  if(pos == n**2):
    return True

  # Try all next moves from the current coordinate x, y
  for i in range(8):
    new_x = curr_x + move_x[i]
    new_y = curr_y + move_y[i]
    if(isSafe(new_x, new_y, board)):
      board[new_x][new_y] = pos
      if(solveKTUtil(n, board, new_x, new_y, move_x, move_y, pos+1)):
        return True

      # Backtracking
      board[new_x][new_y] = -1
  return False

solveKT(n)
```

Tug of War

```
# function that tries every possible solution by calling itself recursively
def TOWUtil(arr, n, curr_elements, no_of_selected_elements,
```

```
      soln, min_diff, Sum, curr_sum, curr_position):

  # checks whether the it is going out of bound
  if (curr_position == n):
    return

  # checks that the numbers of elements left are not less than the number of elements required to
form the solution
  if ((int(n / 2) - no_of_selected_elements) >(n - curr_position)):
    return

  # consider the cases when current element is not included in the solution
  TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln, min_diff, Sum, curr_sum,
curr_position + 1)

  # add the current element to the solution
  no_of_selected_elements += 1
  curr_sum = curr_sum + arr[curr_position]
  curr_elements[curr_position] = True

  # checks if a solution is formed
  if (no_of_selected_elements == int(n / 2)):

    # checks if the solution formed is better than the best solution so far
    if (abs(int(Sum / 2) - curr_sum) < min_diff[0]):
      min_diff[0] = abs(int(Sum / 2) - curr_sum)
      for i in range(n):
        soln[i] = curr_elements[i]
  else:

    # consider the cases where current element is included in the solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln, min_diff, Sum, curr_sum,
curr_position + 1)

  # removes current element before returning
  # to the caller of this function
  curr_elements[curr_position] = False

# main function that generate an arr
def tugOfWar(arr, n):

  # the boolean array that contains the inclusion and exclusion of an element
  # in current set. The number excluded automatically form the other set
  curr_elements = [None] * n

  # The inclusion/exclusion array for final solution
  soln = [None] * n

  min_diff = [999999999999]
  Sum = 0

  for i in range(n):
    Sum += arr[i]
    curr_elements[i] = soln[i] = False

  # Find the solution using recursive function TOWUtil()
  TOWUtil(arr, n, curr_elements, 0, soln, min_diff, Sum, 0, 0)

  # Print the solution
  print("The first subset is: ")
```

```
  for i in range(n):
    if (soln[i] == True):
      print(arr[i], end = " ")
  print()
  print("The second subset is: ")
  for i in range(n):
    if (soln[i] == False):
      print(arr[i], end = " ")


arr = [23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4]
n = len(arr)
tugOfWar(arr, n)
```

Find shortest safe route in a path with landmines

```
# Python3 program to find shortest safe Route
# in the matrix with landmines
import sys

R = 12
C = 10

# These arrays are used to get row and column
# numbers of 4 neighbours of a given cell
rowNum = [ -1, 0, 0, 1 ]
colNum = [ 0, -1, 1, 0 ]

min_dist = sys.maxsize

# A function to check if a given cell (x, y)
# can be visited or not
def isSafe(mat, visited, x, y):

  if (mat[x][y] == 0 or visited[x][y]):
    return False

  return True

# A function to check if a given cell (x, y) is
# a valid cell or not
def isValid(x, y):

  if (x < R and y < C and x >= 0 and y >= 0):
    return True

  return False

# A function to mark all adjacent cells of
# landmines as unsafe. Landmines are shown with
# number 0
def markUnsafeCells(mat):

  for i in range(R):
    for j in range(C):
      # If a landmines is found
      if (mat[i][j] == 0):
        # Mark all adjacent cells
```

```
        for k in range(4):
          if (isValid(i + rowNum[k], j + colNum[k])):
            mat[i + rowNum[k]][j + colNum[k]] = -1

  # Mark all found adjacent cells as unsafe
  for i in range(R):
    for j in range(C):
      if (mat[i][j] == -1):
        mat[i][j] = 0

  print(mat)
"""
      for i in range(R):
          for j in range(C):
              print(mat[i][j], end='')
          print()
"""

# Function to find shortest safe Route in the matrix with landmines
# mat[][] - binary input matrix with safe cells marked as 1
# visited[][] - store info about cells already visited in current route
# (i, j) are coordinates of the current cell
# min_dist --> stores minimum cost of shortest path so far
# dist --> stores current path cost

def findShortestPathUtil(mat, visited, i, j, dist):
  global min_dist

  # If destination is reached
  if (j == C - 1):
    # Update shortest path found so far
    min_dist = min(dist, min_dist)
    return

  # If current path cost exceeds minimum so far
  if (dist > min_dist):
    return

  # include (i, j) in current path
  visited[i][j] = True

  # Recurse for all safe adjacent neighbours
  for k in range(4):
    if (isValid(i + rowNum[k], j + colNum[k]) and isSafe(mat, visited, i + rowNum[k], j +
colNum[k])):
      findShortestPathUtil(mat, visited, i + rowNum[k], j + colNum[k], dist + 1)

  # Backtrack
  visited[i][j] = False

# A wrapper function over findshortestPathUtil()
def findShortestPath(mat):

  global min_dist

  # Stores minimum cost of shortest path so far
  min_dist = sys.maxsize

  # Create a boolean matrix to store info about
  # cells already visited in current route
```

```
  visited = [[False for i in range(C)] for j in range(R)]

  # Mark adjacent cells of landmines as unsafe
  markUnsafeCells(mat)

  # Start from first column and take minimum
  for i in range(R):
    # If path is safe from current cell
    if (mat[i][0] == 1):
      # Find shortest route from (i, 0) to any
      # cell of last column (x, C - 1) where
      # 0 <= x < R
      findShortestPathUtil(mat, visited, i, 0, 0)

      # If min distance is already found
      if (min_dist == C - 1):
        break

  # If destination can be reached
  if (min_dist != sys.maxsize):
    print("Length of shortest safe route is", min_dist)
  else:
    # If the destination is not reachable
    print("Destination not reachable from given source")

mat = [
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 0, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 ],
    [ 1, 0, 1, 1, 1, 1, 1, 1, 0, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 0, 1, 1, 1, 1, 0, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 ] ]

# Find shortest path
findShortestPath(mat)
```

Combinational Sum

```
"""Find all combinations that sum to a given value.
Input : arr[] = 2, 4, 6, 8
            x = 8
Output : [2, 2, 2, 2]
         [2, 2, 4]
         [2, 6]
         [4, 4]
         [8]
"""
def combinationSum(arr, sum):
  ans = []
  temp = []

  # first do hashing nothing but set{} since set does not always sort removing the duplicates
  # using Set and Sorting the List
```

```
  arr = sorted(list(set(arr)))
  findNumbers(ans, arr, temp, sum, 0)
  return ans

def findNumbers(ans, arr, temp, sum, index):

  if(sum == 0):

    # Adding deep copy of list to ans
    ans.append(list(temp))
    return

  # Iterate from index to len(arr) - 1
  for i in range(index, len(arr)):

    # checking that sum does not become negative
    if(sum - arr[i]) >= 0:

      # adding element which can contribute to
      # sum
      temp.append(arr[i])
      findNumbers(ans, arr, temp, sum-arr[i], i)

      # removing element from list (backtracking)
      temp.remove(arr[i])


arr = [2, 4, 6, 8]
sum = 8
ans = combinationSum(arr, sum)

# If result is empty, then
if len(ans) <= 0:
  print("empty")

# print all combinations stored in ans
for i in range(len(ans)):

  print("(", end=' ')
  for j in range(len(ans[i])):
    print(str(ans[i][j])+" ", end=' ')
  print(")", end=' ')
```

Find Maximum number possible by doing at-most K swaps

```
"""
Given a positive integer, find the maximum integer possible by doing at-most K swap operations on
its digits.
Examples:
Input: M = 254, K = 1
Output: 524
Swap 5 with 2 so number becomes 524
"""

# function to find maximum integer possible by doing at-most K swap operations on its digits
def findMaximumNum(string, k, maxm, ctr):

  # return if no swaps left
```

```
  if k == 0:
    return

  n = len(string)
  # Consider every digit after the cur position
  mx = string[ctr]

  for i in range(ctr+1,n):
    # Find maximum digit greater than at ctr among rest
    if int(string[i]) > int(mx):
      mx=string[i]

  # If maxm is not equal to str[ctr], decrement k
  if(mx!=string[ctr]):
    k=k-1

  # search this maximum among the rest from behind first swap the last maximum digit if it occurs
more then 1 time
  # example str= 1293498 and k=1 then max string is 9293418 instead of 9213498
  for i in range(ctr,n):
    # If digit equals maxm swap the digit with current digit and recurse for the rest
    if(string[i]==mx):
      # swap str[ctr] with str[j]
      string[ctr], string[i] = string[i], string[ctr]
      new_str = "".join(string)
      # If current num is more than maximum so far
      if int(new_str) > int(maxm[0]):
        maxm[0] = new_str

      # recurse of the other k - 1 swaps
      findMaximumNum(string, k , maxm, ctr+1)

      # backtrack
      string[ctr], string[i] = string[i], string[ctr]


string = "129814999"
k = 4
maxm = [string]
string = [char for char in string]
findMaximumNum(string, k, maxm, 0)
print(maxm[0])
```

Print all permutations of a string

```
def permute(s, answer):
  if (len(s) == 0):
    print(answer, end = " ")
    return

  for i in range(len(s)):
    ch = s[i]
    left_substr = s[0:i]
    right_substr = s[i + 1:]
    rest = left_substr + right_substr
    permute(rest, answer + ch)

answer=""
s = "alex"
```

```
print("All possible strings are : ")
permute(s, answer)
```

Find if there is a path of more than k length from a source

```
# Program to find if there is a simple path with weight more than k

# This class represents a dipathted graph using adjacency list representation
class Graph:
  # Allocates memory for adjacency list
  def __init__(self, V):
    self.V = V
    self.adj = [[] for i in range(V)]

  # Returns true if graph has path more than k length
  def pathMoreThanK(self,src, k):
    # Create a path array with nothing included in path
    path = [False]*self.V

    # Add source vertex to path
    path[src] = 1

    return self.pathMoreThanKUtil(src, k, path)

  # Prints shortest paths from src to all other vertices
  def pathMoreThanKUtil(self,src, k, path):
    # If k is 0 or negative, return true
    if (k <= 0):
      return True

    # Get all adjacent vertices of source vertex src and recursively explore all paths from src.
    i = 0
    while i != len(self.adj[src]):
      # Get adjacent vertex and weight of edge
      v = self.adj[src][i][0]
      w = self.adj[src][i][1]
      i += 1

      # If vertex v is already there in path, then there is a cycle (we ignore this edge)
      if (path[v] == True):
        continue

      # If weight of is more than k, return true
      if (w >= k):
        return True

      # Else add this vertex to path
      path[v] = True

      # If this adjacent can provide a path longer than k, return true.
      if (self.pathMoreThanKUtil(v, k-w, path)):
        return True

      # Backtrack
      path[v] = False

    # If no adjacent could produce longer path, return false
    return False
```

```
  # Utility function to an edge (u, v) of weight w
  def addEdge(self,u, v, w):
    self.adj[u].append([v, w])
    self.adj[v].append([u, w])


# create the graph given in above figure
V = 9
g = Graph(V)
# making above shown graph
g.addEdge(0, 1, 4)
g.addEdge(0, 7, 8)
g.addEdge(1, 2, 8)
g.addEdge(1, 7, 11)
g.addEdge(2, 3, 7)
g.addEdge(2, 8, 2)
g.addEdge(2, 5, 4)
g.addEdge(3, 4, 9)
g.addEdge(3, 5, 14)
g.addEdge(4, 5, 10)
g.addEdge(5, 6, 2)
g.addEdge(6, 7, 1)
g.addEdge(6, 8, 6)
g.addEdge(7, 8, 7)

#calling in the function
src = 0
k = 62
print("Yes") if g.pathMoreThanK(src, k) else print("No")
k = 60
print("Yes") if g.pathMoreThanK(src, k) else print("No")
```

Longest Possible Route in a Matrix with Hurdles

```
# Python program to find Longest Possible Route in a matrix with hurdles
import sys
R,C = 3,10

# A Pair to store status of a cell. found is set to True of destination is reachable and value
stores
# distance of longest path
class Pair:
  def __init__(self, found, value):
    self.found = found
    self.value = value

# Function to find Longest Possible Route in the matrix with hurdles. If the destination is not
reachable
# the function returns false with cost sys.maxsize. (i, j) is source cell and (x, y) is
destination cell.
def findLongestPathUtil(mat, i, j, x, y, visited):

  # if (i, j) itself is destination, return True
  if (i == x and j == y):
    p = Pair( True, 0 )
    return p
```

```
  # if not a valid cell, return false
  if (i < 0 or i >= R or j < 0 or j >= C or mat[i][j] == 0 or visited[i][j]) :
    p = Pair( False, sys.maxsize )
    return p

  # include (i, j) in current path i.e. set visited(i, j) to True
  visited[i][j] = True

  # res stores longest path from current cell (i, j) to destination cell (x, y)
  res = -sys.maxsize -1

  # go left from current cell
  sol = findLongestPathUtil(mat, i, j - 1, x, y, visited)

  # if destination can be reached on going left from current cell, update res
  if (sol.found):
    res = max(res, sol.value)

  # go right from current cell
  sol = findLongestPathUtil(mat, i, j + 1, x, y, visited)

  # if destination can be reached on going right from current cell, update res
  if (sol.found):
    res = max(res, sol.value)

  # go up from current cell
  sol = findLongestPathUtil(mat, i - 1, j, x, y, visited)

  # if destination can be reached on going up from current cell, update res
  if (sol.found):
    res = max(res, sol.value)

  # go down from current cell
  sol = findLongestPathUtil(mat, i + 1, j, x, y, visited)

  # if destination can be reached on going down from current cell, update res
  if (sol.found):
    res = max(res, sol.value)

  # Backtrack
  visited[i][j] = False

  # if destination can be reached from current cell, return True
  if (res != -sys.maxsize -1):
    p = Pair( True, 1 + res )
    return p

  # if destination can't be reached from current cell, return false
  else:
    p = Pair( False, sys.maxsize )
    return p

# A wrapper function over findLongestPathUtil()
def findLongestPath(mat, i, j, x,y):

  # create a boolean matrix to store info about cells already visited in current route initialize
visited to false
  visited = [[False for i in range(C)]for j in range(R)]

  # find longest route from (i, j) to (x, y) and print its maximum cost
```

```
  p = findLongestPathUtil(mat, i, j, x, y, visited)
  if (p.found):
    print("Length of longest possible route is ",str(p.value))

  # If the destination is not reachable
  else:
    print("Destination not reachable from given source")

# input matrix with hurdles shown with number 0
mat = [ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 0, 1, 1, 0, 1, 1, 0, 1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]

# find longest path with source (0, 0) and destination (1, 7)
findLongestPath(mat, 0, 0, 1, 7)
```

Print all possible paths from top left to bottom right of a mXn matrix

```
"""
The problem is to print all the possible paths from top left to bottom right of a mXn matrix with
the constraints that from each cell you can either move only to right or down.

Examples :

Input : 1 2 3
        4 5 6
Output : 1 4 5 6
         1 2 5 6
         1 2 3 6

Input : 1 2
        3 4
Output : 1 2 4
         1 3 4
"""

def printAllPaths(M, m, n):
  mapping = {}
  if not mapping.get((m,n)):
    if m == 1 and n == 1:
      return [M[m-1][n-1]]
    else:
      res = []
      if n > 1:
        a = printAllPaths(M, m, n-1)
        for i in a:
          if not isinstance(i, list):
            i = [i]
          res.append(i+[M[m-1][n-1]])
      if m > 1:
        b =printAllPaths(M, m-1, n)
        for i in b:
          if not isinstance(i, list):
            i = [i]
          res.append(i+[M[m-1][n-1]])
    mapping[(m,n)] = res
  return mapping.get((m,n))
```

```
M = [[1, 2, 3], [4, 5, 6], [7,8,9]]
m, n = len(M), len(M[0])
res = printAllPaths(M, m, n)
for i in res:
  print(i)
```

Partition of a set intoK subsets with equal sum

```
"""
Input : arr = [2, 1, 4, 5, 6], K = 3
Output : Yes
we can divide above array into 3 parts with equal
sum as [[2, 4], [1, 5], [6]]

Input  : arr = [2, 1, 5, 5, 6], K = 3
Output : No
It is not possible to divide above array into 3
parts with equal sum
"""

"""
array   - given input array
subsetSum array - sum to store each subset of the array
taken   -boolean array to check whether element
is taken into sum partition or not
K     - number of partitions needed
N     - total number of element in array
curIdx   - current subsetSum index
limitIdx   - lastIdx from where array element should
be taken """

def isKPartitionPossibleRec(arr, subsetSum, taken, subset, K, N, curIdx, limitIdx):
  if subsetSum[curIdx] == subset:

    """
    current index (K - 2) represents (K - 1)
    subsets of equal sum last partition will
    already remain with sum 'subset'
    """
    if (curIdx == K - 2):
      return True

    # recursive call for next subsetition
    return isKPartitionPossibleRec(arr, subsetSum, taken,
                  subset, K, N, curIdx + 1 , N - 1)

  # start from limitIdx and include elements into current partition
  for i in range(limitIdx, -1, -1):

    # if already taken, continue
    if (taken[i]):
      continue
    tmp = subsetSum[curIdx] + arr[i]

    # if temp is less than subset, then only include the element and call recursively
    if (tmp <= subset):
```

```
      # mark the element and include into current partition sum
      taken[i] = True
      subsetSum[curIdx] += arr[i]
      nxt = isKPartitionPossibleRec(arr, subsetSum, taken,
                    subset, K, N, curIdx, i - 1)

      # after recursive call unmark the element and remove from subsetition sum
      taken[i] = False
      subsetSum[curIdx] -= arr[i]
      if (nxt):
        return True
  return False

# Method returns True if arr can be partitioned into K subsets with equal sum
def isKPartitionPossible(arr, N, K):

  # If K is 1, then complete array will be our answer
  if (K == 1):
    return True

  # If total number of partitions are more than N, then division is not possible
  if (N < K):
    return False

  # if array sum is not divisible by K then we can't divide array into K partitions
  sum = 0
  for i in range(N):
    sum += arr[i]
  if (sum % K != 0):
    return False

  # the sum of each subset should be subset (= sum / K)
  subset = sum // K
  subsetSum = [0] * K
  taken = [0] * N

  # Initialize sum of each subset from 0
  for i in range(K):
    subsetSum[i] = 0

  # mark all elements as not taken
  for i in range(N):
    taken[i] = False

  # initialize first subset sum as last element of array and mark that as taken
  subsetSum[0] = arr[N - 1]
  taken[N - 1] = True

  # call recursive method to check K-substitution condition
  return isKPartitionPossibleRec(arr, subsetSum, taken,
              subset, K, N, 0, N - 1)

arr = [2, 1, 4, 5, 3, 3 ]
N = len(arr)
K = 3
if (isKPartitionPossible(arr, N, K)):
  print("Partitions into equal sum is possible.\n")
else:
  print("Partitions into equal sum is not possible.\n")
```

Find the K-th Permutation Sequence of first N natural numbers

```
"""
Input: N = 3, K = 4
Output: 231
Explanation:
The ordered list of permutation sequence from integer 1 to 3 is : 123, 132, 213, 231, 312, 321.
So, the 4th permutation sequence is “231”.

Input: N = 2, K = 1
Output: 12
Explanation:
For n = 2, only 2 permutations are possible 12 21. So, the 1st permutation sequence is “12”.
"""


# Function to find the index of number at first position of kth sequence of set of size n
def findFirstNumIndex(k, n):
  if (n == 1):
    return 0, k
  n -= 1
  first_num_index = 0

  # n_actual_fact = n!
  n_partial_fact = n

  while (k >= n_partial_fact and n > 1):
    n_partial_fact = n_partial_fact * (n - 1)
    n -= 1

  # First position of the kth sequence will be occupied by the number present at index = k / (n-
1)!
  first_num_index = k // n_partial_fact
  k = k % n_partial_fact
  return first_num_index, k

# Function to find the kth permutation of n numbers
def findKthPermutation(n, k):

  # Store final answer
  ans = ""
  s = set()

  # Insert all natural number upto n in set
  for i in range(1, n + 1):
    s.add(i)

  # Subtract 1 to get 0 based indexing
  k = k - 1

  for i in range(n):

    # Mark the first position
    itr = list(s)

    index, k = findFirstNumIndex(k, n - i)
```

```
    # itr now points to the number at index in set s
    ans += str(itr[index])

    # remove current number from the set
    itr.pop(index)

    s = set(itr)

  return ans

n = 3
k = 4
kth_perm_seq = findKthPermutation(n, k)
print(kth_perm_seq)
```

Binary Trees

level order traversal AKA BFS

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


def printLevelOrder(root):
  # Base Case
  if root is None:
    return

  # Create an empty queue for level order traversal
  queue = []

  # Enqueue Root and initialize height
  queue.append(root)

  while(len(queue) > 0):

    # Print front of queue and remove it from queue
    print(queue[0].data)
    node = queue.pop(0)

    # Enqueue left child
    if node.left is not None:
      queue.append(node.left)

    # Enqueue right child
    if node.right is not None:
      queue.append(node.right)


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

```
root.left.left = Node(4)
root.left.right = Node(5)

print("Level Order Traversal of binary tree is -")
printLevelOrder(root)
```

Reverse Level Order traversal

```
from collections import deque

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

def reverseLevelOrder(root):
  q = deque()
  q.append(root)
  ans = deque()
  while q:
    node = q.popleft()
    if node is None:
      continue

    ans.appendleft(node.data)

    if node.right:
      q.append(node.right)

    if node.left:
      q.append(node.left)

  return ans

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

print ("Level Order traversal of binary tree is", reverseLevelOrder(root))
```

Height of a tree

```
"""
Given a binary tree, find height of it. Height of empty tree is -1, height of tree with one node
is 0
"""
class Node:

  # Constructor to create a new node
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None
```

```
# Compute the "maxDepth" of a tree -- the number of nodes along the longest path from the root
node down to the
# farthest leaf node
def maxDepth(node):
  if node is None:
    return 0 ;

  else :

    # Compute the depth of each subtree
    lDepth = maxDepth(node.left)
    rDepth = maxDepth(node.right)

    # Use the larger one
    if (lDepth > rDepth):
      return lDepth+1
    else:
      return rDepth+1


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

print ("Height of tree is %d" %(maxDepth(root)))
```

Diameter of a tree

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

def height(node):

  # Base Case : Tree is empty
  if node is None:
    return 0

  # If tree is not empty then height = 1 + max of left
  # height and right heights
  return 1 + max(height(node.left), height(node.right))

# Function to get the diameter of a binary tree
def diameter(root):

  # Base Case when tree is empty
  if root is None:
    return 0

  # Get the height of left and right sub-trees
  lheight = height(root.left)
  rheight = height(root.right)

  # Get the diameter of left and right sub-trees
```

```
  ldiameter = diameter(root.left)
  rdiameter = diameter(root.right)

  # Return max of the following tree:
  # 1) Diameter of left subtree
  # 2) Diameter of right subtree
  # 3) Height of left subtree + height of right subtree +1
  return max(lheight + rheight + 1, max(ldiameter, rdiameter))

"""
Constructed binary tree is
      1
    / \
    2   3
  / \
  4   5
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print(diameter(root))
```

Mirror of a tree / Invert Binary Tree

```
from collections import deque

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


# Function to perform preorder traversal on a given binary tree
def preorder(root):
    if root is None:
        return
    print(root.data, end=' ')
    preorder(root.left)
    preorder(root.right)

# Utility function to swap left subtree with right subtree
def swap(root):
    if root is None:
        return
    temp = root.left
    root.left = root.right
    root.right = temp


# Iterative function to invert a given binary tree using a queue
def invertBinaryTree(root):
    # base case: if the tree is empty
    if root is None:
        return

    # maintain a queue and push the root node
```

```
    q = deque()
    q.append(root)

    # loop till queue is empty
    while q:

        # dequeue front node
        curr = q.popleft()

        # swap the left child with the right child
        swap(curr)

        # enqueue left child of the popped node
        if curr.left:
            q.append(curr.left)

        # enqueue right child of the popped node
        if curr.right:
            q.append(curr.right)


'''
Construct the following tree
          1
        /   \
       /     \
      2       3
     / \     / \
    4   5   6   7
'''

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)

invertBinaryTree(root)
preorder(root)
```

Inorder, Preorder and Postorder Tree Traversal (Recursive Method)

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

def preorder(root):
    if root is None:
        return
    print(root.data, end=' ')
    preorder(root.left)
    preorder(root.right)
```

```
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.data, end=' ')
    inorder(root.right)

def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.data, end=' ')


''' Construct the following tree
           1
         /   \
        /     \
       2       3
      /      /   \
     /      /     \
    4      5       6
          / \
         /   \
        7     8
'''

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)

print("Preorder: ",preorder(root))
print("Inorder: ",inorder(root))
print("PostOrder: ",postorder(root))
```

Left View of a tree

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


def leftView(root, level=1, last_level=0):
    # base case: empty tree
    if root is None:
        return last_level

    # if the current node is the first node of the current level
    if last_level < level:

        # print the node's data
```

```
        print(root.key, end=' ')

        # update the last level to the current level
        last_level = level

    # recur for the left and right subtree by increasing the level by 1
    last_level = leftView(root.left, level + 1, last_level)
    last_level = leftView(root.right, level + 1, last_level)
    return last_level

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
leftView(root)
```

Right View of Tree

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


def RightView(root, level=1, lastLevel=0):
    if root is None:
        return lastLevel

    # if the current node is the last node of the current level
    if lastLevel < level:

        # print the node's data
        print(root.key, end=' ')

        # update the last level to the current level
        lastLevel = level

    # recur for the right and left subtree by increasing level by 1
    lastLevel = RightView(root.right, level + 1, lastLevel)
    lastLevel = RightView(root.left, level + 1, lastLevel)
    return lastLevel

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
RightView(root)
```

Top View of a tree

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

# Recursive function to perform preorder traversal on the tree and fill the dictionary.
# Here, the node has `dist` horizontal distance from the tree's root, and the level represents the
node's level.
def printTop(root, dist, level, d):
    if root is None:
        return

    # if the current level is less than the maximum level seen so far for the same horizontal
distance, or if
    # the horizontal distance is seen for the first time, update the dictionary
    if dist not in d or level < d[dist][1]:
        # update value and level for current distance
        d[dist] = (root.key, level)

    # recur for the left subtree by decreasing horizontal distance and increasing level by 1
    printTop(root.left, dist - 1, level + 1, d)

    # recur for the right subtree by increasing both level and horizontal distance by 1
    printTop(root.right, dist + 1, level + 1, d)


def printTopView(root):

    # create a dictionary where
    # key -> relative horizontal distance of the node from the root node, and
    # value -> pair containing the node's value and its level
    d = {}

    # perform preorder traversal on the tree and fill the dictionary
    printTop(root, 0, 0, d)

    # traverse the dictionary in sorted order of keys and print the top view
    for key in sorted(d.keys()):
        print(d.get(key)[0], end=' ')

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
printTopView(root)
```

Bottom View of a tree

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

```python
# Recursive function to perform preorder traversal on the tree and fill the map.
# Here, the node has `dist` horizontal distance from the tree's root, and the `level` represents
the node's level.
def printBottom(root, dist, level, d):

    # base case: empty tree
    if root is None:
        return

    # if the current level is more than or equal to the maximum level seen so far for the same
horizontal distance
    # or horizontal distance is seen for the first time, update the dictionary
    if dist not in d or level >= d[dist][1]:
        # update value and level for the current distance
        d[dist] = (root.key, level)

    # recur for the left subtree by decreasing horizontal distance and increasing level by 1
    printBottom(root.left, dist - 1, level + 1, d)

    # recur for the right subtree by increasing both level and horizontal distance by 1
    printBottom(root.right, dist + 1, level + 1, d)


# Function to print the bottom view of a given binary tree
def printBottomView(root):

    # create a dictionary where
    # key -> relative horizontal distance of the node from the root node, and
    # value -> pair containing the node's value and its level
    d = {}

    # perform preorder traversal on the tree and fill the dictionary
    printBottom(root, 0, 0, d)

    # traverse the dictionary in sorted order of their keys and print the bottom view
    for key in sorted(d.keys()):
        print(d.get(key)[0], end=' ')


root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.right = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
printBottomView(root)
```

Zig-Zag traversal of a binary tree

```python
from collections import deque

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

```python
# Traverse the tree in a preorder fashion and store nodes in a dictionary corresponding to their
level
def preorder(root, level, d):
    if root is None:
        return

    # insert the current node and its level into the dictionary
    # if the level is odd, insert at the back; otherwise, search at front
    if level % 2 == 1:
        d.setdefault(level, deque()).append(root.key)
    else:
        d.setdefault(level, deque()).appendleft(root.key)

    # recur for the left and right subtree by increasing the level by 1
    preorder(root.left, level + 1, d)
    preorder(root.right, level + 1, d)


# Recursive function to print spiral order traversal of a given binary tree
def SpiralOrderTraversal(root):

    # create an empty dictionary to store nodes between given levels
    d = {}

    # traverse the tree and insert its nodes into the dictionary corresponding to their level
    preorder(root, 1, d)

    # iterate through the dictionary and print all nodes present at every level
    for i in range(1, len(d) + 1):
        print(f'Level {i}:', list(d[i]))


root = Node(15)
root.left = Node(10)
root.right = Node(20)
root.left.left = Node(8)
root.left.right = Node(12)
root.right.left = Node(16)
root.right.right = Node(25)
root.left.left.left = Node(20)
root.right.right.right = Node(30)
SpiralOrderTraversal(root)
```

Check if a tree is balanced or not

```python
"""
Given a binary tree, write an efficient algorithm to check if it is height-balanced or not. In a
height-balanced tree, the absolute difference between the height of the left and right subtree for
every node is 0 or 1.
"""
class Node:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right
```

```
# Recursive function to check if a given binary tree is height-balanced or not
def isHeightBalanced(root, isBalanced=True):

    # base case: tree is empty or not balanced
    if root is None or not isBalanced:
        return 0, isBalanced

    # get the height of the left subtree
    left_height, isBalanced = isHeightBalanced(root.left, isBalanced)

    # get the height of the right subtree
    right_height, isBalanced = isHeightBalanced(root.right, isBalanced)

    # tree is unbalanced if the absolute difference between the height of
    # its left and right subtree is more than 1
    if abs(left_height - right_height) > 1:
        isBalanced = False

    # return height of subtree rooted at the current node
    return max(left_height, right_height) + 1, isBalanced

'''
Construct the following tree
          1
        /   \
       /     \
      2       3
     / \     /
    4   5   6
'''
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
if isHeightBalanced(root)[1]:
    print('Binary tree is balanced')
else:
    print('Binary tree is not balanced')
```

Diagonal Traversal of a Binary tree

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


# Recursive function to perform preorder traversal on the tree and
# fill the dictionary with diagonal elements
def printDiagonal(node, diagonal, d):

    # base case: empty tree
    if node is None:
        return

    # insert the current node into the current diagonal
```

```
    d.setdefault(diagonal, []).append(node.data)

    # recur for the left subtree by increasing diagonal by 1
    printDiagonal(node.left, diagonal + 1, d)

    # recur for the right subtree with the same diagonal
    printDiagonal(node.right, diagonal, d)


# Function to print the diagonal elements of a given binary tree
def printDiagonalElements(root):

    # create an empty dictionary to store the diagonal element in every slope
    d = {}

    # perform preorder traversal on the tree and fill the dictionary
    printDiagonal(root, 0, d)

    # traverse the dictionary and print diagonal elements
    for i in range(len(d)):
        print(d.get(i))

''' Construct the following tree
           1
         /   \
        /     \
      2         3
     /         /  \
    /         /    \
   4         5      6
            / \
          /     \
         7       8
'''
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
printDiagonalElements(root)
```

Boundary traversal of a Binary tree

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

    def isLeaf(self):
        return self.left is None and self.right is None

# Recursive function to print the left boundary of the given binary tree
# in a top-down fashion, except for the leaf nodes
def printLeftBoundary(root):
    if root is None:
```

```
        return
    node = root

    while not node.isLeaf():
        print(node.data, end=' ')

        # next process, the left child of `root` if it exists; otherwise, move to the right child
        node = node.left if node.left else node.right

# Recursive function to print the right boundary of the given binary tree
# in a bottom-up fashion, except for the leaf nodes
def printRightBoundary(root):
    if root is None or root.isLeaf():
        return

    # recur for the right child of `root` if it exists; otherwise, recur for the left child
    printRightBoundary(root.right if root.right else root.left)

    # To ensure bottom-up order, print the value of the nodes after recursion unfolds
    print(root.data, end=' ')


# Recursive function to print the leaf nodes of the given binary tree in an inorder fashion
def printLeafNodes(root):
    if root is None:
        return
    printLeafNodes(root.left)

    # print only leaf nodes
    if root.isLeaf():
        print(root.data, end=' ')

    # recur for the right subtree
    printLeafNodes(root.right)


# Function to perform the boundary traversal on a given tree
def performBoundaryTraversal(root):
    if root is None:
        return

    # print the root node
    print(root.data, end=' ')

    # print the left boundary (except leaf nodes)
    printLeftBoundary(root.left)

    # print all leaf nodes
    if not root.isLeaf():
        printLeafNodes(root)

    # print the right boundary (except leaf nodes)
    printRightBoundary(root.right)

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
```

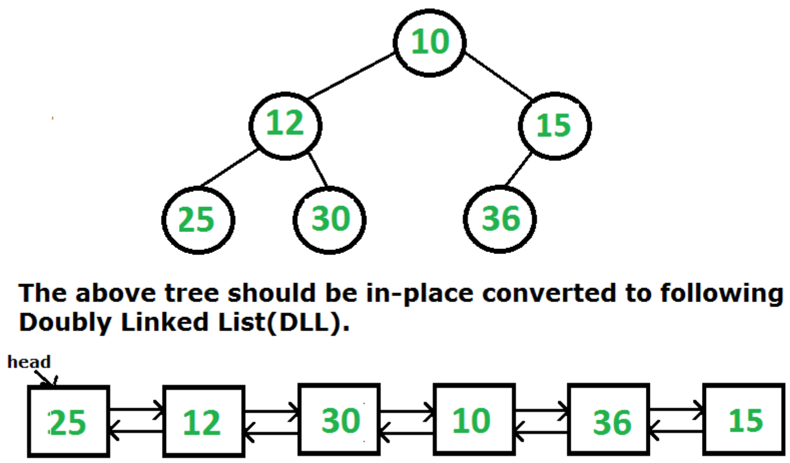
```
root.right.right = Node(7)
root.left.left.left = Node(8)
root.left.left.right = Node(9)
root.left.right.right = Node(10)
root.right.right.left = Node(11)
root.left.left.right.left = Node(12)
root.left.left.right.right = Node(13)
root.right.right.left.left = Node(14)
performBoundaryTraversal(root)
```

Construct Binary Tree from String with Bracket Representation

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

def preOrder(node):
    if node is None:
        return
    print(node.data, end=" ")
    preOrder(node.left)
    preOrder(node.right)

# function to return the index of close parenthesis
def findIndex(Str, si, ei):
    if (si > ei):
        return -1

    s = []
    for i in range(si, ei + 1):

        # if open parenthesis, push it
        if (Str[i] == '('):
            s.append(Str[i])

        elif (Str[i] == ')'):
            if (s[-1] == '('):
                s.pop(-1)

                # if stack is empty, this is
                # the required index
                if len(s) == 0:
                    return i
    # if not found return -1
    return -1

# function to construct tree from String
def treeFromString(Str, si, ei):
    # Base case
    if (si > ei):
        return None

    # new root
    root = newNode(ord(Str[si]) - ord('0'))
    index = -1
```

```
    # if next char is '(' find the
    # index of its complement ')'
    if (si + 1 <= ei and Str[si + 1] == '('):
        index = findIndex(Str, si + 1, ei)

    # if index found
    if (index != -1):

        # call for left subtree
        root.left = treeFromString(Str, si + 2, index - 1)

        # call for right subtree
        root.right = treeFromString(Str, index + 2, ei - 1)
    return root


Str = "4(2(3)(1))(6(5))"
root = treeFromString(Str, 0, len(Str) - 1)
preOrder(root)
```

Convert Binary tree into Doubly Linked List




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

def BTToDLLUtil(root):

    """This is a utility function to convert the binary tree to doubly linked list.
    Most of the core task is done by this function."""
    if root is None:
        return root

    # Convert left subtree and link to root
    if root.left:

        # Convert the left subtree
        left = BTToDLLUtil(root.left)

        # Find inorder predecessor, After this loop, left will point to the
        # inorder predecessor of root
        while left.right:
            left = left.right
```

```
        # Make root as next of predecessor
        left.right = root

        # Make predecessor as previous of root
        root.left = left

    # Convert the right subtree and link to root
    if root.right:

        # Convert the right subtree
        right = BTToDLLUtil(root.right)

        # Find inorder successor, After this loop, right will point to the inorder successor of
root
        while right.left:
            right = right.left

        # Make root as previous of successor
        right.left = root

        # Make successor as next of root
        root.right = right

    return root

def BTToDLL(root):
    if root is None:
        return root

    # Convert to doubly linked list using BLLToDLLUtil
    root = BTToDLLUtil(root)

    # We need pointer to left most node which is head of the constructed Doubly Linked list
    while root.left:
        root = root.left
    return root

def print_list(head):
    if head is None:
        return
    while head:
        print(head.data, end = " ")
        head = head.right


root = Node(10)
root.left = Node(12)
root.right = Node(15)
root.left.left = Node(25)
root.left.right = Node(30)
root.right.left = Node(36)
head = BTToDLL(root)
print_list(head)
```

Convert Binary tree into Sum tree

```
# Given a Binary Tree where each node has positive and negative values. Convert this to a tree
where each node contains the sum of the left and right sub trees in the original tree. The values
of leaf nodes are changed to 0.

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

# Convert a given tree to a tree where every node contains sum of values of
# nodes in left and right subtrees in the original tree
def toSumTree(Node) :
    if Node is None:
        return 0

    # Store the old value
    old_val = Node.data

    # Recursively call for left and right subtrees and store the sum as new value of this node
    Node.data = toSumTree(Node.left) + toSumTree(Node.right)

    # Return the sum of values of nodes in left and right subtrees and old_value of this node
    return Node.data + old_val

# A utility function to print inorder traversal of a Binary Tree
def printInorder(Node):
    if Node is None:
        return
    printInorder(Node.left)
    print(Node.data, end = " ")
    printInorder(Node.right)

# Utility function to create a new Binary Tree node
def newNode(data) :
    temp = node(0)
    temp.data = data
    temp.left = None
    temp.right = None
    return temp

root = newNode(10)
root.left = newNode(-2)
root.right = newNode(6)
root.left.left = newNode(8)
root.left.right = newNode(-4)
root.right.left = newNode(7)
root.right.right = newNode(5)
toSumTree(root)
print("Inorder Traversal of the resultant tree is: ")
printInorder(root)
```

Construct Binary tree from Inorder and preorder traversal

```
# A class to store a binary tree node
class Node:
    # Constructor
    def __init__(self, data, left=None, right=None):
        self.data = data
```

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

# Recursive function to perform inorder traversal on a given binary tree
def inorderTraversal(root):
    if root is None:
        return
    inorderTraversal(root.left)
    print(root.data, end=' ')
    inorderTraversal(root.right)

# Recursive function to perform postorder traversal on a given binary tree
def preorderTraversal(root):
    if root is None:
        return
    print(root.data, end=' ')
    preorderTraversal(root.left)
    preorderTraversal(root.right)

# Recursive function to construct a binary tree from a given inorder and preorder sequence
def construct(start, end, preorder, pIndex, d):

    # base case
    if start > end:
        return None, pIndex

    # The next element in `preorder[]` will be the root node of subtree
    # formed by sequence represented by `inorder[start, end]`
    root = Node(preorder[pIndex])
    pIndex = pIndex + 1

    # get the index of the root node in inorder to determine the
    # left and right subtree boundary
    index = d[root.data]

    # recursively construct the left subtree
    root.left, pIndex = construct(start, index - 1, preorder, pIndex, d)

    # recursively construct the right subtree
    root.right, pIndex = construct(index + 1, end, preorder, pIndex, d)

    # return current node
    return root, pIndex


# Construct a binary tree from inorder and preorder traversals.
# This function assumes that the input is valid i.e., given inorder and preorder sequence forms a
binary tree
def constructTree(inorder, preorder):

    # create a dictionary to efficiently find the index of any element in a given inorder sequence
    d = {}
    for i, e in enumerate(inorder):
        d[e] = i

    # `pIndex` stores the index of the next unprocessed node in a preorder sequence;
    # start with the root node (present at 0th index)
    pIndex = 0

    return construct(0, len(inorder) - 1, preorder, pIndex, d)[0]
```

```
''' Construct the following tree
           1
         /   \
        /     \
       2       3
      /       / \
     /       /   \
    4       5     6
           / \
          /   \
         7     8
'''
inorder = [4, 2, 1, 7, 5, 8, 3, 6]
preorder = [1, 2, 4, 3, 5, 7, 8, 6]
root = constructTree(inorder, preorder)
print('The inorder traversal is ', end='')
inorderTraversal(root)
print('\nThe preorder traversal is ', end='')
preorderTraversal(root)
```

Find minimum swaps required to convert a Binary tree into BST

```
"""
The idea is to use the fact that inorder traversal of Binary Search Tree is in increasing order of
their value. So, find the inorder traversal of the Binary Tree and store it in the array and try
to sort the array. The minimum number of swap required to get the array sorted will be the answer.
"""
def inorder(a, n, index):
    global v

    # If index is greater or equal to vector size
    if (index >= n):
        return

    inorder(a, n, 2 * index + 1)

    # Push elements in vector
    v.append(a[index])
    inorder(a, n, 2 * index + 2)

def minSwaps():
    global v
    t = [[0, 0] for _ in range(len(v))]
    ans = -2

    for i in range(len(v)):
        t[i][0], t[i][1] = v[i], i

    t, i = sorted(t), 0

    while i < len(t):
        if (i == t[i][1]):
            i += 1
            continue
        else:
```

```
            # Swapping of elements
            t[i][0], t[t[i][1]][0] = t[t[i][1]][0], t[i][0]
            t[i][1], t[t[i][1]][1] = t[t[i][1]][1], t[i][1]

        # Second is not equal to i
        if (i == t[i][1]):
            i -= 1

        i += 1
        ans += 1
    return ans

v = []
a = [5, 6, 7, 8, 9, 10, 11]
n = len(a)
inorder(a, n, 0)
print(minSwaps())
```

Check if Binary tree is Sum tree or not

```
class node:
  def __init__(self, x):
    self.data = x
    self.left = None
    self.right = None

def isLeaf(node):
  if node is None:
    return 0
  if node.left is None and node.right is None:
    return 1
  return 0

# returns data if SumTree property holds for the given tree else return -1
def isSumTree(node):
  if node is None:
    return 0
  ls = isSumTree(node.left)

  #To stop for further traversal of tree if found not sumTree
  if(ls == -1):
    return -1

  rs = isSumTree(node.right)
  #To stop for further traversal of tree if found not sumTree
  if(rs == -1):
    return -1

  return ls + rs + node.data if (isLeaf(node) or ls + rs == node.data) else -1

root = node(26)
root.left = node(10)
root.right = node(3)
root.left.left = node(4)
root.left.right = node(6)
root.right.right = node(3)
if(isSumTree(root)):
  print("The given tree is a SumTree ")
else:
```

```
    print("The given tree is not a SumTree ")
```

Check if all leaf nodes are at same level or not

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

# Recursive function which check whether all leaves are at same level
def checkUtil(root, level):
  if root is None:
    return True

  # If a tree node is encountered
  if root.left is None and root.right is None:

    # When a leaf node is found first time
    if check.leafLevel == 0 :
      check.leafLevel = level # Set first leaf found
      return True

    # If this is not first leaf node, compare its level  with first leaf's level
    return level == check.leafLevel

  # If this is not first leaf node, compare its level with first leaf's level
  return (checkUtil(root.left, level+1)and
      checkUtil(root.right, level+1))

def check(root):
  level = 0
  check.leafLevel = 0
  return (checkUtil(root, level))

root = Node(12)
root.left = Node(5)
root.left.left = Node(3)
root.left.right = Node(9)
root.left.left.left = Node(1)
root.left.right.left = Node(2)
if(check(root)):
  print("Leaves are at same level")
else:
  print("Leaves are not at same level")
```

Check if a Binary Tree contains duplicate subtrees of size 2 or more [IMP]

```
# Helper function that allocates a new node with the given data and None left and right pointers.
class newNode:
  def __init__(self, data):
    self.data = data
    self.left = self.right = None

def inorder(node, m):
  if (not node):
```

```
    return ""

  Str = "("
  Str += inorder(node.left, m)
  Str += str(node.data)
  Str += inorder(node.right, m)
  Str += ")"

  # Subtree already present (Note that we use unordered_map instead of unordered_set because we want to print multiple duplicates only once, consider example of 4 in above subtree, it should be printed only once.
  if (Str in m and m[Str] == 1):
    print(node.data, end = " ")
  if Str in m:
    m[Str] += 1
  else:
    m[Str] = 1

  return Str

# Wrapper over inorder()
def printAllDups(root):
  m = {}
  inorder(root, m)


root = None
root = newNode(1)
root.left = newNode(2)
root.right = newNode(3)
root.left.left = newNode(4)
root.right.left = newNode(2)
root.right.left.left = newNode(4)
root.right.right = newNode(4)
printAllDups(root)
```

Check if 2 trees are mirror or not

```
def checkMirrorTree(M, N, u1, v1, u2, v2):
  mp = {}

  # Traverse first tree nodes
  for i in range(N):
    if u1[i] in mp:
      mp[u1[i]].append(v1[i])
    else:
      mp[u1[i]] = []

  # Traverse second tree nodes
  for i in range(N):
    if u2[i] in mp and len(mp[u2[i]]) > 0:
      if(mp[u2[i]][-1] != v2[i]):
        return 0
      mp[u2[i]].pop()
  return 1

M, N = 7, 6

#Tree 1
```

```
u1 = [ 1, 1, 1, 10, 10, 10 ]
v1 = [ 10, 7, 3, 4, 5, 6 ]

#Tree 2
u2 = [ 1, 1, 1, 10, 10, 10 ]
v2 = [ 3, 7, 10, 6, 5, 4 ]

if(checkMirrorTree(M, N, u1, v1, u2, v2)):
    print("Yes")
else:
    print("No")
```

Sum of Nodes on the Longest path from root to leaf node

```
"""
Input : Binary tree:
        4
       / \
      2   5
     / \ / \
    7  1 2  3
      /
     6
Output : 13

        4
       / \
      2   5
     / \ / \
    7  1 2  3
      /
     6

The highlighted nodes (4, 2, 1, 6) above are
part of the longest root to leaf path having
sum = (4 + 2 + 1 + 6) = 13
"""

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

# function to find the Sum of nodes on the longest path from root to leaf node
def SumOfLongRootToLeafPath(root, Sum, Len, maxLen, maxSum):

    # if true, then we have traversed a root to leaf path
    if (not root):

        # update maximum Length and maximum Sum according to the given conditions
        if (maxLen[0] < Len):
            maxLen[0] = Len
            maxSum[0] = Sum
        elif (maxLen[0]== Len and
            maxSum[0] < Sum):
            maxSum[0] = Sum
        return
```

```
    # recur for left subtree
    SumOfLongRootToLeafPath(root.left, Sum + root.data, Len + 1, maxLen, maxSum)

    # recur for right subtree
    SumOfLongRootToLeafPath(root.right, Sum + root.data, Len + 1, maxLen, maxSum)

# utility function to find the Sum of nodes on the longest path from root to leaf node
def SumOfLongRootToLeafPathUtil(root):
    # if tree is NULL, then Sum is 0
    if (not root):
        return 0

    maxSum = [-999999999999]
    maxLen = [0]

    # finding the maximum Sum 'maxSum' for the maximum Length root to leaf path
    SumOfLongRootToLeafPath(root, 0, 0, maxLen, maxSum)
    return maxSum[0]

root = getNode(4)
root.left = getNode(2)
root.right = getNode(5)
root.left.left = getNode(7)
root.left.right = getNode(1)
root.right.left = getNode(2)
root.right.right = getNode(3)
root.left.right.left = getNode(6)
print("Sum = ", SumOfLongRootToLeafPathUtil(root))
```

Check if given graph is tree or not. [IMP]

```
from collections import defaultdict

class Graph():
  def __init__(self, V):
    self.V = V
    self.graph = defaultdict(list)

  def addEdge(self, v, w):
    self.graph[v].append(w)
    self.graph[w].append(v)

  # A recursive function that uses visited[] and parent to detect cycle in subgraph reachable from
vertex v.
  def isCyclicUtil(self, v, visited, parent):

    # Mark current node as visited
    visited[v] = True

    # Recur for all the vertices adjacent for this vertex
    for i in self.graph[v]:
      # If an adjacent is not visited, then recur for that adjacent
      if visited[i] == False:
        if self.isCyclicUtil(i, visited, v) == True:
          return True

      # If an adjacent is visited and not parent of current vertex, then there is a cycle.
      elif i != parent:
        return True
```

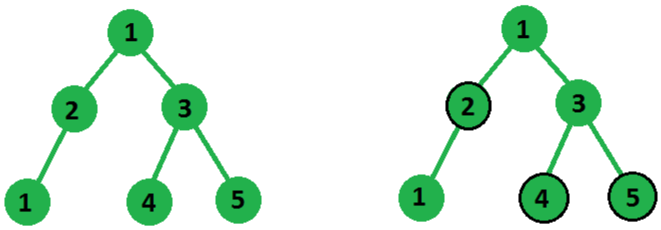
```
    return False

  # Returns true if the graph is a tree, else false.
  def isTree(self):

    # Mark all the vertices as not visited and not part of recursion stack
    visited = [False] * self.V

    # The call to isCyclicUtil serves multiple purposes. It returns true if graph reachable from
vertex 0 is cyclic.
    # It also marks all vertices reachable from 0.
    if self.isCyclicUtil(0, visited, -1) == True:
      return False

    return all(visited[i] != False for i in range(self.V))

# Driver program to test above functions
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
print ("Graph is a Tree" if g1.isTree() == True else "Graph is a not a Tree")

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
print ("Graph is a Tree" if g2.isTree() == True else "Graph is a not a Tree")
```

Find Largest subtree sum in a tree

```
# Function to create new tree node.
class newNode:
 def __init__(self, key):
  self.key = key
  self.left = self.right = None

# Helper function to find largest subtree sum recursively.
def findLargestSubtreeSumUtil(root, ans):
 if (root == None):
  return 0

 # Subtree sum rooted at current node.
 currSum = (root.key + findLargestSubtreeSumUtil(root.left, ans) +
findLargestSubtreeSumUtil(root.right, ans))

 # Update answer if current subtree sum is greater than answer so far.
 ans[0] = max(ans[0], currSum)

 # Return current subtree sum to its parent node.
 return currSum

# Function to find largest subtree sum.
def findLargestSubtreeSum(root):
 if (root == None):
```

```
  return 0
 ans = [float('-inf')]
 findLargestSubtreeSumUtil(root, ans)
 return ans[0]

# Constructed Tree
#     1
#    / \
#      /  \
#     -2   3
#     /\   /\
#    / \    / \
#      4  5  -6  2
root = newNode(1)
root.left = newNode(-2)
root.right = newNode(3)
root.left.left = newNode(4)
root.left.right = newNode(5)
root.right.left = newNode(-6)
root.right.right = newNode(2)
print(findLargestSubtreeSum(root))
```

Maximum Sum of nodes in Binary tree such that no two are adjacent



Input Binary tree   Chosen nodes with maximum sum

```
"""
Given a binary tree with a value associated with each node, we need to choose a subset of these
nodes such that the sum of
selected nodes is maximum under a constraint that no two chosen nodes in the subset should be
directly connected, that is,
if we have taken a node in our sum then we can't take any of its children in consideration and
vice versa.
"""

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

def maxSumHelper(root) :
 if (root == None):
  sum = [0, 0]
  return sum

 sum1 = maxSumHelper(root.left)
 sum2 = maxSumHelper(root.right)
 sum = [0, 0]
```

```
 # This node is included (Left and right children are not included)
 sum[0] = sum1[1] + sum2[1] + root.data

 # This node is excluded (Either left or right child is included)
 sum[1] = (max(sum1[0], sum1[1]) + max(sum2[0], sum2[1]))
 return sum

def maxSum(root) :
 res = maxSumHelper(root)
 return max(res[0], res[1])

root = newNode(10)
root.left = newNode(1)
root.left.left = newNode(2)
root.left.left.left = newNode(1)
root.left.right = newNode(3)
root.left.right.left = newNode(4)
root.left.right.right = newNode(5)
print(maxSum(root))
```

Print all "K" Sum paths in a Binary tree

```
"""
A binary tree and a number k are given. Print every path in the tree with sum of the nodes in the path as k.
A path can start from any node and end at any node and must be downward only,
i.e. they need not be root node and leaf node; and negative numbers can also be there in the tree.
"""
def printVector(v, i):
    for j in range(i, len(v)):
        print(v[j], end = " ")
    print()

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

# This function prints all paths   that have sum k
def printKPathUtil(root, path, k):
    if (not root) :
        return

    # add current node to the path
    path.append(root.data)

    # check if there's any k sum path in the left sub-tree.
    printKPathUtil(root.left, path, k)

    # check if there's any k sum path in the right sub-tree.
    printKPathUtil(root.right, path, k)

    # check if there's any k sum path that terminates at this node
    # Traverse the entire path as there can be negative elements too
    f = 0
    for j in range(len(path) - 1, -1, -1):
        f += path[j]
```

```
        # If path sum is k, prthe path
        if (f == k) :
            printVector(path, j)

    # Remove the current element from the path
    path.pop(-1)

# A wrapper over printKPathUtil()
def printKPath(root, k):
    path =[]
    printKPathUtil(root, path, k)

root = newNode(1)
root.left = newNode(3)
root.left.left = newNode(2)
root.left.right = newNode(1)
root.left.right.left = newNode(1)
root.right = newNode(-1)
root.right.left = newNode(4)
root.right.left.left = newNode(1)
root.right.left.right = newNode(2)
root.right.right = newNode(5)
root.right.right.right = newNode(2)
k = 5
printKPath(root, k)
```

Find Least Common Ancestor in a Binary tree

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

# This function returns pointer to LCA of two given values n1 and n2
# This function assumes that n1 and n2 are present in Binary Tree
def findLCA(root, n1, n2):
 if root is None:
  return None

 # If either n1 or n2 matches with root's key, report the presence by returning root (Note that if a key is
 # ancestor of other, then the ancestor key becomes LCA
 if root.key == n1 or root.key == n2:
  return root

 # Look for keys in left and right subtrees
 left_lca = findLCA(root.left, n1, n2)
 right_lca = findLCA(root.right, n1, n2)

 # If both of the above calls return Non-NULL, then one key is present in once subtree and other is present in other,
 # So this node is the LCA
 if left_lca and right_lca:
  return root

 # Otherwise check if left subtree or right subtree is LCA
 return left_lca if left_lca is not None else right_lca
```

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
print ("LCA(4,5) = ", findLCA(root, 4, 5).key)
print ("LCA(4,6) = ", findLCA(root, 4, 6).key)
print ("LCA(3,4) = ", findLCA(root, 3, 4).key)
print ("LCA(2,4) = ", findLCA(root, 2, 4).key)
```

Find distance between 2 nodes in a Binary tree

```
"""
A python program to find distance between n1 and n2 in binary tree
"""

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


# This function returns pointer to LCA of two given values n1 and n2.
def find_least_common_ancestor(root, n1, n2):
 if root is None:
  return root

 # If either n1 or n2 matches with root's key, report the presence by returning root
 if root.data == n1 or root.data == n2:
  return root

 # Look for keys in left and right subtrees
 left = find_least_common_ancestor(root.left, n1, n2)
 right = find_least_common_ancestor(root.right, n1, n2)

 if left and right:
  return root

 # Otherwise check if left subtree or right subtree is Least Common Ancestor
 if left:
  return left
 else:
  return right

# function to find distance of any node from root
def find_distance_from_ancestor_node(root, data):
 # case when we reach a beyond leaf node or when tree is empty
 if root is None:
  return -1

 # Node is found then return 0
 if root.data == data:
  return 0

 left = find_distance_from_ancestor_node(root.left, data)
```

```
 right = find_distance_from_ancestor_node(root.right, data)
 distance = max(left, right)
 return distance+1 if distance >= 0 else -1

# function to find distance between two nodes in a binary tree
def find_distance_between_two_nodes(root: Node, n1: int, n2: int):
 lca = find_least_common_ancestor(root, n1, n2)
 return find_distance_from_ancestor_node(lca, n1) + find_distance_from_ancestor_node(lca, n2) if
lca else -1

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
print("Dist(4,5) = ", find_distance_between_two_nodes(root, 4, 5))
print("Dist(4,6) = ", find_distance_between_two_nodes(root, 4, 6))
print("Dist(3,4) = ", find_distance_between_two_nodes(root, 3, 4))
print("Dist(2,4) = ", find_distance_between_two_nodes(root, 2, 4))
print("Dist(8,5) = ", find_distance_between_two_nodes(root, 8, 5))
```

Kth Ancestor of node in a Binary tree

```
"""
Given a binary tree in which nodes are numbered from 1 to n. Given a node and a positive integer
K.
We have to print the Kth ancestor of the given node in the binary tree.
If there does not exist any such ancestor then print -1.
"""

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

# recursive function to calculate Kth ancestor
def kthAncestorDFS(root, node, k):
    if (not root):
        return None

    if (root.data == node or
       (kthAncestorDFS(root.left, node, k)) or
       (kthAncestorDFS(root.right, node, k))):

        if (k[0] > 0):
            k[0] -= 1

        elif (k[0] == 0):

            # print the kth ancestor
            print("Kth ancestor is:", root.data)

            # return None to stop further backtracking
            return None
```

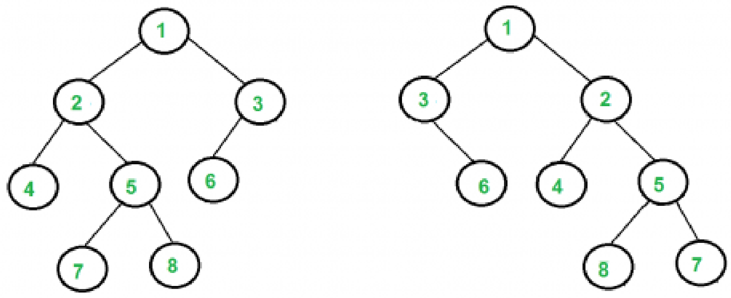
```
        # return current node to previous call
        return root


root = newNode(1)
root.left = newNode(2)
root.right = newNode(3)
root.left.left = newNode(4)
root.left.right = newNode(5)

k = [2]
node = 5

# print kth ancestor of given node
parent = kthAncestorDFS(root,node,k)

# check if parent is not None, it means there is no Kth ancestor of the node
if (parent):
    print("-1")
```

Find all Duplicate subtrees in a Binary tree [IMP]

```
# Helper function that allocates a new node with the given data and None left and right pointers.
class newNode:
 def __init__(self, data):
  self.data = data
  self.left = self.right = None

def inorder(node, m):
 if (not node):
  return ""

 Str = "("
 Str += inorder(node.left, m)
 Str += str(node.data)
 Str += inorder(node.right, m)
 Str += ")"

 # Subtree already present (Note that we use unordered_map instead of unordered_set because we
want to print
 # multiple duplicates only once, consider example of 4 in above subtree, it should be printed
only once.
 if (Str in m and m[Str] == 1):
  print(node.data, end = " ")
 if Str in m:
  m[Str] += 1
 else:
  m[Str] = 1

 return Str

# Wrapper over inorder()
def printAllDups(root):
 m = {}
 inorder(root, m)


root = None
```

```
root = newNode(1)
root.left = newNode(2)
root.right = newNode(3)
root.left.left = newNode(4)
root.right.left = newNode(2)
root.right.left.left = newNode(4)
root.right.right = newNode(4)
printAllDups(root)
```

Tree Isomorphism Problem




```
"""
Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of
them can be obtained from
other by a series of flips, i.e. by swapping left and right children of a number of nodes.
Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.
"""

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

# Check if the binary tree is isomorphic or not
def isIsomorphic(n1, n2):

 # Both roots are None, trees isomorphic by definition
 if n1 is None and n2 is None:
  return True

 # Exactly one of the n1 and n2 is None, trees are not isomorphic
 if n1 is None or n2 is None:
  return False

 if n1.data != n2.data :
  return False

 # There are two possible cases for n1 and n2 to be isomorphic

 # Case 1: The subtrees rooted at these nodes have NOT been "Flipped".
     # Both of these subtrees have to be isomorphic, hence the &&
 # Case 2: The subtrees rooted at these nodes have been "Flipped"

 return ((isIsomorphic(n1.left, n2.left)and
   isIsomorphic(n1.right, n2.right)) or
   (isIsomorphic(n1.left, n2.right) and
   isIsomorphic(n1.right, n2.left))
   )
```

```
n1 = Node(1)
n1.left = Node(2)
n1.right = Node(3)
n1.left.left = Node(4)
n1.left.right = Node(5)
n1.right.left = Node(6)
n1.left.right.left = Node(7)
n1.left.right.right = Node(8)

n2 = Node(1)
n2.left = Node(3)
n2.right = Node(2)
n2.right.left = Node(4)
n2.right.right = Node(5)
n2.left.right = Node(6)
n2.right.right.left = Node(8)
n2.right.right.right = Node(7)

print ("Yes" if (isIsomorphic(n1, n2) == True) else "No")
```

Bit Manipulation

Count set bits in an integer

```
def countSetBits(n):
    if (n == 0):
        return 0
    else:
        return 1 + countSetBits(n & (n - 1))

n = 9
print(countSetBits(n))
```

Find the two non-repeating elements in an array of repeating elements

```
def get2NonRepeatingNos(arr, n):
    s = set()
    for i in range(n):

        # Iterate through the array and check if each element is present or not in the set. If the
        # element is present, remove it from the array otherwise add it to the set

        if (arr[i] in s):
            s.remove(arr[i])
        else:
            s.add(arr[i])
    print("The 2 non repeating numbers are :",end=" ")
    for it in s:
        print(it,end=" ")
    print()

arr = [2, 3, 7, 9, 11, 2, 3, 11]
n = len(arr)
```

```
get2NonRepeatingNos(arr, n)
```

Count number of bits to be flipped to convert A to B

```
def countSetBits( n ):
    count = 0
    while n:
        count += 1
        n &= (n-1)
    return count

def FlippedCount(a , b):
    return countSetBits(a^b)

a = 10
b = 20
print(FlippedCount(a, b))
```

Count total set bits in all numbers from 1 to n

```
# Function to return the sum of the count of set bits in the integers from 1 to n
def countSetBits(n) :
  # Ignore 0 as all the bits are unset
  n += 1;
  # To store the powers of 2
  powerOf2 = 2;

  # To store the result, it is initialized with n/2 because the count of set
  # least significant bits in the integers from 1 to n is n/2
  cnt = n // 2;

  # Loop for every bit required to represent n
  while (powerOf2 <= n) :

    # Total count of pairs of 0s and 1s
    totalPairs = n // powerOf2;

    # totalPairs/2 gives the complete count of the pairs of 1s Multiplying it with the current
power of 2 will
    # give the count of 1s in the current bit
    cnt += (totalPairs // 2) * powerOf2;

    # If the count of pairs was odd then add the remaining 1s which could not be groupped together
    if (totalPairs & 1) :
      cnt += (n % powerOf2)
    else :
      cnt += 0

    # Next power of 2
    powerOf2 <<= 1;

  return cnt;

n = 14;
print(countSetBits(n));
```

Program to find whether a no is power of two

```
def isPowerOfTwo (x):
    # First x in the below expression is for the case when x is 0
    return (x and (not(x & (x - 1))) )

print('Yes') if(isPowerOfTwo(31)) else print('No')
print('Yes') if(isPowerOfTwo(64)) else print('No')
```

Find position of the only set bit

```
def isPowerOfTwo(n) :
    return (n and ( not (n & (n-1))))

# Returns position of the only set bit in 'n'
def findPosition(n) :
    if not isPowerOfTwo(n) :
        return "Invalid Number (has more than one set digits)"

    count = 0

    # One by one move the only set bit to right till it reaches end
    while (n) :
        n = n >> 1
        # increment count of shifts
        count += 1

    return count

n = 0
print(findPosition(n))
n = 12
print(findPosition(n))
n = 128
print(findPosition(n))
```

Copy set bits in a range

```
"""
Given two numbers x and y, and a range [l, r] where 1 <= l, r <= 32. The task is consider set bits
of y in range [l, r] and set these bits in x also.
Examples :

Input  : x = 10, y = 13, l = 2, r = 3
Output : x = 14
Binary representation of 10 is 1010 and
that of y is 1101. There is one set bit
in y at 3'rd position (in given range).
After we copy this bit to x, x becomes 1110
which is binary representation of 14.

Input  : x = 8, y = 7, l = 1, r = 2
Output : x = 11
"""
def copySetBits(x, y, l, r):
```

```
    # l and r must be between 1 to 3 (assuming ints are stored using 32 bits)
    if (l < 1 or r > 32):
        return x;

    # Traverse in given range
    for i in range(l, r + 1):

        # Find a mask (A number whose only set bit is at i'th position)
        mask = 1 << (i - 1);

        # If i'th bit is set in y, set i'th bit in x also.
        if ((y & mask) != 0):
            x = x | mask;
    return x;


x = 10;
y = 13;
l = 1;
r = 32;
x = copySetBits(x, y, l, r);
print("Modified x is ", x);
```

Divide two integers without using multiplication, division and mod operator

```
def divide(dividend, divisor):

    # Calculate sign of divisor i.e., sign will be negative only if
    # either one of them is negative otherwise it will be positive
    sign = -1 if ((dividend < 0) ^ (divisor < 0)) else 1

    # Update both divisor and dividend positive
    dividend = abs(dividend)
    divisor = abs(divisor)

    # Initialize the quotient
    quotient = 0
    while (dividend >= divisor):
        dividend -= divisor
        quotient += 1

    # if the sign value computed earlier is -1 then negate the value of quotient
    if sign == -1:
        quotient = -quotient
    return quotient

a = 10
b = 3
print(divide(a, b))
a = 43
b = -8
print(divide(a, b))
```

Calculate square of a number without using *, / and pow()

```
def square(n):
```

```
    # handle negative input
    if (n < 0):
        n = -n

    # Initialize result
    res = n

    # Add n to res n-1 times
    for i in range(1, n):
        res += n
    return res

for n in range(1, 6):
    print("n =", n, end=", ")
    print("n^2 =", square(n))
```

Power Set

```
#Python program to find powerset
from itertools import combinations
def print_powerset(string):
 for i in range(0,len(string)+1):
  for element in combinations(string,i):
   print(''.join(element))
string=['a','b','c']
print_powerset(string)
```

Binary Search Trees

Find a value in a BST

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


# Recursive function to insert a key into a BST
def insert(root, key):

    # if the root is None, create a new node and return it
    if root is None:
        return Node(key)

    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = insert(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    else:
        root.right = insert(root.right, key)

    return root


# Recursive function to search in a given BST
```

```
def search(root, key, parent):

    # if the key is not present in the key
    if root is None:
        print('Key not found')
        return

    # if the key is found
    if root.data == key:

        if parent is None:
            print(f'The node with key {key} is root node')
        elif key < parent.data:
            print('The given key is the left node of the node with key', parent.data)
        else:
            print('The given key is the right node of the node with key', parent.data)

        return

    # if the given key is less than the root node, recur for the left subtree;
    # otherwise, recur for the right subtree
    if key < root.data:
        search(root.left, key, root)
    else:
        search(root.right, key, root)


keys = [15, 10, 20, 8, 12, 16, 25]
root = None
for key in keys:
    root = insert(root, key)
search(root, 25, None)
```

Deletion of a node in a BST

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


# Function to perform inorder traversal on the BST
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.data, end=' ')
    inorder(root.right)


# Function to find the maximum value node in the subtree rooted at `ptr`
def findMaximumKey(ptr):
    while ptr.right:
        ptr = ptr.right
    return ptr


# Recursive function to insert a key into a BST
```

```python
def insert(root, key):
    if root is None:
        return Node(key)

    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = insert(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    else:
        root.right = insert(root.right, key)

    return root


# Function to delete a node from a BST
def deleteNode(root, key):
    if root is None:
        return root

    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = deleteNode(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    elif key > root.data:
        root.right = deleteNode(root.right, key)

    # key found
    else:
        # Case 1: node to be deleted has no children (it is a leaf node)
        if root.left is None and root.right is None:
            # update root to None
            return None

        # Case 2: node to be deleted has two children
        elif root.left and root.right:
            # find its inorder predecessor node
            predecessor = findMaximumKey(root.left)

            # copy value of the predecessor to the current node
            root.data = predecessor.data

            # recursively delete the predecessor. Note that the
            # predecessor will have at most one child (left child)
            root.left = deleteNode(root.left, predecessor.data)

        # Case 3: node to be deleted has only one child
        else:
            # choose a child node
            child = root.left if root.left else root.right
            root = child

    return root


keys = [15, 10, 20, 8, 12, 25]
root = None
for key in keys:
    root = insert(root, key)
```

```
root = deleteNode(root, 12)
inorder(root)
```

Find min and max value in a BST

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

# Function to perform inorder traversal on the BST
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.data, end=' ')
    inorder(root.right)

# Function to find the maximum value node in the subtree rooted at `ptr`
def findMaximumKey(ptr):
    while ptr.right:
        ptr = ptr.right
    return ptr.data

# Function to find the maximum value node in the subtree rooted at `ptr`
def findMinimumKey(ptr):
    while ptr.left:
        ptr = ptr.left
    return ptr.data


# Recursive function to insert a key into a BST
def insert(root, key):
    if root is None:
        return Node(key)

    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = insert(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    else:
        root.right = insert(root.right, key)
    return root


keys = [15, 10, 20, 8, 12, 25]
root = None
for key in keys:
    root = insert(root, key)
inorder(root)
print()
print("Minimum: ",findMinimumKey(root))
print("Maximum: ",findMaximumKey(root))
```

Find inorder successor and inorder predecessor in a BST

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

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

def findMinimum(root):
    while root.left:
        root = root.left
    return root

def findMaximum(root):
    while root.right:
        root = root.right
    return root

def findSuccessor(root, succ, key):
    if root is None:
        return succ

    # if a node with the desired value is found, the successor is the minimum value
    # node in its right subtree (if any)
    if root.data == key:
        if root.right:
            return findMinimum(root.right)

    # if the given key is less than the root node, recur for the left subtree
    elif key < root.data:

        # update successor to the current node before recursing in the left subtree
        succ = root
        return findSuccessor(root.left, succ, key)

    # if the given key is more than the root node, recur for the right subtree
    else:
        return findSuccessor(root.right, succ, key)
    return succ

def findPredecessor(root, prec, key):
    if root is None:
        return prec

    # if a node with the desired value is found, the predecessor is the maximum value
    # node in its left subtree (if any)
    if root.data == key:
        if root.left:
            return findMaximum(root.left)

    # if the given key is less than the root node, recur for the left subtree
    elif key < root.data:
        return findPredecessor(root.left, prec, key)
```

```
    # if the given key is more than the root node, recur for the right subtree
    else:
        # update predecessor to the current node before recursing in the right subtree
        prec = root
        return findPredecessor(root.right, prec, key)

    return prec

keys = [15, 10, 20, 8, 12, 16, 25]
''' Construct the following BST
          15
        /    \
       /      \
      10       20
     / \      /  \
    /   \    /    \
   8    12  16    25
'''
root = None
for key in keys:
    root = insert(root, key)

print("SUCCESSOR")
# find inorder successor for each key
for key in keys:
    succ = findSuccessor(root, None, key)
    if succ:
        print(f'The successor of node {key} is {succ.data}')
    else:
        print(f'No Successor exists for node {key}')


print("PREDECESSOR")
# find inorder predecessor for each key
for key in keys:
    prec = findPredecessor(root, None, key)
    if prec:
        print(f'Predecessor of node {key} is {prec.data}')
    else:
        print('The predecessor doesn\'t exist for node', key)
```

Check if a tree is a BST or not

```
import sys
class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right


def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
```

```
    return root


# Function to perform inorder traversal on the given binary tree and
# check if it is a BST or not. Here, `prev` is the previously processed node
def isBST(root, prev):

    # base case: empty tree is a BST
    if root is None:
        return True

    # check if the left subtree is BST or not
    left = isBST(root.left, prev)

    # value of the current node should be more than that of the previous node
    if root.data <= prev.data:
        return False

    # update previous node data and check if the right subtree is BST or not
    prev.data = root.data
    return left and isBST(root.right, prev)


# Function to determine whether a given binary tree is a BST
def checkForBST(node):

    # pointer to store previously processed node in the inorder traversal
    prev = Node(-sys.maxsize)

    # check if nodes are processed in sorted order
    if isBST(node, prev):
        print('The tree is a BST!')
    else:
        print('The tree is not a BST!')

def swap(root):
    left = root.left
    root.left = root.right
    root.right = left


# keys = [15, 10, 20, 8, 12, 16, 25]
keys=[8,3,1,6,7,10,14,4]
root = None
for key in keys:
    root = insert(root, key)
# swap nodes
swap(root)
checkForBST(root)
```

Populate Inorder successor of all nodes

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

```python
# Function to set the next pointer of all nodes in a binary tree.
# curr -> current node
# prev -> previously processed node
def setNextNode(curr, prev=None):
    if curr is None:
        return prev

    # recur for the left subtree
    prev = setNextNode(curr.left, prev)

    # set the previous node's next pointer to the current node
    if prev:
        prev.next = curr

    # update the previous node to the current node
    prev = curr

    # recur for the right subtree
    return setNextNode(curr.right, prev)


# Function to print inorder successor of all nodes of binary tree using the next pointer
def printInorderSuccessors(root):

    # go to the leftmost node
    curr = root
    while curr.left:
        curr = curr.left

    # print inorder successor of all nodes
    while curr.next:
        print(f'The inorder successor of {curr.data} is {curr.next.data}')
        curr = curr.next


''' Construct the following tree
          1
        /   \
       /     \
      2       3
     /       / \
    /       /   \
   4       5     6
          / \
         /   \
        7     8
'''
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.right.left = Node(5)
root.right.right = Node(6)
root.right.left.left = Node(7)
root.right.left.right = Node(8)
setNextNode(root)
printInorderSuccessors(root)
```

Find LCA of 2 nodes in a BST

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


def insert(root, key):
    if root is None:
        return Node(key)

    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = insert(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    else:
        root.right = insert(root.right, key)

    return root


# Iterative function to search a given node in a BST
def search(root, key):

    # traverse the tree and search for the key
    while root:

        # if the given key is less than the current node, go to the left
        # subtree; otherwise, go to the right subtree

        if key.data < root.data:
            root = root.left
        elif key.data > root.data:
            root = root.right
        # if the key is found, return true
        elif key == root:
            return True
        else:
            return False

    # we reach here if the key is not present in the BST
    return False


# Recursive function to find the lowest common ancestor of given nodes
# `x` and `y`, where both `x` and `y` are present in a BST
def LCARecursive(root, x, y):
    if root is None:
        return None

    # if both `x` and `y` is smaller than the root, LCA exists in the left subtree
    if root.data > max(x.data, y.data):
        return LCARecursive(root.left, x, y)

    # if both `x` and `y` are greater than the root, LCA exists in the right subtree
    elif root.data < min(x.data, y.data):
        return LCARecursive(root.right, x, y)
```

```
    # if one key is greater (or equal) than the root and one key is smaller
    # (or equal) than the root, then the current node is LCA
    return root


# Print lowest common ancestor of two nodes in a BST
def LCA(root, x, y):

    # return if the tree is empty, or `x` or `y` is not present in the tree
    if not root or not search(root, x) or not search(root, y):
        return

    # `lca` stores the lowest common ancestor of `x` and `y`
    lca = LCARecursive(root, x, y)

    # if the lowest common ancestor exists, print it
    if lca:
        print('LCA is', lca.data)
    else:
        print('LCA does not exist')

keys = [15, 10, 20, 8, 12, 16, 25]
''' Construct the following tree
         15
        /  \
       /    \
      10     20
     / \     / \
    /   \   /   \
   8    12 16   25
'''
root = None
for key in keys:
    root = insert(root, key)
LCA(root, root.left.left, root.left.right)
```

Construct BST from preorder traversal

```
import sys

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

def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.key, end=' ')
    inorder(root.right)

# Recursive function to build a BST from a preorder sequence.
# start from the root node (the first element in a preorder sequence)
# set the root node's range as [-INFINITY, INFINITY]
def buildBST(preorder, pIndex=0, min=-sys.maxsize, max=sys.maxsize):

    # Base case
```

```
    if pIndex == len(preorder):
        return None, pIndex

    # Return if the next element of preorder traversal is not in the valid range
    val = preorder[pIndex]
    if val < min or val > max:
        return None, pIndex

    # Construct the root node and increment `pIndex`
    root = Node(val)
    pIndex = pIndex + 1

    # Since all elements in the left subtree of a BST must be less
    # than the root node's value, set range as `[min, val-1]` and recur
    root.left, pIndex = buildBST(preorder, pIndex, min, val - 1)

    # Since all elements in the right subtree of a BST must be greater
    # than the root node's value, set range as `[val+1…max]` and recur
    root.right, pIndex = buildBST(preorder, pIndex, val + 1, max)

    return root, pIndex


''' Construct the following BST
          15
        /    \
       /      \
      10       20
     /  \     /  \
    /    \   /    \
   8     12 16    25
'''
preorder = [15, 10, 8, 12, 20, 16, 25]
root = buildBST(preorder)[0]
print('Inorder traversal of BST is:', end=' ')
inorder(root)
```

Convert Binary tree into BST

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

# Function to perform inorder traversal on the tree
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.data, end=' ')
    inorder(root.right)

# Function to traverse the binary tree and store its keys in a set
def extractKeys(root, keys):
    # base case
    if root is None:
        return
    extractKeys(root.left, keys)
```

```python
    keys.append(root.data)
    extractKeys(root.right, keys)


# Function to put keys back into a set in their correct order in a BST
# by doing inorder traversal
def convertToBST(root, it):
    if root is None:
        return
    convertToBST(root.left, it)
    root.data = next(it)
    convertToBST(root.right, it)


# Function to convert a binary tree to BST by maintaining its original structure
def convert(root):
    # traverse the binary tree and store its keys in a set
    keys = []
    extractKeys(root, keys)

    # put back keys present in the set to their correct order in the BST
    it = iter(sorted(keys))
    convertToBST(root, it)

'''
Construct the following tree
           8
         /   \
        /     \
       3       5
      / \     / \
     /   \   /   \
    10    2 4     6
'''
root = Node(8)
root.left = Node(3)
root.right = Node(5)
root.left.left = Node(10)
root.left.right = Node(2)
root.right.left = Node(4)
root.right.right = Node(6)
convert(root)
inorder(root)
```

Convert a normal BST into a Balanced BST

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


# Function to perform the preorder traversal on a BST
def preorder(root):
    if root is None:
        return
    print(root.data, end=' ')
    preorder(root.left)
```

```
    preorder(root.right)


# Recursive function to push nodes of a given binary search tree into a
# list in an inorder fashion
def pushTreeNodes(root, nodes):
    if root is None:
        return

    pushTreeNodes(root.left, nodes)
    nodes.append(root)
    pushTreeNodes(root.right, nodes)


# Recursive function to construct a height-balanced BST from
# given nodes in sorted order
def buildBalancedBST(nodes, start, end):
    if start > end:
        return None

    # find the middle index
    mid = (start + end) // 2

    # The root node will be a node present at the mid-index
    root = nodes[mid]

    # recursively construct left and right subtree
    root.left = buildBalancedBST(nodes, start, mid - 1)
    root.right = buildBalancedBST(nodes, mid + 1, end)

    # return root node
    return root


# Function to construct a height-balanced BST from an unbalanced BST
def constructBalancedBST(root):

    # Push nodes of a given binary search tree into a list in sorted order
    nodes = []
    pushTreeNodes(root, nodes)

    # Construct a height-balanced BST from sorted BST nodes
    return buildBalancedBST(nodes, 0, len(nodes) - 1)


root = Node(20)
root.left = Node(15)
root.left.left = Node(10)
root.left.left.left = Node(5)
root.left.left.left.left = Node(2)
root.left.left.left.right = Node(8)
root = constructBalancedBST(root)
print('Preorder traversal of the constructed BST is ', end='')
preorder(root)
```

Merge two BST

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

```
        self.data = data
        self.left = left
        self.right = right

# Function to push a BST node at the front of a doubly linked list
def push(root, head):
    root.right = head
    if head:
        head.left = root
    head = root
    return head

# Function to print and count the total number of nodes in a doubly-linked list
def size(node):
    counter = 0
    while node:
        node = node.right
        counter = counter + 1
    return counter

# Function to print preorder traversal of the BST
def preorder(root):
    if root is None:
        return
    print(root.data, end=' ')
    preorder(root.left)
    preorder(root.right)

# Recursive method to construct a balanced BST from a sorted doubly linked list
def convertSortedDLLToBalancedBST(head, n):
    if n <= 0:
        return None, head

    # recursively construct the left subtree
    leftSubTree, head = convertSortedDLLToBalancedBST(head, n // 2)

    # `head` now points to the middle node of the sorted DDL
    # make the middle node of the sorted DDL as the root node of the BST
    root = head

    # update left child of the root node
    root.left = leftSubTree

    # update the head reference of the doubly linked list
    head = head.right

    # recursively construct the right subtree with the remaining nodes
    root.right, head = convertSortedDLLToBalancedBST(head, n - (n // 2 + 1))
    # +1 for the root
    # return the root node
    return root, head


# Recursive method to convert a BST into a doubly-linked list. It takes
# the BST's root node and the head node of the doubly linked list as an argument
def convertBSTtoSortedDLL(root, head=None):
    if root is None:
        return head

    # recursively convert the right subtree
```

```
    head = convertBSTtoSortedDLL(root.right, head)

    # push the current node at the front of the doubly linked list
    head = push(root, head)

    # recursively convert the left subtree
    head = convertBSTtoSortedDLL(root.left, head)

    return head


# Recursive method to merge two doubly-linked lists into a
# single doubly linked list in sorted order
def sortedMerge(first, second):

    # if the first list is empty, return the second list
    if first is None:
        return second

    # if the second list is empty, return the first list
    if second is None:
        return first

    # if the head node of the first list is smaller
    if first.data < second.data:
        first.right = sortedMerge(first.right, second)
        first.right.left = first
        return first

    # if the head node of the second list is smaller
    else:
        second.right = sortedMerge(first, second.right)
        second.right.left = second
        return second


# Function to merge two balanced BSTs into a single balanced BST
def merge(first, second):

    # merge both BSTs into a sorted doubly linked list
    head = sortedMerge(convertBSTtoSortedDLL(first), convertBSTtoSortedDLL(second))

    # construct a balanced BST from a sorted doubly linked list
    root, head = convertSortedDLLToBalancedBST(head, size(head))
    return root


'''
Construct the first BST
      20
     /  \
    10   30
        / \
       25 100
'''
first = Node(20)
first.left = Node(10)
first.right = Node(30)
first.right.left = Node(25)
first.right.right = Node(100)
```

```
'''
Construct the second BST
      50
     /  \
    5   70
'''
second = Node(50)
second.left = Node(5)
second.right = Node(70)
# merge both BSTs
root = merge(first, second)
preorder(root)
```

Find Kth largest element and Kth smallest element in a BST

```
import sys
class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    return root

# Function to find the k'th largest node in a BST. Here, `i` denotes the total number of nodes
processed so far
def kthLargest(root, i, k):
    if root is None:
        return None, i

    # search in the right subtree
    left, i = kthLargest(root.right, i, k)

    # if k'th largest is found in the left subtree, return it
    if left:
        return left, i

    i = i + 1

    # if the current node is k'th largest, return its value
    if i == k:
        return root, i

    # otherwise, search in the left subtree
    return kthLargest(root.left, i, k)


def findKthLargest(root, k):
    i = 0
    # traverse the tree in an inorder fashion and return k'th node
    return kthLargest(root, i, k)[0]
```

```
def kthSmallest(root, counter, k):
    if root is None:
        return None, counter

    # recur for the left subtree
    left, counter = kthSmallest(root.left, counter, k)

    # if k'th smallest node is found
    if left:
        return left, counter

    # if the root is k'th smallest node
    counter = counter + 1
    if counter == k:
        return root, counter

    # recur for the right subtree only if k'th smallest node is not found
    # in the right subtree
    ret, counter = kthSmallest(root.right, counter, k)
    return ret, counter

def findKthSmallest(root, k):
    counter = 0
    # recursively find the k'th smallest node
    return kthSmallest(root, counter, k)[0]

keys = [15, 10, 20, 8, 12, 16, 25]
root = None
for key in keys:
    root = insert(root, key)


k = 2
print(f"{k}th LARGEST NODE")
result = findKthLargest(root, k)
if result != sys.maxsize:
    print(result.data)
else:
    print('Invalid Input')


k = 4
print(f"{k}th SMALLEST NODE")
result = findKthSmallest(root, k)
if result:
    print(result.data)
else:
    print(f'{k}\'th smallest node does not exist.')
```

Count pairs from 2 BST whose sum is equal to given value "X"

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

```
root1,root2 = None,None

# def to count pairs from two BSTs whose sum is equal to a given value x
pairCount = 0
def traverseTree(root1, root2, sum):
 if root1 is None or root2 is None:
  return
 traverseTree(root1.left, root2, sum)
 traverseTree(root1.right, root2, sum)
 diff = sum - root1.data
 findPairs(root2, diff)


def findPairs(root2 , diff):
 global pairCount
 if root2 is None:
  return

 if (diff > root2.data) :
  findPairs(root2.right, diff)
 else :
  findPairs(root2.left, diff)
 if (root2.data == diff):
  pairCount += 1

def countPairs(root1, root2, sum):
 global pairCount

 traverseTree(root1, root2, sum)
 return pairCount


root1 = Node(5)
root1.left = Node(3)
root1.right = Node(7)
root1.left.left = Node(2)
root1.left.right = Node(4)
root1.right.left = Node(6)
root1.right.right = Node(8)

# formation of BST 2
root2 = Node(10)
root2.left = Node(6)
root2.right = Node(15)
root2.left.left = Node(3)
root2.left.right = Node(8)
root2.right.left = Node(11)
root2.right.right = Node(18)

x = 16
print(f"Pairs = {countPairs(root1, root2, x)}")
```

Find the median of BST in O(n) time and O(1) space

```
_MIN=float('-inf')
_MAX=float('inf')

# Helper function that allocates a new node with the given data and None left and right pointers.
```

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

# A utility function to insert a new node with given key in BST
def insert(node,key):
 if node is None:
  return newNode(key)

 # Otherwise, recur down the tree
 if (key < node.data):
  node.left = insert(node.left, key)
 elif (key > node.data):
  node.right = insert(node.right, key)

 # return the (unchanged) node pointer
 return node


#Function to count nodes in a binary search tree using Morris Inorder traversal
def counNodes(root):
 count = 0
 if root is None:
  return count

 current = root
 while (current != None):
  if current.left is None:
   # Count node if its left is None
   count+=1
   # Move to its right
   current = current.right

  else:
   # Find the inorder predecessor of current
   pre = current.left

   while pre.right not in [None, current]:
    pre = pre.right

   #Make current as right child of its inorder predecessor
   if pre.right is None:
    pre.right = current
    current = current.left
   else:
    pre.right = None
    # Increment count if the current node is to be visited
    count += 1
    current = current.right

 return count


def findMedian(root):
 if root is None:
  return 0
 count = counNodes(root)
 currCount = 0
```

```
current = root

while (current != None):

 if current.left is None:

  # count current node
  currCount += 1

  # check if current node is the median
  # Odd case
  if (count % 2 != 0 and
   currCount == (count + 1)//2):
   return prev.data

  # Even case
  elif (count % 2 == 0 and
    currCount == (count//2)+1):
   return (prev.data + current.data)//2

  # Update prev for even no. of nodes
  prev = current

  #Move to the right
  current = current.right

 else:

  # Find the inorder predecessor of current
  pre = current.left
  while pre.right not in [None, current]:
   pre = pre.right

  # Make current as right child of its inorder predecessor
  if pre.right is None:

   pre.right = current
   current = current.left
  else:

   pre.right = None

   prev = pre

   # Count current node
   currCount+= 1

   # Check if the current node is the median
   if (count % 2 != 0 and
    currCount == (count + 1) // 2 ):
    return current.data

   elif (count%2 == 0 and
    currCount == (count // 2) + 1):
    return (prev.data+current.data)//2

   # update prev node for the case of even
   # no. of nodes
   prev = current
   current = current.right
```

```
"""
Constructed binary tree is
 50
 / \
    30 70
    / \ / \
    20 40 60 80
"""
root = newNode(50)
insert(root, 30)
insert(root, 20)
insert(root, 40)
insert(root, 70)
insert(root, 60)
insert(root, 80)
print("Median of BST is ",findMedian(root))
```

Count BST nodes that lie in a given range

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

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

def countNodes(root, low, high):
    if root is None:
        return 0

    # keep track of the total number of nodes in the tree rooted with `root`.
    # that lies within the current range [low, high]
    count = 0

    # increment count if the current node lies within the current range
    if low <= root.data <= high:
        count += 1

    # recur for the left subtree
    count += countNodes(root.left, low, high)

    # recur for the right subtree and return the total count
    return count + countNodes(root.right, low, high)


low, high = 12, 20
keys = [15, 25, 20, 22, 30, 18, 10, 8, 9, 12, 6]
root = None
for key in keys:
```

```
    root = insert(root, key)

print('The total number of nodes is', countNodes(root, low, high))
```

Replace every element with the least greater element on its right

```
"""
Given an array of integers, replace every element with the least greater element on its right side
in the array. If there are no greater elements on the right side, replace it with -1.

Examples:
Input: [8, 58, 71, 18, 31, 32, 63, 92,
         43, 3, 91, 93, 25, 80, 28]
Output: [18, 63, 80, 25, 32, 43, 80, 93,
         80, 25, 93, -1, 28, -1, -1]
"""

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

# A utility function to insert a new node with given data in BST and find its successor
def insert(node, data):
 global succ
 # If the tree is empty, return a new node
 root = node

 if node is None:
  return Node(data)

 # If key is smaller than root's key, go to left subtree and set successor as current node
 if (data < node.data):

  #print("1")
  succ = node
  root.left = insert(node.left, data)

 # Go to right subtree
 elif (data > node.data):
  root.right = insert(node.right, data)

 return root

# Function to replace every element with the least greater element on its right
def replace(arr, n):

 global succ
 root = None

 # Start from right to left
 for i in range(n - 1, -1, -1):
  succ = None

  # Insert current element into BST and find its inorder successor
  root = insert(root, arr[i])
```

```
  # Replace element by its inorder successor in BST
  arr[i] = succ.data if succ else -1
 return arr



arr = [ 8, 58, 71, 18, 31, 32, 63,
  92, 43, 3, 91, 93, 25, 80, 28 ]
n = len(arr)
succ = None
arr = replace(arr, n)
print(*arr)
```

Given "n" appointments, find the conflicting appointments

```
"""
Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}}
Output: Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
"""



class Interval:
 def __init__(self):
  self.low = None
  self.high = None

# Structure to represent a node in Interval Search Tree
class ITNode:
 def __init__(self):
  self.max = None
  self.i = None
  self.left = None
  self.right = None

def newNode(j):
 #print(j)
 temp = ITNode()
 temp.i = j
 temp.max = j[1]
 return temp

# A utility function to check if given two intervals overlap
def doOverlap(i1, i2):
 if (i1[0] < i2[1] and i2[0] < i1[1]):
  return True
 return False

# Function to create a new node
def insert(node, data):
 global succ

 # If the tree is empty, return a new node
 root = node
 if node is None:
```

```
  return newNode(data)

 # If key is smaller than root's key, go to left subtree and set successor as current node
print(node)
 if (data[0] < node.i[0]):
  root.left = insert(node.left, data)

 # Go to right subtree
 else:
  root.right = insert(node.right, data)
 if root.max < data[1]:
  root.max = data[1]

 return root

# The main function that searches a given interval i in a given Interval Tree.
def overlapSearch(root, i):
 if root is None:
  return None

 # If given interval overlaps with root
 if (doOverlap(root.i, i)):
  return root.i

 # If left child of root is present and max of left child is greater than or
 # equal to given interval, then i may overlap with an interval is left subtree
 if (root.left != None and root.left.max >= i[0]):
  return overlapSearch(root.left, i)

 # Else interval can only overlap with right subtree
 return overlapSearch(root.right, i)

# This function prints all conflicting appointments in a given array of appointments.
def printConflicting(appt, n):
 # Create an empty Interval Search Tree, add first appointment
 root = None
 root = insert(root, appt[0])

 # Process rest of the intervals
 for i in range(1, n):
  # If current appointment conflicts with any of the existing intervals, print it
  res = overlapSearch(root, appt[i])

  if (res != None):
   print("[", appt[i][0], ",", appt[i][1],
    "] Conflicts with [", res[0],
    ",", res[1], "]")

  # Insert this appointment
  root = insert(root, appt[i])



appt = [ [ 1, 5 ], [ 3, 7 ],
  [ 2, 6 ], [ 10, 15 ],
  [ 5, 6 ], [ 4, 100 ]
    ]
n = len(appt)
print("Following are conflicting intervals")
printConflicting(appt, n)
```

Preorder to Postorder

```
#  A class to store a binary tree node
class Node:
    def __init__(self, key):
        self.key = key

def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.key, end=' ')



# Recursive function to build a BST from a preorder sequence.
def constructBST(preorder, start, end):

    # base case
    if start > end:
        return None

    # Construct the root node of the subtree formed by keys of the
    # preorder sequence in range `[start, end]`
    node = Node(preorder[start])

    # search the index of the first element in the current range of preorder
    # sequence larger than the root node's value
    i = start
    while i <= end:
        if preorder[i] > node.key:
            break
        i = i + 1

    # recursively construct the left subtree
    node.left = constructBST(preorder, start + 1, i - 1)

    # recursively construct the right subtree
    node.right = constructBST(preorder, i, end)

    # return current node
    return node


'''
Construct the following BST
          15
        /    \
       /      \
      10       20
     /  \     /  \
    /    \   /    \
   8     12 16     25
'''
```

```
preorder = [15, 10, 8, 12, 20, 16, 25]
root = constructBST(preorder, 0, len(preorder) - 1)
print('Postorder traversal of BST is ', end='')
postorder(root)
```

Check whether BST contains Dead end

```
"""
Given a Binary search Tree that contains positive integer values greater than 0. The task is to
check whether the BST contains a dead end or not. Here Dead End means, we are not able to insert
any element after that node.

Examples:

Input :        8
             /   \
           5      9
         /   \
        2     7
       /
      1
Output : Yes
Explanation : Node "1" is the dead End because after that we cant insert any element.

Input :       8
            /   \
           7     10
         /      /   \
        2      9     13

Output : Yes
Explanation : We can't insert any element at node 9.
"""

all_nodes = set()
leaf_nodes = set()

# A BST node
class newNode:

 def __init__(self, data):

  self.data = data
  self.left = None
  self.right = None

# A utility function to insert a new Node with given key in BST
def insert(node, key):
 if node is None:
  return newNode(key)

 # Otherwise, recur down the tree
 if (key < node.data):
  node.left = insert(node.left,
      key)
 elif (key > node.data):
  node.right = insert(node.right,
       key)
```

```
 # return the (unchanged) Node pointer
 return node

# Function to store all node of given binary search tree
def storeNodes(root):

 global all_nodes
 global leaf_nodes
 if root is None:
  return

 # store all node of binary search tree
 all_nodes.add(root.data)

 # store leaf node in leaf_hash
 if root.left is None and root.right is None:
  leaf_nodes.add(root.data)
  return

 # recur call rest tree
 storeNodes(root. left)
 storeNodes(root.right)

# Returns true if there is a dead end in tree, else false.
def isDeadEnd(root):

 global all_nodes
 global leaf_nodes

 if root is None:
  return False

 # create two empty hash sets that store all BST elements and leaf nodes respectively.

 # insert 0 in 'all_nodes' for handle case if bst contain value 1
 all_nodes.add(0)

 # Call storeNodes function to store all BST Node
 storeNodes(root)

 return any(((x + 1) in all_nodes and (x - 1) in all_nodes) for x in leaf_nodes)

root = None
root = insert(root, 8)
root = insert(root, 5)
root = insert(root, 2)
root = insert(root, 3)
root = insert(root, 7)
root = insert(root, 11)
root = insert(root, 4)

if(isDeadEnd(root) == True):
 print("Yes")
else:
 print("No")
```

Largest BST in a Binary Tree [V.V.V.V.V IMP]

```
import sys
sys.setrecursionlimit(1000000)
from collections import deque

IMIN = float('-inf')
IMAX = float('inf')

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

def largestBst(root):
 if root is None:
  return IMAX,IMIN,0
 if root.left is None and root.right is None:
  return root.data,root.data,1

 left=largestBst(root.left)
 right=largestBst(root.right)

 ans=[0,0,0]

 if left[1]<root.data and right[0]>root.data:
  ans[0]=min(left[0],right[0],root.data)
  ans[1]=max(right[1],left[1],root.data)
  ans[2]=1+left[2]+right[2]
  return ans

 ans[0]=IMIN
 ans[1]=IMAX
 ans[2]=max(left[2],right[2])
 return ans[2]


"""
 50
 / \
  75   45
 /
40
"""
root = newNode(50)
root.left = newNode(75)
root.right = newNode(45)
root.left.left = newNode(40)
print("Size of the largest BST is",largestBst(root))
```

Flatten BST to sorted list

```
global prev
class node :
 def __init__(self, data):
  self.data = data
  self.left = None
  self.right = None
```

```
def printTree(parent):
 root = parent
 while root is not None:
  print(root.data,end=' ')
  root = root.right

def inorder(root):
 global prev
 if root is None:
  return
 inorder(root.left)
 print(root.data,end=' ')
 inorder(root.right)


# Function to flatten binary tree using level order traversal BFS
def flatten(parent):
 global prev
    # Dummy node
 dummy = node(-1)

 # Pointer to previous element
 prev = dummy

 # Calling in-order traversal
 inorder(parent)

 prev.left = None
 prev.right = None

 # Delete dummy node
 return dummy.right

root = node(5)
root.left = node(3)
root.right = node(7)
root.left.left = node(2)
root.left.right = node(4)
root.right.left = node(6)
root.right.right = node(8)
printTree(flatten(root))
```

Dynamic Programming

Coin ChangeProblem

```
"""
Given an unlimited supply of coins of given denominations, find the total number of distinct ways
to get the desired change.

For example,

Input: S = { 1, 3, 5, 7 }, target = 8

The total number of ways is 6

{ 1, 7 }
{ 3, 5 }
```

```
{ 1, 1, 3, 3 }
{ 1, 1, 1, 5 }
{ 1, 1, 1, 1, 1, 3 }
{ 1, 1, 1, 1, 1, 1, 1, 1 }


Input: S = { 1, 2, 3 }, target = 4

The total number of ways is 4

{ 1, 3 }
{ 2, 2 }
{ 1, 1, 2 }
{ 1, 1, 1, 1 }
"""

def count(S, n, target):
    if target == 0:
        return 1

    # return 0 (solution does not exist) if total becomes negative, no elements are left
    if target < 0 or n < 0:
        return 0

    # Case 1. Include current coin `S[n]` in solution and recur
    # with remaining change `target-S[n]` with the same number of coins
    incl = count(S, n, target - S[n])

    # Case 2. Exclude current coin `S[n]` from solution and recur for remaining coins `n-1`
    excl = count(S, n - 1, target)

    # return total ways by including or excluding current coin
    return incl + excl


# `n` coins of given denominations
S = [1, 2, 3]
# total change required
target = 4
print('The total number of ways to get the desired change is',
    count(S, len(S) - 1, target))
```

Knapsack Problem

```
"""
In the 0-1 Knapsack problem, we are given a set of items, each with a weight and a value, and we
need to determine the number of each item to include in a collection so that the total weight is
less than or equal to a given limit and the total value is as large as possible.

Please note that the items are indivisible; we can either take an item or not (0-1 property). For
example,

Input:

value = [ 20, 5, 10, 40, 15, 25 ]
weight = [ 1, 2, 3, 8, 7, 4 ]
int W = 10
```

```
Output: Knapsack value is 60

value = 20 + 40 = 60
weight = 1 + 8 = 9 < W
"""

import sys


# Values (stored in list `v`)
# Weights (stored in list `w`)
# Total number of distinct items `n`
# Knapsack capacity `W`
def knapsack(v, w, n, W):
    if W < 0:
        return -sys.maxsize

    # base case: no items left or capacity becomes 0
    if n < 0 or W == 0:
        return 0

    # Case 1. Include current item `n` in knapsack `v[n]` and recur for
    # remaining items `n-1` with decreased capacity `W-w[n]`
    include = v[n] + knapsack(v, w, n - 1, W - w[n])

    # Case 2. Exclude current item `v[n]` from the knapsack and recur for
    # remaining items `n-1`
    exclude = knapsack(v, w, n - 1, W)

    # return maximum value we get by including or excluding the current item
    return max(include, exclude)



# input: a set of items, each with a weight and a value
v = [20, 5, 10, 40, 15, 25]
w = [1, 2, 3, 8, 7, 4]
# knapsack capacity
W = 10
print('Knapsack value is', knapsack(v, w, len(v) - 1, W))
```

Binomial CoefficientProblem

```
"""
A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects
can be chosen from among n objects more formally, the number of k-element subsets (or k-
combinations) of a n-element set.
"""

def binomialCoeff(n, k):
    C = [0 for i in range(k+1)]
    C[0] = 1  # since nC0 is 1

    for i in range(1, n+1):
        # Compute next row of pascal triangle using the previous row
        j = min(i, k)
        while (j > 0):
```

```
            C[j] = C[j] + C[j-1]
            j -= 1

    return C[k]


n = 5
k = 2
print ("Value of C(%d,%d) is %d" % (n, k, binomialCoeff(n, k)))
```

Permutation CoefficientProblem

```
"""
P(10, 2) = 90
P(10, 3) = 720
P(10, 0) = 1
P(10, 1) = 10
"""

def permutationCoeff(n, k):
    # P(n,k)=n*(n-1)*(n-2)*....(n-k-1)
    f=1
    for i in range(k):
        f*=(n-i)
    return f

n = 10
k = 2
print("Value of P(", n, ",", k, ") is ", permutationCoeff(n, k))
```

Program for nth Catalan Number

```
"""
Catalan numbers are a sequence of natural numbers that occurs in many interesting counting
problems like the following.
1) Count the number of expressions containing n pairs of parentheses which are correctly matched.
For n = 3, possible expressions are ((())), ()(()), ()()(), (())(), (()()).

2) Count the number of possible Binary Search Trees with n keys (See this)

The first few Catalan numbers for n = 0, 1, 2, 3, … are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862,
…
"""

def catalan(n):
    return 1 if n <= 1 else sum(catalan(i) * catalan(n-i-1) for i in range(n))

for i in range(10):
    print(catalan(i), end=' ')
```

Matrix Chain Multiplication

```
"""
Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
```

```
multiplications are obtained by putting parenthesis in following way
(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}
Output: 6000
There are only two matrices of dimensions 10x20 and 20x30. So there
is only one way to multiply the matrices, cost of which is 10*20*30
"""

import sys

# Matrix A[i] has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, i, j):
    if i == j:
        return 0
    _min = sys.maxsize

    # place parenthesis at different places between first and last matrix,
    # recursively calculate count of multiplications for each parenthesis
    # placement and return the minimum count
    for k in range(i, j):
        count = (MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i-1] * p[k] * p[j])
        if count < _min:
            _min = count
    # Return minimum count
    return _min

arr = [1, 2, 3, 4, 3]
n = len(arr)
print("Minimum number of multiplications is ",
    MatrixChainOrder(arr, 1, n-1))
```

Edit Distance

```
"""
The Levenshtein distance (or Edit distance) is a way of quantifying how different two strings are
from one another by counting the minimum number of operations required to transform one string
into the other
('ABA', 'ABC') --> ('ABAC', 'ABC') == ('ABA', 'AB')
"""

def dist(X, Y):
    # `m` and `n` is the total number of characters in `X` and `Y`, respectively
    (m, n) = (len(X), len(Y))
    # For all pairs of `i` and `j`, `T[i, j]` will hold the Levenshtein distance
    # between the first `i` characters of `X` and the first `j` characters of `Y`.
    # Note that `T` holds `(m+1)x(n+1)` values.
    T = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    # we can transform source prefixes into an empty string by dropping all characters
    for i in range(1, m + 1):
```

```
        T[i][0] = i                     # (case 1)

    # we can reach target prefixes from empty source prefix by inserting every character
    for j in range(1, n + 1):
        T[0][j] = j                     # (case 1)

    # fill the lookup table in a bottom-up manner
    for i in range(1, m + 1):

        for j in range(1, n + 1):
            cost = 0 if X[i - 1] == Y[j - 1] else 1
            T[i][j] = min(T[i - 1][j] + 1,      # deletion
                          T[i][j - 1] + 1,      # insertion
                          T[i - 1][j - 1] + cost) # replace

    return T[m][n]


X = 'kitten'
Y = 'sitting'
print('The Levenshtein distance is', dist(X, Y))
```

Subset Sum Problem

```
"""
Given a set of positive integers and an integer k, check if there is any non-empty subset that
sums to k.
A = { 7, 3, 2, 5, 8 }
k = 14

Output: Subset with the given sum exists
Subset { 7, 2, 5 } sums to 14
"""

def subsetSum(A, k):
    n = len(A)
    # `T[i][j]` stores true if subset with sum `j` can be attained
    # using items up to first `i` items
    T = [[False for _ in range(k + 1)] for _ in range(n + 1)]

    # if the sum is zero
    for i in range(n + 1):
        T[i][0] = True

    # do for i'th item
    for i in range(1, n + 1):
        # consider all sum from 1 to sum
        for j in range(1, k + 1):
            # don't include the i'th element if `j-A[i-1]` is negative
            if A[i - 1] > j:
                T[i][j] = T[i - 1][j]
            else:
                # find the subset with sum `j` by excluding or including the i'th item
                T[i][j] = T[i - 1][j] or T[i - 1][j - A[i - 1]]

    # return maximum value
    return T[n][k]
```

```
# Input: a set of items and a sum
A = [7, 3, 2, 5, 8]
k = 18
if subsetSum(A, k):
    print('Subsequence with the given sum exists')
else:
    print('Subsequence with the given sum does not exist')
```

Friends Pairing Problem

```
"""
 Input  : n = 3
 Output : 4
 Explanation:
 {1}, {2}, {3} : all single
 {1}, {2, 3} : 2 and 3 paired but 1 is single.
 {1, 2}, {3} : 1 and 2 are paired but 3 is single.
 {1, 3}, {2} : 1 and 3 are paired but 2 is single.
 Note that {1, 2} and {2, 1} are considered same.

 Mathematical Explanation:
 The problem is simplified version of how many ways we can divide n elements into multiple groups.
 (here group size will be max of 2 elements).
 In case of n = 3, we have only 2 ways to make a group:
     1) all elements are individual(1,1,1)
     2) a pair and individual (2,1)
 In case of n = 4, we have 3 ways to form a group:
     1) all elements are individual (1,1,1,1)
     2) 2 individuals and one pair (2,1,1)
     3) 2 separate pairs (2,2)
"""


# Returns count of ways n people can remain single or paired up.
def countFriendsPairings(n):

    dp = [0 for _ in range(n + 1)]

    # Filling dp[] in bottom-up manner using recursive formula explained above.
    for i in range(n + 1):

        dp[i] = i if (i <= 2) else dp[i - 1] + (i - 1) * dp[i - 2]
    return dp[n]

n = 4
print(countFriendsPairings(n))
```

Gold Mine Problem

```
"""
Given a gold mine of n*m dimensions. Each field in this mine contains a positive integer which is
the amount of gold in tons. Initially the miner is at first column but can be at any row. He can
move only (right->,right up /,right down\) that is from a given cell, the miner can move to the
cell diagonally up towards the right or right or diagonally down towards the right. Find out
maximum amount of gold he can collect.
Examples:
```

```
Input : mat[][] = {{1, 3, 3},
                   {2, 1, 4},
                  {0, 6, 4}};
Output : 12
{(1,0)->(2,1)->(1,2)}

Input: mat[][] = { {1, 3, 1, 5},
                   {2, 2, 4, 1},
                   {5, 0, 2, 3},
                   {0, 6, 1, 2}};
Output : 16
(2,0) -> (1,1) -> (1,2) -> (0,3) OR
(2,0) -> (3,1) -> (2,2) -> (2,3)

"""



def collectGold(gold, x, y, n, m):
    if ((x < 0) or (x == n) or (y == m)):
        return 0

    # Right upper diagonal
    rightUpperDiagonal = collectGold(gold, x - 1, y + 1, n, m)

    # right
    right = collectGold(gold, x, y + 1, n, m)

    # Lower right diagonal
    rightLowerDiagonal = collectGold(gold, x + 1, y + 1, n, m)

    # Return the maximum and store the value
    return gold[x][y] + max(max(rightUpperDiagonal, rightLowerDiagonal), right)


def getMaxGold(gold,n,m):
    maxGold = 0
    for i in range(n):
        goldCollected = collectGold(gold, i, 0, n, m)
        maxGold = max(maxGold, goldCollected)
    return maxGold

gold = [[1, 3, 1, 5],
        [2, 2, 4, 1],
        [5, 0, 2, 3],
        [0, 6, 1, 2]
]
m,n = 4,4
print(getMaxGold(gold, n, m))
```

Assembly Line SchedulingProblem

```
def carAssembly(a, t, e, x):
    NUM_STATION = len(a[0])
    T1 = [0 for _ in range(NUM_STATION)]
    T2 = [0 for _ in range(NUM_STATION)]

    # time taken to leave first station in line 1
    T1[0] = e[0] + a[0][0]
    # time taken to leave first station in line 2
```

```
    T2[0] = e[1] + a[1][0]

    # Fill tables T1[] and T2[] using above given recursive relations
    for i in range(1, NUM_STATION):
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i])
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i] )

    # consider exit times and return minimum
    return min(T1[NUM_STATION - 1] + x[0], T2[NUM_STATION - 1] + x[1])

a = [[4, 5, 3, 2],
     [2, 10, 1, 4]]
t = [[0, 7, 4, 5],
     [0, 9, 2, 8]]
e = [10, 12]
x = [18, 7]

print(carAssembly(a, t, e, x))
```

Painting the Fence Problem

```
'''
Given a fence with n posts and k colors, find out the number of ways of painting the fence such that at most 2 adjacent posts have the same color. Since answer can be large return it modulo 10^9 + 7.
Examples:

Input : n = 2 k = 4
Output : 16
We have 4 colors and 2 posts.
Ways when both posts have same color : 4
Ways when both posts have diff color :
4(choices for 1st post) * 3(choices for
2nd post) = 12

Input : n = 3 k = 2
Output : 6
'''



# Returns count of ways to color k posts using k colors
def countWays(n, k):
    # There are k ways to color first post
    total = k
    mod = 1000000007

    # There are 0 ways for single post to violate (same color_ and k ways to not violate (different color)
    same, diff = 0, k

    # Fill for 2 posts onwards
    for _ in range(2, n + 1):
        # Current same is same as previous diff
        same = diff

        # We always have k-1 choices for next post
        diff = total * (k - 1)
        diff = diff % mod
```

```
        # Total choices till i.
        total = (same + diff) % mod
    return total

n, k = 3, 2
print(countWays(n, k))
```

Rod Cutting Problem

```
'''
Given a rod of length n and a list of rod prices of length i, where 1 <= i <= n, find the optimal
way to cut the rod into smaller rods to maximize profit.
For example, consider the following rod lengths and values:
Input:
length[] = [1, 2, 3, 4, 5, 6, 7, 8]
price[] = [1, 5, 8, 9, 10, 17, 17, 20]
Rod length: 4
Best: Cut the rod into two pieces of length 2 each to gain revenue of 5 + 5 = 10
'''

def rodCut(price, n):

    # `T[i]` stores the maximum profit achieved from a rod of length `i`
    T = [0] * (n + 1)

    # consider a rod of length `i`
    for i in range(1, n + 1):
        # divide the rod of length `i` into two rods of length `j` and `i-j` each and take maximum
        for j in range(1, i + 1):
            T[i] = max(T[i], price[j - 1] + T[i - j])

    # `T[n]` stores the maximum profit achieved from a rod of length `n`
    return T[n]

price = [1, 5, 8, 9, 10, 17, 17, 20]
n = 4        # rod length
print('Profit is', rodCut(price, n))
```

Longest Common Subsequence

```
# Function to return all LCS of substrings `X[0…m-1]`, `Y[0…n-1]`
def LCS(X, Y, m, n, lookup):
    # if the end of either sequence is reached
    if m == 0 or n == 0:
        # create a list with one empty string and return
        return ['']

    # if the last character of `X` and `Y` matches
    if X[m - 1] == Y[n - 1]:
        # ignore the last characters of `X` and `Y` and find all LCS of substring
        # `X[0…m-2]`, `Y[0…n-2]` and store it in a list
        lcs = LCS(X, Y, m - 1, n - 1, lookup)

        # append current character `X[m-1]` or `Y[n-1]`
        # to all LCS of substring `X[0…m-2]` and `Y[0…n-2]`
        for i in range(len(lcs)):
            lcs[i] = lcs[i] + (X[m - 1])
```

```
        return lcs

    # we reach here when the last character of `X` and `Y` don't match

    # if a top cell of the current cell has more value than the left cell,
    # then ignore the current character of string `X` and find all LCS of
    # substring `X[0…m-2]`, `Y[0…n-1]`
    if lookup[m - 1][n] > lookup[m][n - 1]:
        return LCS(X, Y, m - 1, n, lookup)

    # if a left cell of the current cell has more value than the top cell,
    # then ignore the current character of string `Y` and find all LCS of
    # substring `X[0…m-1]`, `Y[0…n-2]`
    if lookup[m][n - 1] > lookup[m - 1][n]:
        return LCS(X, Y, m, n - 1, lookup)

    # if the top cell has equal value to the left cell, then consider both characters

    top = LCS(X, Y, m - 1, n, lookup)
    left = LCS(X, Y, m, n - 1, lookup)

    # merge two lists and return
    return top + left


# Function to fill the lookup table by finding the length of LCS
# of substring `X` and `Y`
def LCSLength(X, Y, lookup):

    # fill the lookup table in a bottom-up manner
    for i in range(1, len(X) + 1):
        for j in range(1, len(Y) + 1):
            # if current character of `X` and `Y` matches
            if X[i - 1] == Y[j - 1]:
                lookup[i][j] = lookup[i - 1][j - 1] + 1

            # otherwise, if the current character of `X` and `Y` don't match
            else:
                lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1])


# Function to find all LCS of string `X[0…m-1]` and `Y[0…n-1]`
def findLCS(X, Y):

    # lookup[i][j] stores the length of LCS of substring `X[0…i-1]` and `Y[0…j-1]`
    lookup = [[0 for _ in range(len(Y) + 1)] for _ in range(len(X) + 1)]

    # fill lookup table
    LCSLength(X, Y, lookup)

    # find all the longest common subsequences
    lcs = LCS(X, Y, len(X), len(Y), lookup)

    # since a list can contain duplicates, "convert" it to a set and return
    return set(lcs)

X = 'ABCBDAB'
Y = 'BDCABA'
lcs = findLCS(X, Y)
print(lcs)
```

Longest Repeated Subsequence

```
def LRS(X, m, n, lookup):
    # if the end of either sequence is reached, return an empty string
    if m == 0 or n == 0:
        return ''

    if X[m - 1] == X[n - 1] and m != n:
        return LRS(X, m - 1, n - 1, lookup) + X[m - 1]
    # otherwise, if characters at index `m` and `n` don't match
    if lookup[m - 1][n] > lookup[m][n - 1]:
        return LRS(X, m - 1, n, lookup)
    else:
        return LRS(X, m, n - 1, lookup)


# Function to fill the lookup table by finding the length of LRS of substring `X[0…n-1]`
def LRSLength(X, lookup):
    # Fill the lookup table in a bottom-up manner.The first row and first column of the lookup
table are already 0.
    for i in range(1, len(X) + 1):
        for j in range(1, len(X) + 1):
            # if characters at index `i` and `j` matches and the index are different
            if X[i - 1] == X[j - 1] and i != j:
                lookup[i][j] = lookup[i - 1][j - 1] + 1
            # otherwise, if characters at index `i` and `j` are different
            else:
                lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1])

X = 'ATACTCGGA'
# lookup[i][j] stores the length of LRS of substring `X[0…i-1]` and `X[0…j-1]`
lookup = [[0 for _ in range(len(X) + 1)] for _ in range(len(X) + 1)]
# fill lookup table
LRSLength(X, lookup)
# find the longest repeating subsequence
print(LRS(X, len(X), len(X), lookup))
```

Longest Increasing Subsequence

```
def findLIS(arr):
    if not arr:
        return []

    # LIS[i] stores the longest increasing subsequence of sublist `arr[0…i]` that ends with
`arr[i]`
    LIS = [[] for _ in range(len(arr))]

    # LIS[0] denotes the longest increasing subsequence ending at `arr[0]`
    LIS[0].append(arr[0])

    # start from the second element in the list
    for i in range(1, len(arr)):
        # do for each element in sublist `arr[0…i-1]`
        for j in range(i):

            # find the longest increasing subsequence that ends with `arr[j]`
            # where `arr[j]` is less than the current element `arr[i]`
```

```
            if arr[j] < arr[i] and len(LIS[j]) > len(LIS[i]):
                LIS[i] = LIS[j].copy()

        # include `arr[i]` in `LIS[i]`
        LIS[i].append(arr[i])

    # `j` will store the index of LIS
    j = 0
    for i in range(len(arr)):
        if len(LIS[j]) < len(LIS[i]):
            j = i
    print(LIS[j])

arr = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]
findLIS(arr)
```

Space Optimized Solution of LCS (Print only length)

```
def lcs(text1, text2):
    m, n = len(text1), len(text2)
    if m > n : text1, text2 = text2, text1
    dp = [0] * (n + 1)
    for c in text1:
        prev = 0
        for i, d in enumerate(text2):
            prev, dp[i + 1] = dp[i + 1], prev + 1 if c == d else max(dp[i], dp[i + 1])
    return dp[-1]
X = "AGGTAB"
Y = "GXTXAYB"

print("Length of LCS is", lcs(X, Y))
```

LCS (Longest Common Subsequence) of three strings

```
"""
Given 3 strings of all having length < 100,the task is to find the longest common sub-sequence in
all three given sequences.

Examples:

Input : str1 = "geeks"
        str2 = "geeksfor"
        str3 = "geeksforgeeks"
Output : 5
Longest common subsequence is "geeks"
i.e., length = 5
"""


X = "AGGT12"
Y = "12TXAYB"
Z = "12XBA"

dp = [[[-1 for _ in range(100)] for _ in range(100)] for _ in range(100)]

# Returns length of LCS for X[0..m-1], Y[0..n-1] and Z[0..o-1]
def lcsOf3(i, j, k):

    if(i == -1 or j == -1 or k == -1) :
```

```
        return 0

    if(dp[i][j][k] != -1) :
        return dp[i][j][k]

    if X[i] == Y[j] == Z[k]:
        dp[i][j][k] = 1 + lcsOf3(i - 1, j - 1, k - 1)

    else:
        dp[i][j][k] = max(max(lcsOf3(i - 1, j, k), lcsOf3(i, j - 1, k)), lcsOf3(i, j, k - 1))

    return dp[i][j][k]


m = len(X)
n = len(Y)
o = len(Z)
print("Length of LCS is", lcsOf3(m - 1, n - 1, o - 1))
```

Maximum Sum Increasing Subsequence

```
"""
Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence
of the given array such that the integers in the subsequence are sorted in increasing order. For
example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if
the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array
is {10, 5, 4, 3}, then output should be 10
"""

def maxSumIS(arr, n):
    maxx = 0
    msis = [0 for _ in range(n)]

    # Initialize msis values for all indexes
    for i in range(n):
        msis[i] = arr[i]

    # Compute maximum sum values in bottom up manner
    for i in range(1, n):
        for j in range(i):
            if (arr[i] > arr[j] and
                msis[i] < msis[j] + arr[i]):
                msis[i] = msis[j] + arr[i]

    # Pick maximum of all msis values
    for i in range(n):
        if maxx < msis[i]:
            maxx = msis[i]
    return maxx

arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing " + "subsequence is " +str(maxSumIS(arr, n)))
```

Count all subsequences having product less than K

```
"""
Input : [1, 2, 3, 4]
```

```
        k = 10
Output :11
Explanation: The subsequences are {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {1,
2, 3}, {1, 2, 4}

Input  : [4, 8, 7, 2]
         k = 50
Output : 9
"""

def productSubSeqCount(arr, k):
    n = len(arr)
    dp = [[0 for _ in range(n + 1)] for _ in range(k + 1)]
    for i in range(1, k + 1):
        for j in range(1, n + 1):

            # number of subsequence using j-1 terms
            dp[i][j] = dp[i][j - 1]

            # if arr[j-1] > i it will surely make product greater thus it won't contribute then
            if arr[j - 1] <= i and arr[j - 1] > 0:

                # number of subsequence using 1 to j-1 terms and j-th term
                dp[i][j] += dp[i // arr[j - 1]][j - 1] + 1
    return dp[k][n]

A = [1,2,3,4]
k = 10
print(productSubSeqCount(A, k))
```

Longest subsequence such that difference between adjacent is one

```
"""
Input :  arr[] = {10, 9, 4, 5, 4, 8, 6}
Output :  3
As longest subsequences with difference 1 are, "10, 9, 8",
"4, 5, 4" and "4, 5, 6"

Input :  arr[] = {1, 2, 3, 2, 3, 7, 2, 1}
Output :  7
As longest consecutive sequence is "1, 2, 3, 2, 3, 2, 1"
"""

def longestSubseqWithDiffOne(arr, n):
    # Initialize the dp[] array with 1 as a single element will be of 1 length
    dp = [1 for _ in range(n)]

    # Start traversing the given array
    for i in range(n):
        # Compare with all the previous elements
        for j in range(i):
            # If the element is consecutive then consider this subsequence and update dp[i] if
required.
            if arr[i] in [arr[j] + 1, arr[j] - 1]:
                dp[i] = max(dp[i], dp[j]+1)

    # Longest length will be the maximum value of dp array.
```

```
    result = 1
    for i in range(n):
        if (result < dp[i]):
            result = dp[i]
    return result

arr = [1, 2, 3, 4, 5, 3, 2]
# Longest subsequence with one difference is {1, 2, 3, 4, 3, 2}
n = len(arr)
print (longestSubseqWithDiffOne(arr, n))
```

Maximum subsequence sum such that no three are consecutive

```
# sourcery skip: avoid-builtin-shadow
"""
Input: arr[] = {1, 2, 3}
Output: 5
We can't take three of them, so answer is
2 + 3 = 5

Input: arr[] = {3000, 2000, 1000, 3, 10}
Output: 5013
3000 + 2000 + 3 + 10 = 5013
"""
arr = [100, 1000, 100, 1000, 1]
sum = [-1] * 10000

# Returns maximum subsequence sum such
# that no three elements are consecutive
def maxSumWO3Consec(n) :
    if(sum[n] != -1):
        return sum[n]

    # 3 Base cases (process first three elements)
    if(n == 0) :
        sum[n] = 0
        return sum[n]

    if(n == 1) :
        sum[n] = arr[0]
        return sum[n]

    if(n == 2) :
        sum[n] = arr[1] + arr[0]
        return sum[n]

    # Process rest of the elements We have three cases
    sum[n] = max(max(maxSumWO3Consec(n - 1), maxSumWO3Consec(n - 2) + arr[n-1]), arr[n-1] + arr[n
- 2] + maxSumWO3Consec(n - 3))
    return sum[n]

n = len(arr)
print(maxSumWO3Consec(n))
```

Egg Dropping Problem

```python
import sys

# Function to get minimum number of trials needed in worst case with n eggs and k floors
def eggDrop(n, k):
    # If there are no floors, then no trials needed. OR if there is one floor, one trial needed.
    if k in [1, 0]:
        return k

    # We need k trials for one egg and k floors
    if (n == 1):
        return k
    min = sys.maxsize

    # Consider all droppings from 1st floor to kth floor and return the minimum of these values
plus 1.
    for x in range(1, k + 1):
        res = max(eggDrop(n - 1, x - 1),
                  eggDrop(n, k - x))
        if (res < min):
            min = res
    return min + 1

n = 2
k = 10
print("Minimum number of trials in worst case with", n, "eggs and", k, "floors is", eggDrop(n, k))
```

Maximum Length Chain of Pairs

```python
"""
You are given n pairs of numbers. In every pair, the first number is always smaller than the
second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed
in this fashion. Find the longest chain which can be formed from a given set of pairs.

For example, if the given pairs are {{5, 24}, {39, 60}, {15, 28}, {27, 40}, {50, 90} }, then the
longest chain that can be formed is of length 3, and the chain is {{5, 24}, {27, 40}, {50, 90}}
"""



class Pair(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

# This function assumes that arr[] is sorted in increasing
# order according the first (or smaller) values in pairs.
def maxChainLength(arr, n):
    max = 0

    # Initialize MCL(max chain length) values for all indices
    mcl = [1 for _ in range(n)]

    # Compute optimized chain length values in bottom up manner
    for i in range(1, n):
        for j in range(i):
            if (arr[i].a > arr[j].b and
                mcl[i] < mcl[j] + 1):
                mcl[i] = mcl[j] + 1

    # mcl[i] now stores the maximum chain length ending with pair i
```

```
    # Pick maximum of all MCL values
    for i in range(n):
        if (max < mcl[i]):
            max = mcl[i]
    return max

arr = [Pair(5, 24), Pair(15, 25),
    Pair(27, 40), Pair(50, 60)]
print('Length of maximum size chain is', maxChainLength(arr, len(arr)))
```

Maximum size square sub-matrix with all 1s

```
R = 6
C = 5

def printMaxSubSquare(M):
    global R,C
    Max = 0
    # set all elements of S to 0 first
    S = [[0 for _ in range(C)] for _ in range(2)]

    # Construct the entries
    for i in range(R):
        for j in range(C):

            # Compute the entrie at the current position
            Entrie = M[i][j]
            if Entrie and j:
                Entrie = 1 + min(S[1][j - 1],min(S[0][j - 1], S[1][j]))

            # Save the last entrie and add the new one
            S[0][j] = S[1][j]
            S[1][j] = Entrie

            # Keep track of the max square length
            Max = max(Max, Entrie)

    # Print the square
    print("Maximum size sub-matrix is: ")
    for _ in range(Max):
        for _ in range(Max):
            print("1",end=" ")
        print()

M = [[0, 1, 1, 0, 1],
    [1, 1, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0]]
printMaxSubSquare(M)
```

Maximum sum of pairs with specific difference

```
"""
Input  : arr[] = {3, 5, 10, 15, 17, 12, 9}, K = 4
Output : 62
Explanation:
```

```
Then disjoint pairs with difference less than K are, (3, 5), (10, 12), (15, 17)
So maximum sum which we can get is 3 + 5 + 12 + 10 + 15 + 17 = 62
Note that an alternate way to form disjoint pairs is, (3, 5), (9, 12), (15, 17), but this pairing
produces lesser sum.

Input  : arr[] = {5, 15, 10, 300}, k = 12
Output : 25
"""

def maxSumPairWithDifferenceLessThanK(arr, N, k):
    maxSum = 0

    # Sort elements to ensure every i and i-1 is closest possible pair
    arr.sort()

    # To get maximum possible sum, iterate from largest to smallest, giving larger numbers
priority over smaller numbers.
    i = N - 1
    while (i > 0):

        # Case I: Diff of arr[i] and arr[i-1] is less than K, add to maxSum
        # Case II: Diff between arr[i] and arr[i-1] is not less than K, move to next i since with
sorting we know,
        # arr[i]-arr[i-1] < arr[i]-arr[i-2] and so on.

        if (arr[i] - arr[i - 1] < k):
            # Assuming only positive numbers.
            maxSum += arr[i]
            maxSum += arr[i - 1]

            # When a match is found skip this pair
            i -= 1
        i -= 1
    return maxSum

arr = [3, 5, 10, 15, 17, 12, 9]
N = len(arr)
K = 4
print(maxSumPairWithDifferenceLessThanK(arr, N, K))
```

Min Cost PathProblem

```
"""
Given a n*n matrix where all numbers are distinct, find the maximum length path (starting from any
cell) such that all cells along the path are in increasing order with a difference of 1.
We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or
(i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1.

Example:

Input:  mat[][] = {{1, 2, 9}
                   {5, 3, 8}
                   {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.
"""
n = 3
# Returns length of the longest path beginning with mat[i][j]. This function mainly uses lookup
table dp[n][n]
```

```
def findLongestFromACell(i, j, mat, dp):
    if (i < 0 or i >= n or j < 0 or j >= n):
        return 0

    # If this subproblem is already solved
    if (dp[i][j] != -1):
        return dp[i][j]

    # To store the path lengths in all the four directions
    x, y, z, w = -1, -1, -1, -1

    # Since all numbers are unique and in range from 1 to n * n,
    # there is atmost one possible direction from any cell
    if (j < n-1 and ((mat[i][j] + 1) == mat[i][j + 1])):
        x = 1 + findLongestFromACell(i, j + 1, mat, dp)

    if (j > 0 and (mat[i][j] + 1 == mat[i][j-1])):
        y = 1 + findLongestFromACell(i, j-1, mat, dp)

    if (i > 0 and (mat[i][j] + 1 == mat[i-1][j])):
        z = 1 + findLongestFromACell(i-1, j, mat, dp)

    if (i < n-1 and (mat[i][j] + 1 == mat[i + 1][j])):
        w = 1 + findLongestFromACell(i + 1, j, mat, dp)

    # If none of the adjacent fours is one greater we will take 1
    # otherwise we will pick maximum from all the four directions
    dp[i][j] = max(x, max(y, max(z, max(w, 1))))
    return dp[i][j]

# Returns length of the longest path beginning with any cell
def finLongestOverAll(mat):
    result = 1 # Initialize result

    # Create a lookup table and fill all entries in it as -1
    dp = [[-1 for _ in range(n)] for _ in range(n)]

    # Compute longest path beginning from all cells
    for i in range(n):
        for j in range(n):
            if (dp[i][j] == -1):
                findLongestFromACell(i, j, mat, dp)
            # Update result if needed
            result = max(result, dp[i][j])
    return result

mat = [[1, 2, 9],
    [5, 3, 8],
    [4, 6, 7]]
print("Length of the longest path is ", finLongestOverAll(mat))
```

Maximum difference of zeros and ones in binary string

```
"""
Given a binary string of 0s and 1s. The task is to find the length of the substring which is
having a maximum difference between the number of 0s and the number of 1s (number of 0s - number
of 1s). In case of all 1s print -1.

Examples:
```

```
Input : S = "11000010001"
Output : 6
From index 2 to index 9, there are 7
0s and 1 1s, so number of 0s - number
of 1s is 6.
Input : S = "1111"
Output : -1
"""


MAX = 100

# Return true if there all 1s
def allones(s, n):
    # Checking each index is 0 or not.
    co = sum(1 if i == '1' else 0 for i in s)
    return co == n

# Find the length of substring with maximum difference of zeroes and ones in binary string
def findlength(arr, s, n, ind, st, dp):

    # If string is over
    if ind >= n:
        return 0

    # If the state is already calculated.
    if dp[ind][st] != -1:
        return dp[ind][st]

    if not st:
        dp[ind][st] = max(arr[ind] +
        findlength(arr, s, n, ind + 1, 1, dp),
            (findlength(arr, s, n, ind + 1, 0, dp)))
    else:
        dp[ind][st] = max(arr[ind] +
        findlength(arr, s, n, ind + 1, 1, dp), 0)
    return dp[ind][st]

# Returns length of substring which is having maximum difference of number of 0s and number of 1s
def maxLen(s, n):
    # If all 1s return -1.
    if allones(s, n):
        return -1

    # Else find the length.
    arr = [0] * MAX
    for i in range(n):
        arr[i] = 1 if s[i] == '0' else -1
    dp = [[-1] * 3 for _ in range(MAX)]
    return findlength(arr, s, n, 0, 0, dp)

s = "11000010001"
n = 11
print(maxLen(s, n))
```

Minimum number of jumps to reach end

```
"""
```

```
Given an array of integers where each element represents the max number of steps that can be made
forward from that element. Write a function to return the minimum number of jumps to reach the end
of the array (starting from the first element). If an element is 0, then we cannot move through
that element. If we can't reach the end, return -1.

Examples:
    Input:  arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
    Output: 3 (1-> 3 -> 8 -> 9)
    Explanation: Jump from 1st element to
    2nd element as there is only 1 step,
    now there are three options 5, 8 or 9.
    If 8 or 9 is chosen then the end node 9
    can be reached. So 3 jumps are made.

    Input:  arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
    Output: 10
    Explanation: In every step a jump is
    needed so the count of jumps is 10.
"""


def minJumps(arr, n):
  # The number of jumps needed to reach the starting index is 0
  if (n <= 1):
    return 0

  # Return -1 if not possible to jump
  if (arr[0] == 0):
    return -1

  # initialization
  maxReach = arr[0]    # stores all time the maximal reachable index in the array
  step = arr[0]   # stores the amount of steps we can still take
  jump = 1   # stores the amount of jumps necessary to reach that maximal reachable position

  # Start traversing array

  for i in range(1, n):
    # Check if we have reached the end of the array
    if (i == n-1):
      return jump

    # updating maxReach
    maxReach = max(maxReach, i + arr[i])

    # we use a step to get to the current index
    step -= 1;

    # If no further steps left
    if (step == 0):
      # we must have used a jump
      jump += 1

      # Check if the current index / position or lesser index is the maximum reach point from the
previous indexes
      if(i >= maxReach):
        return -1

      # re-initialize the steps to the amount of steps to reach maxReach from position i.
      step = maxReach - i;
  return -1
```

```
arr = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]
size = len(arr)
print("Minimum number of jumps to reach end is % d " % minJumps(arr, size))
```

Minimum cost to fill given weight in a bag

```
"""
You are given a bag of size W kg and you are provided costs of packets different weights of
oranges in array cost[] where cost[i] is basically the cost of ‘i’ kg packet of oranges. Where
cost[i] = -1 means that ‘i’ kg packet of orange is unavailable
Find the minimum total cost to buy exactly W kg oranges and if it is not possible to buy exactly W
kg oranges then print -1. It may be assumed that there is an infinite supply of all available
packet types.
Note: array starts from index 1.

Examples:

Input  : W = 5, cost[] = {20, 10, 4, 50, 100}
Output : 14
We can choose two oranges to minimize cost. First
orange of 2Kg and cost 10. Second orange of 3Kg
and cost 4.

Input  : W = 5, cost[] = {1, 10, 4, 50, 100}
Output : 5
We can choose five oranges of weight 1 kg.
"""

import sys

# Returns the best obtainable price for a rod of length n and price[] as prices of different
pieces
def minCost(cost, n):
    dp = [0 for _ in range(n + 1)]
    # Build the table val[] in bottom up manner and return the last entry from the table
    for i in range(1, n + 1):
        min_cost = sys.maxsize
        for j in range(i):
            if j<len(cost) and cost[j]!=-1:
                min_cost = min(min_cost, cost[j] + dp[i - j - 1])
        dp[i] = min_cost

    return dp[n]

cost = [ 10,-1,-1,-1,-1 ]
W = len(cost)
print(minCost(cost, W))
```

Minimum removals from array to make max –min <= K

```
"""
Input : a[] = {1, 3, 4, 9, 10, 11, 12, 17, 20}
          k = 4
Output : 5
Explanation: Remove 1, 3, 4 from beginning
and 17, 20 from the end.
```

```
Input : a[] = {1, 5, 6, 2, 8}  K=2
Output : 3
Explanation: There are multiple ways to
remove elements in this case.
One among them is to remove 5, 6, 8.
The other is to remove 1, 2, 5
"""
def removal(a, n, k):
    # sort the array
    a.sort()
    # to store the max length of array with difference <= k
    maxLen = 0
    # pointer to keep track of starting of each subarray
    i = 0
    for j in range(i+1, n):
        # if the subarray from i to j index is valid the store it's length
        if a[j]-a[i] <= k:
            maxLen = max(maxLen, j-i+1)
        else:
            i += 1
        if i >= n:
            break
    return n-maxLen

a = [1, 3, 4, 9, 10, 11, 12, 17, 20]
n = len(a)
k = 4
print(removal(a, n, k))
```

Longest Common Substring

```
"""
Input : X = “GeeksforGeeks”, y = “GeeksQuiz”
Output : 5
Explanation:
The longest common substring is “Geeks” and is of length 5.
"""

def lcs(i, j, count):
    if (i == 0 or j == 0):
        return count
    if (X[i - 1] == Y[j - 1]):
        count = lcs(i - 1, j - 1, count + 1)
    count = max(count, max(lcs(i, j - 1, 0), lcs(i - 1, j, 0)))
    return count

X = "abcdxyz"
Y = "xyzabcd"
n = len(X)
m = len(Y)
print(lcs(n, m, 0))
```

Count number of ways to reacha given score in a game

```
"""
Consider a game where a player can score 3 or 5 or 10 points in a move. Given a total score n,
find number of ways to reach the given score.
```

```
Examples:

Input: n = 20
Output: 4
There are following 4 ways to reach 20
(10, 10)
(5, 5, 10)
(5, 5, 5, 5)
(3, 3, 3, 3, 3, 5)
"""

def count(n):
    # table[i] will store count of solutions for value i. Initialize all table values as 0.
    table = [0 for _ in range(n+1)]

    # Base case (If given value is 0)
    table[0] = 1

    # One by one consider given 3 moves and update the table[] values after the index greater than
    # or equal to the value of the picked move.
    for i in range(3, n+1):
        table[i] += table[i-3]
    for i in range(5, n+1):
        table[i] += table[i-5]
    for i in range(10, n+1):
        table[i] += table[i-10]
    return table[n]

n = 20
print('Count for', n, 'is', count(n))
n = 13
print('Count for', n, 'is', count(n))
```

Count Balanced Binary Trees of Height h

```
"""
Given a height h, count and return the maximum number of balanced binary trees possible with
height h. A balanced binary tree is one in which for every node, the difference between heights of
left and right subtree is not more than 1.

Input : h = 3
Output : 15

Input : h = 4
Output : 315
"""
def countBT(h) :
    BIG_PRIME = 1000000007
    if h < 2:
          return 1
    dp0 = dp1 = 1
    dp2 = 3
    for _ in range(2,h+1):
        dp2 = (dp1*dp1 + 2*dp1*dp0)%BIG_PRIME
        dp0 = dp1
        dp1 = dp2
    return dp2

h = 3
```

```
print(f"No. of balanced binary trees of height {h} is: {str(countBT(h))}")
```

Smallest sum contiguous subarray

```
"""
Given an array containing n integers. The problem is to find the sum of the elements of the
contiguous subarray having the smallest(minimum) sum.
Examples:
Input : arr[] = {3, -4, 2, -3, -1, 7, -5}
Output : -6
Subarray is {-4, 2, -3, -1} = -6
"""

maxsize=float('inf')

def smallestSumSubarr(arr, n):
    # to store the minimum value that is ending up to the current index
    min_ending_here = maxsize

    # to store the minimum value encountered so far
    min_so_far = maxsize

    # traverse the array elements
    for i in range(n):
        # if min_ending_here > 0, then it could not possibly contribute to the minimum sum further
        if (min_ending_here > 0):
            min_ending_here = arr[i]

        # else add the value arr[i] to min_ending_here
        else:
            min_ending_here += arr[i]

        # update min_so_far
        min_so_far = min(min_so_far, min_ending_here)
    return min_so_far


arr = [3, -4, 2, -3, -1, 7, -5]
n = len(arr)
print ("Smallest sum: ", smallestSumSubarr(arr, n))
```

Unbounded Knapsack (Repetition of items allowed)

```
"""
Given a knapsack weight W and a set of n items with certain value vali and weight wti, we need to
calculate the maximum amount that could make up this quantity exactly. This is different from
classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.
Examples:

Input : W = 100
       val[]  = {1, 30}
       wt[] = {1, 50}
Output : 100
There are many ways to fill knapsack.
1) 2 instances of 50 unit weight item.
2) 100 instances of 1 unit weight item.
3) 1 instance of 50 unit weight item and 50
   instances of 1 unit weight items.
```

```
We get maximum value with option 2.
"""
def unboundedKnapsack(W, n, val, wt):
    # dp[i] is going to store maximum value with knapsack capacity i.
    dp = [0 for _ in range(W + 1)]
    ans = 0

    # Fill dp[] using above recursive formula
    for i in range(W + 1):
        for j in range(n):
            if (wt[j] <= i):
                dp[i] = max(dp[i], dp[i - wt[j]] + val[j])
    return dp[W]

W = 100
val = [10, 30, 20]
wt = [5, 10, 15]
n = len(val)
print(unboundedKnapsack(W, n, val, wt))
```

Largest Independent Set Problem

```
"""
Given a Binary Tree, find size of the Largest Independent Set(LIS) in it. A subset of all tree
nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60,
70, 80} and size of the LIS is 5.
"""

class node:
    def __init__(self, data):

        self.data = data
        self.left = self.right = None
        self.liss = 0

# A memoization function returns size of the largest independent set in a given binary tree
def liss(root):
    if root is None:
        return 0
    if root.liss != 0:
        return root.liss
    if root.left is None and root.right is None:
        root.liss = 1
        return root.liss

    # Calculate size excluding the current node
    liss_excl = (liss(root.left) + liss(root.right))

    # Calculate size including the current node
    liss_incl = 1
    if root.left != None:
        liss_incl += (liss(root.left.left) + liss(root.left.right))

    if root.right != None:
        liss_incl += (liss(root.right.left) + liss(root.right.right))

    # Maximum of two sizes is LISS, store it for future uses.
```

```
    root.liss = max(liss_excl, liss_incl)
    return root.liss

root = node(20)
root.left = node(8)
root.left.left = node(4)
root.left.right = node(12)
root.left.right.left = node(10)
root.left.right.right = node(14)
root.right = node(22)
root.right.right = node(25)
print("Size of the Largest Independent Set is ", liss(root))
```

Partition problem

```
def isPossible(elements, target):
    dp = [False]*(target+1)
    dp[0] = True
    for ele in elements:
        for j in range(target, ele - 1, -1):
            if dp[j - ele]:
                dp[j] = True
    return dp[target]

arr = [6, 2, 5]
target = 7
if isPossible(arr, target):
    print("YES")
else:
    print("NO")
```

Longest Palindromic Subsequence

```

"""
As another example, if the given sequence is “BBABCBCAB”, then the output should be 7 as “BABCBAB”
is the longest palindromic subsequence in it. “BBBBB” and “BBCBB” are also palindromic
subsequences of the given sequence, but not the longest ones.
"""

dp = [[-1 for _ in range(1001)] for _ in range(1001)]
def lps(s1, s2, n1, n2):
    if (n1 == 0 or n2 == 0):
        return 0

    if (dp[n1][n2] != -1):
        return dp[n1][n2]

    if (s1[n1 - 1] == s2[n2 - 1]):
        dp[n1][n2] = 1 + lps(s1, s2, n1 - 1, n2 - 1)
    else:
        dp[n1][n2] = max(lps(s1, s2, n1 - 1, n2), lps(s1, s2, n1, n2 - 1))

    return dp[n1][n2]


seq = "GEEKSFORGEEKS"
n = len(seq)
```

```
s2 = seq
s2 = s2[::-1]
print(f"The length of the LPS is {lps(s2, seq, n, n)}")
```

Count All Palindromic Subsequence in a given String

```
"""
Input : str = "abcd"
Output : 4
Explanation :- palindromic  subsequence are : "a" ,"b", "c" ,"d"

Input : str = "aab"
Output : 4
Explanation :- palindromic subsequence are :"a", "a", "b", "aa"

Input : str = "aaaa"
Output : 15
"""

def countPS(i, j):
    if(i > j):
        return 0

    if(dp[i][j] != -1):
        return dp[i][j]

    if (i == j):
        dp[i][j] = 1
    elif str[i] == str[j]:
        dp[i][j] = (countPS(i + 1, j) + countPS(i, j - 1) + 1)
    else:
        dp[i][j] = (countPS(i + 1, j) + countPS(i, j - 1) - countPS(i + 1, j - 1))

    return dp[i][j]

str = "abcb" # remember to use variable name str otherwise program will fail
dp = [[-1 for _ in range(1000)] for _ in range(1000)]
n = len(str)
print("Total palindromic subsequence are :", countPS(0, n - 1))
```

Longest Palindromic Substring

```
"""
Suppose we have a string S. We have to find the longest palindromic substring in S. We are
assuming that the length of the string S is 1000. So if the string is “BABAC”, then the longest
palindromic substring is “BAB”.
"""

def longestPalindrome( s):
   dp = [[False for _ in range(len(s))] for _ in range(len(s))]
   for i in range(len(s)):
      dp[i][i] = True
   max_length = 1
   start = 0
   for l in range(2,len(s)+1):
      for i in range(len(s)-l+1):
         end = i+l
```

```
            if l==2:
                if s[i] == s[end-1]:
                    dp[i][end-1]=True
                    max_length = l
                    start = i
            elif s[i] == s[end-1] and dp[i+1][end-2]:
                dp[i][end-1]=True
                max_length = l
                start = i
    return s[start:start+max_length]

print(longestPalindrome("ABBABBC"))
```

Longest alternating subsequence

```
"""
Input: arr[] = {1, 5, 4}
Output: 3
The whole arrays is of the form  x1 < x2 > x3

Input: arr[] = {1, 4, 5}
Output: 2
All subsequences of length 2 are either of the form
x1 < x2; or x1 > x2

Input: arr[] = {10, 22, 9, 33, 49, 50, 31, 60}
Output: 6
The subsequences {10, 22, 9, 33, 31, 60} or
{10, 22, 9, 49, 31, 60} or {10, 22, 9, 50, 31, 60}
are longest subsequence of length 6.
"""

def LAS(arr, n):
    # "inc" and "dec" initialized as 1 as single element is still LAS
    inc = 1
    dec = 1

    # Iterate from second element
    for i in range(1,n):
        if (arr[i] > arr[i-1]):
            # "inc" changes iff "dec" changes
            inc = dec + 1

        elif (arr[i] < arr[i-1]):
            # "dec" changes iff "inc" changes
            dec = inc + 1

    # Return the maximum length
    return max(inc, dec)

arr = [10, 22, 9, 33, 49, 50, 31, 60]
n = len(arr)
print(LAS(arr, n))
```

Weighted Job Scheduling

```
"""
```

```
Given N jobs where every job is represented by following three elements of it.

Start Time
Finish Time
Profit or Value Associated (>= 0)
Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

Example:

Input: Number of Jobs n = 4
       Job Details {Start Time, Finish Time, Profit}
       Job 1:  {1, 2, 50}
       Job 2:  {3, 5, 20}
       Job 3:  {6, 19, 100}
       Job 4:  {2, 100, 200}
Output: The maximum profit is 250.
We can get the maximum profit by scheduling jobs 1 and 4.
Note that there is longer schedules possible Jobs 1, 2 and 3
but the profit with this schedule is 20+50+100 which is less than 250.
"""

# Importing the following module to sort array based on our custom comparison function
from functools import cmp_to_key

# A job has start time, finish time and profit
class Job:
    def __init__(self, start, finish, profit):
        self.start = start
        self.finish = finish
        self.profit = profit

# A utility function that is used for sorting events according to finish time
def jobComparator(s1, s2):
    return s1.finish < s2.finish

# Find the latest job (in sorted array) that doesn't conflict with the job[i]. If there is no
compatible job, then it returns -1
def latestNonConflict(arr, i):
    for j in range(i - 1, -1, -1):
        if arr[j].finish <= arr[i - 1].start:
            return j
    return -1

# A recursive function that returns the maximum possible profit from given array of jobs. The
array of jobs must be sorted according to finish time
def findMaxProfitRec(arr, n):
    # Base case
    if n == 1:
        return arr[n - 1].profit

    # Find profit when current job is included
    inclProf = arr[n - 1].profit
    i = latestNonConflict(arr, n)

    if i != -1:
        inclProf += findMaxProfitRec(arr, i + 1)

    # Find profit when current job is excluded
    exclProf = findMaxProfitRec(arr, n - 1)
    return max(inclProf, exclProf)
```

```
# The main function that returns the maximum possible profit from given array of jobs
def findMaxProfit(arr, n):

    # Sort jobs according to finish time
    arr = sorted(arr, key = cmp_to_key(jobComparator))
    return findMaxProfitRec(arr, n)

values = [ (3, 10, 20), (1, 2, 50), (6, 19, 100), (2, 100, 200) ]
arr = [Job(i[0], i[1], i[2]) for i in values]
n = len(arr)
print("The optimal profit is", findMaxProfit(arr, n))
```

Coin game winner where every player has three choices

```
"""
A and B are playing a game. At the beginning there are n coins. Given two more numbers x and y. In
each move a player can pick x or y or 1 coins. A always starts the game. The player who picks the
last coin wins the game or the person who is not able to pick any coin loses the game. For a given
value of n, find whether A will win the game or not if both are playing optimally.

Examples:

Input :  n = 5, x = 3, y = 4
Output : A
There are 5 coins, every player can pick 1 or
3 or 4 coins on his/her turn.
A can win by picking 3 coins in first chance.
Now 2 coins will be left so B will pick one
coin and now A can win by picking the last coin.

Input : 2 3 4
Output : B
"""

# To find winner of game
def findWinner(x, y, n):

    # To store results
    dp = [0 for _ in range(n + 1)]

    # Initial values
    dp[0] = False
    dp[1] = True

    # Computing other values.
    for i in range(2, n + 1):
        # If A losses any of i-1 or i-x or i-y game then he will definitely win game i
        if i >= 1 and not dp[i - 1]:
            dp[i] = True
        elif (i - x >= 0 and not dp[i - x]):
            dp[i] = True
        elif (i - y >= 0 and not dp[i - y]):
            dp[i] = True
        else:
            dp[i] = False

    # If dp[n] is true then A will game otherwise he losses
    return dp[n]
```

```
x = 3; y = 4; n = 5
if (findWinner(x, y, n)):
    print('A')
else:
    print('B')
```

Count Derangements (Permutation such that no element appears in its original position) [IMPORTANT]

```
"""
A and B are playing a game. At the beginning there are n coins. Given two more numbers x and y. In
each move a player can pick x or y or 1 coins. A always starts the game. The player who picks the
last coin wins the game or the person who is not able to pick any coin loses the game. For a given
value of n, find whether A will win the game or not if both are playing optimally.

Examples:

A Derangement is a permutation of n elements, such that no element appears in its original
position. For example, a derangement of {0, 1, 2, 3} is {2, 3, 1, 0}.
Given a number n, find the total number of Derangements of a set of n elements.

Examples :

Input: n = 2
Output: 1
For two elements say {0, 1}, there is only one
possible derangement {1, 0}

Input: n = 3
Output: 2
For three elements say {0, 1, 2}, there are two
possible derangements {2, 0, 1} and {1, 2, 0}
"""

def countDer(n):
    if n in [1, 2]:
        return n-1
    a = 0
    b = 1
    for i in range(3, n + 1):
        cur = (i-1)*(a+b)
        a = b
        b = cur
    return b

n = 4
print("Count of Derangements is ", countDer(n))
```

Maximum profit by buying and selling a share at most twice [IMP]

```
"""
In daily share trading, a buyer buys shares in the morning and sells them on the same day. If the
trader is allowed to make at most 2 transactions in a day, whereas the second transaction can only
start after the first one is complete (Buy->sell->Buy->sell). Given stock prices throughout the
day, find out the maximum profit that a share trader could have made.
```

```
Examples:

Input:   price[] = {10, 22, 5, 75, 65, 80}
Output:  87
Trader earns 87 as sum of 12, 75
Buy at 10, sell at 22,
Buy at 5 and sell at 80
Input:   price[] = {2, 30, 15, 10, 8, 25, 80}
Output:  100
Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30, buy at 8 and sell at 80
Input:   price[] = {100, 30, 15, 10, 8, 25, 80};
Output:  72
Buy at price 8 and sell at 80.
Input:   price[] = {90, 80, 70, 60, 50}
Output:  0
Not possible to earn.
"""

import sys
def maxtwobuysell(arr, size):
    first_buy = -sys.maxsize;
    first_sell = 0;
    second_buy = -sys.maxsize;
    second_sell = 0;

    for i in range(size):
        first_buy = max(first_buy, -arr[i]);
        first_sell = max(first_sell, first_buy + arr[i]);
        second_buy = max(second_buy, first_sell - arr[i]);
        second_sell = max(second_sell, second_buy + arr[i]);
    return second_sell;

arr = [ 2, 30, 15, 10, 8, 25, 80 ];
size = len(arr);
print(maxtwobuysell(arr, size));
```

Optimal Strategy for a Game

```
"""
Consider a row of n coins of values v1 . . . vn, where n is even. We play a game against an
opponent by alternating turns. In each turn, a player selects either the first or last coin from
the row, removes it from the row permanently, and receives the value of the coin. Determine the
maximum possible amount of money we can definitely win if we move first.
Note: The opponent is as clever as the user.

Let us understand the problem with few examples:

5, 3, 7, 10 : The user collects maximum value as 15(10 + 5)
8, 15, 3, 7 : The user collects maximum value as 22(7 + 15)
Does choosing the best at each move gives an optimal solution? No.
In the second example, this is how the game can be finished:

> User chooses 8.
> Opponent chooses 15.
> User chooses 7.
> Opponent chooses 3.
Total value collected by user is 15(8 + 7)
```

```
> User chooses 7.
> Opponent chooses 8.
> User chooses 15.
> Opponent chooses 3.
Total value collected by user is 22(7 + 15)
So if the user follows the second game state, the maximum value can be collected although the
first move is not the best.
"""
def optimalStrategyOfGame(arr, n):
    memo = {}
    # recursive top down memoized solution
    def solve(i, j):
        if i > j or i >= n or j < 0:
            return 0

        k = (i, j)
        if k in memo:
            return memo[k]

        # if the user chooses ith coin, the opponent can choose from i+1th or jth coin.
        # if he chooses i+1th coin, user is left with [i+2,j] range.
        # if opp chooses jth coin, then user is left with [i+1,j-1] range to choose from.
        # Also opponent tries to choose in such a way that the user has minimum value left.
        option1 = arr[i] + min(solve(i+2, j), solve(i+1, j-1))

        # if user chooses jth coin, opponent can choose ith coin or j-1th coin.
        # if opp chooses ith coin, user can choose in range [i+1,j-1].
        # if opp chooses j-1th coin, user can choose in range [i,j-2].
        option2 = arr[j] + min(solve(i+1, j-1), solve(i, j-2))

        # since the user wants to get maximum money
        memo[k] = max(option1, option2)
        return memo[k]

    return solve(0, n-1)

arr1 = [8, 15, 3, 7]
n = len(arr1)
print(optimalStrategyOfGame(arr1, n))

arr2 = [2, 2, 2, 2]
n = len(arr2)
print(optimalStrategyOfGame(arr2, n))

arr3 = [20, 30, 2, 2, 2, 10]
n = len(arr3)
print(optimalStrategyOfGame(arr3, n))
```

Optimal Binary Search Tree

```
"""
Given a sorted array key [0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts,
where freq[i] is the number of searches for keys[i]. Construct a binary search tree of all keys
such that the total cost of all the searches is as small as possible.
Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied
by its frequency. The level of the root is 1.

Examples:
```

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                        12
          \                      /
           12                  10
          I                      II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118
"""

def optCost(freq, i, j):
    if j < i:     # no elements in this subarray
        return 0
    if j == i:    # one element in this subarray
        return freq[i]

    # Get sum of freq[i], freq[i+1], ... freq[j]
    fsum = Sum(freq, i, j)

    # Initialize minimum value
    Min = float('inf')

    # One by one consider all elements as root and recursively find cost of the BST, compare the
cost with min and update min if needed
    for r in range(i, j + 1):
        cost = (optCost(freq, i, r - 1) +
                optCost(freq, r + 1, j))
        if cost < Min:
            Min = cost

    # Return minimum value
    return Min + fsum

# The main function that calculates minimum cost of a Binary Search Tree. It mainly uses optCost()
to find the optimal cost.
def optimalSearchTree(keys, freq, n):

    # Here array keys[] is assumed to be sorted in increasing order. If keys[]
    # is not sorted, then add code to sort keys, and rearrange freq[] accordingly.
    return optCost(freq, 0, n - 1)

# A utility function to get sum of array elements freq[i] to freq[j]
def Sum(freq, i, j):
    return sum(freq[k] for k in range(i, j + 1))

if __name__ == '__main__':
    keys = [10, 12, 20]
    freq = [34, 8, 50]
    n = len(keys)
    print("Cost of Optimal BST is",
        optimalSearchTree(keys, freq, n))
```

Palindrome PartitioningProblem

```
"""
Input : str = “geek”
Output : 2
```

```
We need to make minimum 2 cuts, i.e., “g ee k”
Input : str = “aaaa”
Output : 0
The string is already a palindrome.
Input : str = “abcde”
Output : 4
Input : str = “abbac”
Output : 1

"""


def isPalindrome(x):
    return x == x[::-1]

def minPalPartion(string, i, j):
    if i >= j or isPalindrome(string[i:j + 1]):
        return 0
    ans = float('inf')
    for k in range(i, j):
        count = (
            1 + minPalPartion(string, i, k)
            + minPalPartion(string, k + 1, j)
        )
        ans = min(ans, count)
    return ans

string = "ababbbabbababa"
print("Min cuts needed for Palindrome Partitioning is ", minPalPartion(string, 0, len(string) -
1))
```

Word Wrap Problem

```
"""
Given a sequence of words, and a limit on the number of characters that can be put in one line
(line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume
that the length of each word is smaller than the line width.
The word processors like MS Word do task of placing line breaks. The idea is to have balanced
lines. In other words, not have few lines with lots of extra spaces and some lines with small
amount of extra spaces.



The extra spaces includes spaces put at the end of every line except the last one.
The problem is to minimize the following total cost.
 Cost of a line = (Number of extra spaces in the line)^3
 Total Cost = Sum of costs for all lines

For example, consider the following string and line width M = 15
 "Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines
Geeks for Geeks
presents word
wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively.
So optimal value of total cost is 0 + 2*2*2 + 3*3*3 = 35
Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square
is also used) of extra spaces.
"""
```

```
# A Dynamic programming solution
# for Word Wrap Problem

# A utility function to print
# the solution
# l[] represents lengths of different
# words in input sequence. For example,
# l[] = {3, 2, 2, 5} is for a sentence
# like "aaa bb cc ddddd". n is size of
# l[] and M is line width (maximum no.
# of characters that can fit in a line)
INF = 2147483647
def printSolution(p, n):
    k = 0
    if p[n] == 1:
        k = 1
    else:
        k = printSolution(p, p[n] - 1) + 1
    print('Line number ', k, ': From word no. ',
                                 p[n], 'to ', n)
    return k

def solveWordWrap(l, n, M):
    # For simplicity, 1 extra space is used in all below arrays
    # extras[i][j] will have number of extra spaces if words from i to j are put in a single line
    extras = [[0 for _ in range(n + 1)] for _ in range(n + 1)]

    # lc[i][j] will have cost of a line which has words from i to j
    lc = [[0 for _ in range(n + 1)] for _ in range(n + 1)]

    # c[i] will have total cost of optimal arrangement of words from 1 to i
    c = [0 for _ in range(n + 1)]

    # p[] is used to print the solution.
    p = [0 for _ in range(n + 1)]

    # calculate extra spaces in a single line. The value extra[i][j] indicates
    # extra spaces if words from word number i to j are placed in a single line
    for i in range(n + 1):
        extras[i][i] = M - l[i - 1]
        for j in range(i + 1, n + 1):
            extras[i][j] = (extras[i][j - 1] -
                                    l[j - 1] - 1)

    # Calculate line cost corresponding to the above calculated extra
    # spaces. The value lc[i][j] indicates cost of putting words from word number i to j in a
single line
    for i in range(n + 1):
        for j in range(i, n + 1):
            if extras[i][j] < 0:
                lc[i][j] = INF;
            elif j == n:
                lc[i][j] = 0
            else:
                lc[i][j] = (extras[i][j] *
                                extras[i][j])

    # Calculate minimum cost and find minimum cost arrangement. The value
    # c[j] indicates optimized cost to arrange words from word number 1 to j.
```

```
    c[0] = 0
    for j in range(1, n + 1):
        c[j] = INF
        for i in range(1, j + 1):
            if (c[i - 1] != INF and
                lc[i][j] != INF and
                ((c[i - 1] + lc[i][j]) < c[j])):
                c[j] = c[i-1] + lc[i][j]
                p[j] = i
    printSolution(p, n)

l = [3, 2, 2, 5]
n = len(l)
M = 6
solveWordWrap(l, n, M)
```

Mobile Numeric Keypad Problem [IMP]

```
"""
Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to
the current button. You are not allowed to press bottom row corner buttons (i.e. * and # ).

Mobile-keypad

Given a number N, find out the number of possible numbers of given length.

Examples:

For N=1, number of possible numbers would be 10 (0, 1, 2, 3, …., 9)
For N=2, number of possible numbers would be 36
Possible numbers: 00,08 11,12,14 22,21,23,25 and so on.
If we start with 0, valid numbers will be 00, 08 (count: 2)
If we start with 1, valid numbers will be 11, 12, 14 (count: 3)
If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)
If we start with 3, valid numbers will be 33, 32, 36 (count: 3)
If we start with 4, valid numbers will be 44,41,45,47 (count: 4)
If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)
………………………………
………………………………
We need to print the count of possible numbers.
"""

# left, up, right, down move from current location
row = [0, 0, -1, 0, 1]
col = [0, -1, 0, 1, 0]

# Returns count of numbers of length n starting from key position (i, j) in a numeric keyboard.
def getCountUtil(keypad, i, j, n):
    if (keypad == None or n <= 0):
        return 0

    # From a given key, only one number is possible of length 1
    if (n == 1):
        return 1
    k = 0
    move = 0
    ro = 0
    co = 0
    totalCount = 0
```

```
    # move left, up, right, down from current location and if
    # new location is valid, then get number count of length
    # (n-1) from that new position and add in count obtained so far
    for move in range(5):
        ro = i + row[move]
        co = j + col[move]
        if (ro >= 0 and ro <= 3 and co >= 0 and co <= 2 and
                keypad[ro][co] != '*' and keypad[ro][co] != '#'):
            totalCount += getCountUtil(keypad, ro, co, n - 1)
    return totalCount

# Return count of all possible numbers of length n in a given numeric keyboard
def getCount(keypad, n):
    if keypad is None or n <= 0:
        return 0
    if (n == 1):
        return 10
    i = 0
    j = 0
    totalCount = 0
    for i in range(4): # Loop on keypad row
        for j in range(3): # Loop on keypad column
            # Process for 0 to 9 digits
            if (keypad[i][j] != '*' and keypad[i][j] != '#'):
            # Get count when number is starting from key position (i, j) and add in count obtained
so far
                totalCount += getCountUtil(keypad, i, j, n)
    return totalCount

keypad = [['1', '2', '3'],
        ['4', '5', '6'],
        ['7', '8', '9'],
        ['*', '0', '#']]
print("Count for numbers of length 1:", getCount(keypad, 1))
print("Count for numbers of length 2:", getCount(keypad, 2))
print("Count for numbers of length 3:", getCount(keypad, 3))
print("Count for numbers of length 4:", getCount(keypad, 4))
print("Count for numbers of length 5:", getCount(keypad, 5))
```

Boolean Parenthesization Problem

```
"""
Given a boolean expression with the following symbols.

Symbols
    'T' ---> true
    'F' ---> false
And following operators filled between symbols

Operators
    &   ---> boolean AND
    |   ---> boolean OR
    ^   ---> boolean XOR
Count the number of ways we can parenthesize the expression so that the value of expression
evaluates to true.
Let the input be in form of two arrays one contains the symbols (T and F) in order and the other
contains operators (&, | and ^}
```

```
Examples:

Input: symbol[]    = {T, F, T}
       operator[]  = {^, &}
Output: 2
The given expression is "T ^ F & T", it evaluates true
in two ways "((T ^ F) & T)" and "(T ^ (F & T))"

Input: symbol[]    = {T, F, F}
       operator[]  = {^, |}
Output: 2
The given expression is "T ^ F | F", it evaluates true
in two ways "( (T ^ F) | F )" and "( T ^ (F | F) )".

Input: symbol[]    = {T, T, F, T}
       operator[]  = {|, &, ^}
"""


def countParenth(symb, oper, n):
    F = [[0 for _ in range(n + 1)] for _ in range(n + 1)]
    T = [[0 for _ in range(n + 1)] for _ in range(n + 1)]

    # Fill diagonal entries first
    # All diagonal entries in T[i][i] are 1 if symbol[i] is T (true). Similarly, all F[i][i]
entries are 1 if
    # symbol[i] is F (False)
    for i in range(n):
        F[i][i] = 1 if symb[i] == 'F' else 0
        T[i][i] = 1 if symb[i] == 'T' else 0

    # Now fill T[i][i+1], T[i][i+2],
    # T[i][i+3]... in order And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for gap in range(1, n):
        for i, j in enumerate(range(gap, n)):
            T[i][j] = F[i][j] = 0
            for g in range(gap):

                # Find place of parenthesization using current value of gap
                k = i + g

                # Store Total[i][k] and Total[k+1][j]
                tik = T[i][k] + F[i][k]
                tkj = T[k + 1][j] + F[k + 1][j]

                # Follow the recursive formulas according to the current operator
                if oper[k] == '&':
                    T[i][j] += T[i][k] * T[k + 1][j]
                    F[i][j] += (tik * tkj - T[i][k] *
                                T[k + 1][j])
                if oper[k] == '|':
                    F[i][j] += F[i][k] * F[k + 1][j]
                    T[i][j] += (tik * tkj - F[i][k] *
                                F[k + 1][j])
                if oper[k] == '^':
                    T[i][j] += (F[i][k] * T[k + 1][j] +
                                T[i][k] * F[k + 1][j])
                    F[i][j] += (T[i][k] * T[k + 1][j] +
                                F[i][k] * F[k + 1][j])
    return T[0][n - 1]
```

```
symbols = "TTFT"
operators = "|&^"
n = len(symbols)

# There are 4 ways
# ((T|T)&(F^T)), (T|(T&(F^T))),
# (((T|T)&F)^T) and (T|((T&F)^T))
print(countParenth(symbols, operators, n))
```

Largest rectangular sub-matrix whose sum is 0

```
"""
Given a 2D matrix, find the number non-empty sub matrices, such that the sum of the elements
inside the sub matrix is equal to 0. (note:  elements might be negative).
"""
import itertools
def solve(A):
    if not A or not A[0]: return 0  # SC & guard
    cols = len(A[0]) + 1            # pad left to guard [c - 1]
    A = [[0] + row for row in A]
    for row, c in itertools.product(A, range(2, cols)):
        row[c] += row[c - 1]
    zeros = 0
    for c1 in range(cols - 1):                  # each pair of (c, c2]
        for c2 in range(c1 + 1, cols):
            sofar = 0
            seen = {0: 1}                   # {sum : cnt}, dict to cnt dups
            for row in A:                   # scan top-down as 1D sum 0
                sofar += row[c2] - row[c1]
                if sofar-0 in seen:
                    zeros += seen[sofar-0]
                if sofar in seen:
                    seen[sofar] += 1
                else: seen[sofar] = 1
    return zeros

A=[[-8, 5,  7],
[3 , 7, -8],
[5 ,-8,  9]
]
print(solve(A))
```

Maximum sum rectangle in a 2D matrix

```
# Implementation of Kadane's algorithm for 1D array. The function returns the maximum sum and
stores starting
#  and ending indexes of the maximum sum subarray at addresses pointed by start and finish
pointers respectively.
def kadane(arr, start, finish, n):
    Sum = 0
    maxSum = -999999999999
    i = None

    # Just some initial value to check for all negative values case
    finish[0] = -1

    # local variable
```

```
    local_start = 0

    for i in range(n):
        Sum += arr[i]
        if Sum < 0:
            Sum = 0
            local_start = i + 1
        elif Sum > maxSum:
            maxSum = Sum
            start[0] = local_start
            finish[0] = i

    # There is at-least one non-negative number
    if finish[0] != -1:
        return maxSum

    # Special Case: When all numbers in arr[] are negative
    maxSum = arr[0]
    start[0] = finish[0] = 0

    # Find the maximum element in array
    for i in range(1, n):
        if arr[i] > maxSum:
            maxSum = arr[i]
            start[0] = finish[0] = i
    return maxSum

def findMaxSum(M):
    global ROW, COL

    # Variables to store the final output
    maxSum, finalLeft = -999999999999, None
    finalRight, finalTop, finalBottom = None, None, None
    left, right, i = None, None, None

    temp = [None] * ROW
    Sum = 0
    start = [0]
    finish = [0]

    # Set the left column
    for left in range(COL):

        # Initialize all elements of temp as 0
        temp = [0] * ROW

        # Set the right column for the left column set by outer loop
        for right in range(left, COL):

            # Calculate sum between current left and right for every row 'i'
            for i in range(ROW):
                temp[i] += M[i][right]

            # Find the maximum sum subarray in temp[]. The kadane() function also
            # sets values of start and finish so 'sum' is sum of rectangle between
            # (start, left) and (finish, right) which is the maximum sum with boundary columns
            # strictly as left and right.
            Sum = kadane(temp, start, finish, ROW)
```

```
            # Compare sum with maximum sum so far. If sum is more, then update maxSum and other
output values
            if Sum > maxSum:
                maxSum = Sum
                finalLeft = left
                finalRight = right
                finalTop = start[0]
                finalBottom = finish[0]

    # Prfinal values
    print("(Top, Left)", "(", finalTop,
        finalLeft, ")")
    print("(Bottom, Right)", "(", finalBottom,
        finalRight, ")")
    print("Max sum is:", maxSum)

ROW = 4
COL = 5
M = [[1, 2, -1, -4, -20],
    [-8, -3, 4, 2, 1],
    [3, 8, 10, 1, 3],
    [-4, -1, 1, 7, -6]]
findMaxSum(M)
```

Maximum profit by buying and selling a share at most k times

```
"""
Input:
Price = [10, 22, 5, 75, 65, 80]
    K = 2
Output:  87
Trader earns 87 as sum of 12 and 75
Buy at price 10, sell at 22, buy at
5 and sell at 80

Input:
Price = [12, 14, 17, 10, 14, 13, 12, 15]
    K = 3
Output:  12
Trader earns 12 as the sum of 5, 4 and 3
Buy at price 12, sell at 17, buy at 10
and sell at 14 and buy at 12 and sell
at 15

Input:
Price = [100, 30, 15, 10, 8, 25, 80]
    K = 3
Output:  72
Only one transaction. Buy at price 8
and sell at 80.

Input:
Price = [90, 80, 70, 60, 50]
    K = 1
Output:  0
Not possible to earn.
"""
def maxProfit(prices, n, k):
    profit = [[0 for _ in range(k + 1)] for _ in range(n)]
```

```python
    # Profit is zero for the first day and for zero transactions
    for i in range(1, n):
        for j in range(1, k + 1):
            max_so_far = 0
            for l in range(i):
                max_so_far = max(max_so_far, prices[i] -
                                 prices[l] + profit[l][j - 1])
            profit[i][j] = max(profit[i - 1][j], max_so_far)
    return profit[n - 1][k]

k = 2
prices = [10, 22, 5, 75, 65, 80]
n = len(prices)
print("Maximum profit is:",
    maxProfit(prices, n, k))
```

Find if a string is interleaved of two other strings

```python
"""
Given three strings A, B and C. Write a function that checks whether C is an interleaving of A and
B. C is said to be interleaving A and B, if it contains all and only characters of A and B and
order of all characters in individual strings is preserved.

Example:

Input: strings: "XXXXZY", "XXY", "XXZ"
Output: XXXXZY is interleaved of XXY and XXZ
The string XXXXZY can be made by
interleaving XXY and XXZ
String:    XXXXZY
String 1:    XX Y
String 2:  XX  Z

Input: strings: "XXY", "YX", "X"
Output: XXY is not interleaved of YX and X
XXY cannot be formed by interleaving YX and X.
The strings that can be formed are YXX and XYX
"""


dp = [[0]*101]*101
def dfs(i, j, A, B, C):

    # If path has already been calculated from this index then return calculated value.
    if(dp[i][j]!=-1):
        return dp[i][j]

    # If we reach the destination return 1
    n,m=len(A),len(B)
    if(i==n and j==m):
        return 1

    # If C[i+j] matches with both A[i] and B[j] we explore both the paths
    if (i<n and A[i]==C[i + j] and j<m and B[j]==C[i + j]):
        # go down and store the calculated value in x
        # and go right and store the calculated value in y.
        x = dfs(i + 1, j, A, B, C)
        y = dfs(i, j + 1, A, B, C)

        # return the best of both.
```

```
        dp[i][j] = x|y
        return dp[i][j]

    # If C[i+j] matches with A[i].
    if (i < n and A[i] == C[i + j]):
        # go down
        x = dfs(i + 1, j, A, B, C)

        # Return the calculated value.
        dp[i][j] = x
        return dp[i][j]

    # If C[i+j] matches with B[j].
    if (j < m and B[j] == C[i + j]):
        y = dfs(i, j + 1, A, B, C)

        # Return the calculated value.
        dp[i][j] = y
        return dp[i][j]

    # if nothing matches we return 0
    dp[i][j] = 0
    return dp[i][j]

# The main function that returns true if C is
# an interleaving of A and B, otherwise false.
def isInterleaved(A, B, C):

    # Storing the length in n,m
    n = len(A)
    m = len(B)

    # C can be an interleaving of A and B only of the sum
    # of lengths of A & B is equal to the length of C.
    if((n+m)!=len(C)):
        return 0
    # initializing dp array with -1
    for i in range(n+1):
        for j in range(m+1):
            dp[i][j]=-1
    # calling and returning the answer
    return dfs(0,0,A,B,C)

def test(A, B, C):
    if (isInterleaved(A, B, C)):
        print(C, "is interleaved of", A, "and", B)
    else:
        print(C, "is not interleaved of", A, "and", B)

test("XXY", "XXZ", "XXZXXXY")
test("XY", "WZ", "WZXY")
test("XY", "X", "XXY")
test("YX", "X", "XXY")
test("XXY", "XXZ", "XXXXZY")
test("ACA", "DAS", "DAACSA")
```

Graph

Implement Graph

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)


def printGraph(graph):
    for src in range(len(graph.adjList)):
        for dest in graph.adjList[src]:
            print(f'({src} -> {dest}) ', end='')
        print()

edges = [(0, 1), (1, 2), (2, 0), (2, 1), (3, 2), (4, 5), (5, 4)]
n = 6
graph = Graph(edges, n)
printGraph(graph)
```

Implement Weighted Graph

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [None] * n
        for i in range(n):
            self.adjList[i] = []
        for (src, dest, weight) in edges:
            self.adjList[src].append((dest, weight))


def printGraph(graph):
    for src in range(len(graph.adjList)):
        for (dest, weight) in graph.adjList[src]:
            print(f'({src} -> {dest}, {weight}) ', end='')
        print()


# Input: Edges in a weighted digraph (as per the above diagram)
# Edge (x, y, w) represents an edge from `x` to `y` having weight `w`
edges = [(0, 1, 6), (1, 2, 7), (2, 0, 5), (2, 1, 4), (3, 2, 10),
        (4, 5, 1), (5, 4, 3)]
n = 6
graph = Graph(edges, n)
printGraph(graph)
```

[Implement BFS algorithm]

```
from collections import deque

class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)
```

```
def BFS(graph, v, discovered):
    q = deque()
    discovered[v] = True
    q.append(v)
    while q:
        v = q.popleft()
        print(v, end=' ')
        for u in graph.adjList[v]:
            if not discovered[u]:
                discovered[u] = True
                q.append(u)

edges = [
    (1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (5, 9),
    (5, 10), (4, 7), (4, 8), (7, 11), (7, 12)
    # vertex 0, 13, and 14 are single nodes
]
n = 15
graph = Graph(edges, n)
discovered = [False] * n
for i in range(n):
    if not discovered[i]:
        BFS(graph, i, discovered)
```

Implement DFS Algo

```
from collections import deque
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)

def iterativeDFS(graph, v, discovered):
    stack = deque()
    stack.append(v)
    while stack:
        v = stack.pop()
        if discovered[v]:
            continue
        discovered[v] = True
        print(v, end=' ')
        adjList = graph.adjList[v]
        for i in reversed(range(len(adjList))):
            u = adjList[i]
            if not discovered[u]:
                stack.append(u)



edges = [
    # Notice that node 0 is unconnected
    (1, 2), (1, 7), (1, 8), (2, 3), (2, 6), (3, 4),
    (3, 5), (8, 9), (8, 12), (9, 10), (9, 11)
    # (6, 9) introduces a cycle
]
n = 13
```

```
graph = Graph(edges, n)
discovered = [False] * n
for i in range(n):
    if not discovered[i]:
        iterativeDFS(graph, i, discovered)
```

Detect Cycle in Directed Graph using BFS/DFS Algo

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)


# Perform DFS on the graph and set the departure time of all vertices of the graph
def DFS(graph, v, discovered, departure, time):

    # mark the current node as discovered
    discovered[v] = True

    # do for every edge (v, u)
    for u in graph.adjList[v]:
        # if `u` is not yet discovered
        if not discovered[u]:
            time = DFS(graph, u, discovered, departure, time)

    # ready to backtrack set departure time of vertex `v`
    departure[v] = time
    time = time + 1

    return time


# Returns true if the given directed graph is DAG
def isDAG(graph, n):

    # keep track of whether a vertex is discovered or not
    discovered = [False] * n

    # keep track of the departure time of a vertex in DFS
    departure = [None] * n

    time = 0

    # Perform DFS traversal from all undiscovered vertices to visit all connected components of a
graph
    for i in range(n):
        if not discovered[i]:
            time = DFS(graph, i, discovered, departure, time)

    # check if the given directed graph is DAG or not
    for u in range(n):

        # check if (u, v) forms a back-edge.
        for v in graph.adjList[u]:

            # If the departure time of vertex `v` is greater than equal
            # to the departure time of `u`, they form a back edge.
```

```
            # Note that `departure[u]` will be equal to `departure[v]`
            # only if `u = v`, i.e., vertex contain an edge to itself
            if departure[u] <= departure[v]:
                return False

    # no back edges
    return True



# List of graph edges as per the above diagram
edges = [(0, 1), (0, 3), (1, 2), (1, 3), (3, 2), (3, 4), (3, 0), (5, 6), (6, 3)]
# total number of nodes in the graph (labelled from 0 to 6)
n = 7
graph = Graph(edges, n)
if isDAG(graph, n):
    print('Does not contain a Cycle')
else:
    print('Contains a Cycle')
```

Detect Cycle in UnDirected Graph using BFS/DFS Algo

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)


# Function to perform DFS traversal on the graph on a graph
def DFS(graph, v, discovered, parent=-1):

    # mark the current node as discovered
    discovered[v] = True

    # do for every edge (v, w)
    for w in graph.adjList[v]:

        # if `w` is not discovered
        if not discovered[w]:
            if DFS(graph, w, discovered, v):
                return True

        # if `w` is discovered, and `w` is not a parent
        elif w != parent:
            # we found a back-edge (cycle)
            return True

    # No back-edges were found in the graph
    return False



edges = [
    (0, 1), (0, 6), (0, 7), (1, 2), (1, 5), (2, 3),
    (2, 4), (7, 8), (7, 11), (8, 9), (8, 10), (10, 11)
```

```
    # edge (10, 11) introduces a cycle in the graph
]

# total number of nodes in the graph (0 to 11)
n = 12

graph = Graph(edges, n)
discovered = [False] * n
if DFS(graph, 0, discovered):
    print('The graph contains a cycle')
else:
    print('The graph doesn\'t contain any cycle')
```

Minimum Step by Knight

```
"""Given a chessboard, find the shortest distance (minimum number of steps) taken by a knight to
reach a given destination from a given source.

For example,

Input:

N = 8 (8 × 8 board)
Source = (7, 0)
Destination = (0, 7)

Output: Minimum number of steps required is 6
"""

import sys
from collections import deque


class Node:
    # (x, y) represents chessboard coordinates `dist` represents its minimum distance from the
source
    def __init__(self, x, y, dist=0):
        self.x = x
        self.y = y
        self.dist = dist

    # As we are using `Node` as a key in a dictionary, we need to override the `__hash__()` and
`__eq__()` function
    def __hash__(self):
        return hash((self.x, self.y, self.dist))

    def __eq__(self, other):
        return (self.x, self.y, self.dist) == (other.x, other.y, other.dist)


# Below lists detail all eight possible movements for a knight
row = [2, 2, -2, -2, 1, 1, -1, -1]
col = [-1, 1, 1, -1, 2, -2, 2, -2]


# Check if (x, y) is valid chessboard coordinates.
# Note that a knight cannot go out of the chessboard
def isValid(x, y, N):
    return x >= 0 and y >= 0 and x < N and y < N
```

```python
# Find the minimum number of steps taken by the knight
# from the source to reach the destination using BFS
def findShortestDistance(src, dest, N):

    # set to check if the matrix cell is visited before or not
    visited = set()

    # create a queue and enqueue the first node
    q = deque()
    q.append(src)

    # loop till queue is empty
    while q:

        # dequeue front node and process it
        node = q.popleft()

        x = node.x
        y = node.y
        dist = node.dist

        # if the destination is reached, return distance
        if x == dest.x and y == dest.y:
            return dist

        # skip if the location is visited before
        if node not in visited:
            # mark the current node as visited
            visited.add(node)

            # check for all eight possible movements for a knight
            # and enqueue each valid movement
            for i in range(len(row)):
                # get the knight's valid position from the current position on
                # the chessboard and enqueue it with +1 distance
                x1 = x + row[i]
                y1 = y + col[i]

                if isValid(x1, y1, N):
                    q.append(Node(x1, y1, dist + 1))

    # return infinity if the path is not possible
    return sys.maxsize


N = 8                # N x N matrix
src = Node(0, 7)     # source coordinates
dest = Node(7, 0)    # destination coordinates
print("The minimum number of steps required is",findShortestDistance(src, dest, N))
```

flood fill algo

```python
"""Flood fill (also known as seed fill) is an algorithm that determines the area connected to a
given node in a multi-dimensional array.
"""
```

```python
# Below lists detail all eight possible movements
row = [-1, -1, -1, 0, 0, 1, 1, 1]
col = [-1, 0, 1, -1, 1, -1, 0, 1]


# check if it is possible to go to pixel (x, y) from the
# current pixel. The function returns false if the pixel
# has a different color, or it's not a valid pixel
def isSafe(mat, x, y, target):
    return 0 <= x < len(mat) and 0 <= y < len(mat[0]) and mat[x][y] == target


# Flood fill using DFS
def floodfill(mat, x, y, replacement):

    # base case
    if not mat or not len(mat):
        return

    # get the target color
    target = mat[x][y]

    # target color is same as replacement
    if target == replacement:
        return

    # replace the current pixel color with that of replacement
    mat[x][y] = replacement

    # process all eight adjacent pixels of the current pixel and
    # recur for each valid pixel
    for k in range(len(row)):

        # if the adjacent pixel at position (x + row[k], y + col[k]) is
        # a valid pixel and has the same color as that of the current pixel
        if isSafe(mat, x + row[k], y + col[k], target):
            floodfill(mat, x + row[k], y + col[k], replacement)



mat = [
        ['Y', 'Y', 'Y', 'G', 'G', 'G', 'G', 'G', 'G', 'G'],
        ['Y', 'Y', 'Y', 'Y', 'Y', 'Y', 'G', 'X', 'X', 'X'],
        ['G', 'G', 'G', 'G', 'G', 'G', 'G', 'X', 'X', 'X'],
        ['W', 'W', 'W', 'W', 'W', 'G', 'G', 'G', 'G', 'X'],
        ['W', 'R', 'R', 'R', 'R', 'R', 'G', 'X', 'X', 'X'],
        ['W', 'W', 'W', 'R', 'R', 'G', 'G', 'X', 'X', 'X'],
        ['W', 'B', 'W', 'R', 'R', 'R', 'R', 'R', 'R', 'X'],
        ['W', 'B', 'B', 'B', 'B', 'R', 'R', 'X', 'X', 'X'],
        ['W', 'B', 'B', 'X', 'B', 'B', 'B', 'B', 'X', 'X'],
        ['W', 'B', 'B', 'X', 'X', 'X', 'X', 'X', 'X', 'X']
    ]

# start node
x, y = (3, 9)   # having a target color `X`

# replacement color
replacement = 'C'

# replace the target color with a replacement color using DFS
```

```
floodfill(mat, x, y, replacement)

# print the colors after replacement
for r in mat:
    print(r)
```

Clone a graph

```
TODO
```

Making wired Connections

```
"""There are n computers numbered from 0 to n-1 connected by ethernet cables connections forming a
network where connections[i] = [a, b] represents a connection between computers a and b. Any
computer can reach any other computer directly or indirectly through the network.

Given an initial computer network connections. You can extract certain cables between two directly
connected computers, and place them between any pair of disconnected computers to make them
directly connected. Return the minimum number of times you need to do this in order to make all
the computers connected. If it's not possible, return -1.

Example 1:

Input: n = 4, connections = [[0,1],[0,2],[1,2]]
Output: 1
Explanation: Remove cable between computer 1 and 2 and place between computers 1 and 3.
Example 2:

Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2],[1,3]]
Output: 2
Example 3:

Input: n = 6, connections = [[0,1],[0,2],[0,3],[1,2]]
Output: -1
Explanation: There are not enough cables.
Example 4:

Input: n = 5, connections = [[0,1],[0,2],[3,4],[2,3]]
Output: 0
"""

def makeConnected(n, connections):
    uf = {i: i for i in range(n)}

    def find(x):
        uf.setdefault(x, x)
        if uf[x] != x:
            uf[x] = find(uf[x])
        return uf[x]

    def union(a, b):
        uf[find(a)] = find(b)
    if len(connections) < n - 1:
        return -1
    for a, b in connections:
        union(a, b)
    islands = len({find(x) for x in uf})
    return islands - 1
```

```
n = 4

connections = [[0,1],[0,2],[1,2]]
print(makeConnected(n, connections))
```

word Ladder

```
"""
Given two words (beginWord and endWord), and a dictionary's word list,
find the length of shortest transformation sequence from beginWord to endWord, such that:
(i) Only one letter can be changed at a time, and (ii) each transformed word must exist in
the word list. Note that beginWord is not a transformed word.

EXAMPLES
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log","cog"]
-> 5 (because "hit" -> "hot" -> "dot" -> "dog" -> "cog")

"""

from collections import deque
def ladderLength(beginWord, endWord, wordList):
    """
    :type beginWord: str
    :type endWord: str
    :type wordList: Set[str]
    :rtype: int
    """

    queue = deque()
    queue.append((beginWord, [beginWord]))
    while queue:
        node, path = queue.popleft()
        for next in next_nodes(node, wordList) - set(path):
            if next == endWord:
                return len(path) + 1
            else:
                queue.append((next, path + [next]))
    return 0

def next_nodes(word, word_list):
    to_return = set()
    for w in word_list:
        mismatch_count, w_length = 0, len(w)
        for i in range(w_length):
            if w[i] != word[i]:
                mismatch_count += 1
        if mismatch_count == 1:
            to_return.add(w)
    return to_return

beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log","cog"]
print(ladderLength(beginWord, endWord, wordList))
```

Dijkstra algo

```python
"""
Given a source vertex s from a set of vertices V in a weighted digraph where all its edge weights
w(u, v) are non-negative, find the shortest path weights d(s, v) from source s for all vertices v
present in the graph
"""
import sys
from heapq import heappop, heappush

class Node:
    def __init__(self, vertex, weight=0):
        self.vertex = vertex
        self.weight = weight

    # Override the __lt__() function to make `Node` class work with a min-heap
    def __lt__(self, other):
        return self.weight < other.weight

class Graph:
    def __init__(self, edges, n):
        # allocate memory for the adjacency list
        self.adjList = [[] for _ in range(n)]

        # add edges to the directed graph
        for (source, dest, weight) in edges:
            self.adjList[source].append((dest, weight))

def get_route(prev, i, route):
    if i >= 0:
        get_route(prev, prev[i], route)
        route.append(i)


def findShortestPaths(graph, source, n):

    # create a min-heap and push source node having distance 0
    pq = []
    heappush(pq, Node(source))

    # set initial distance from the source to `v` as infinity
    dist = [sys.maxsize] * n

    # distance from the source to itself is zero
    dist[source] = 0

    # list to track vertices for which minimum cost is already found
    done = [False] * n
    done[source] = True

    # stores predecessor of a vertex (to a print path)
    prev = [-1] * n

    # run till min-heap is empty
    while pq:

        node = heappop(pq)      # Remove and return the best vertex
        u = node.vertex         # get the vertex number

        # do for each neighbor `v` of `u`
        for (v, weight) in graph.adjList[u]:
```

```
            if not done[v] and (dist[u] + weight) < dist[v]:        # Relaxation step
                dist[v] = dist[u] + weight
                prev[v] = u
                heappush(pq, Node(v, dist[v]))

        # mark vertex `u` as done so it will not get picked up again
        done[u] = True

    route = []
    for i in range(n):
        if i != source and dist[i] != sys.maxsize:
            get_route(prev, i, route)
            print(f'Path ({source} —> {i}): Minimum cost = {dist[i]}, Route = {route}')
            route.clear()

# initialize edges as per the above diagram (u, v, w) represent edge from vertex `u` to vertex `v`
having weight `w`
edges = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7),
        (4, 1, 1), (4, 2, 8), (4, 3, 2)]
# total number of nodes in the graph (labelled from 0 to 4)
n = 5
graph = Graph(edges, n)
for source in range(n):
    findShortestPaths(graph, source, n)
```

Implement Topological Sort

```
"""
Given a Directed Acyclic Graph (DAG), print it in topological order using topological sort
algorithm. If the graph has more than one topological ordering, output any of them. Assume valid
Directed Acyclic Graph (DAG).

A Topological sort or Topological ordering of a directed graph is a linear ordering of its
vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the
ordering. A topological ordering is possible if and only if the graph has no directed cycles, i.e.
if the graph is DAG.

"""
# A class to represent a graph object
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)


# Perform DFS on the graph and set the departure time of all vertices of the graph
def DFS(graph, v, discovered, departure, time):
    discovered[v] = True
    time = time + 1
    for u in graph.adjList[v]:
        if not discovered[u]:
            time = DFS(graph, u, discovered, departure, time)
    departure[time] = v
    time = time + 1
    return time


# Function to perform a topological sort on a given DAG
```

```
def doTopologicalSort(graph, n):

    # departure[] stores the vertex number using departure time as an index
    departure = [-1] * 2 * n

    ''' If we had done it the other way around, i.e., fill the array
        with departure time using vertex number as an index, we would
        need to sort it later '''

    # to keep track of whether a vertex is discovered or not
    discovered = [False] * n
    time = 0

    # perform DFS on all undiscovered vertices
    for i in range(n):
        if not discovered[i]:
            time = DFS(graph, i, discovered, departure, time)

    # Print the vertices in order of their decreasing
    # departure time in DFS, i.e., in topological order
    for i in reversed(range(2*n)):
        if departure[i] != -1:
            print(departure[i], end=' ')


# List of graph edges as per the above diagram
edges = [(0, 6), (1, 2), (1, 4), (1, 6), (3, 0), (3, 4), (5, 1), (7, 0), (7, 1)]
# total number of nodes in the graph (labelled from 0 to 7)
n = 8
graph = Graph(edges, n)
doTopologicalSort(graph, n)
```

Minimum time taken by each job to be completed given by a Directed Acyclic Graph

```
"""
Given a Directed Acyclic Graph having V vertices and E edges, where each edge {U, V} represents
the Jobs U and V such that Job V can only be started only after completion of Job U. The task is
to determine the minimum time taken by each job to be completed where each Job takes unit time to
get completed.
"""

from collections import defaultdict

class Graph:
    def __init__(self, vertices, edges):
        self.graph = defaultdict(list)
        self.n = vertices
        self.m = edges

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to find the minimum time needed by each node to get the task
    def printOrder(self, n, m):

        # Create a vector to store indegrees of all vertices. Initialize all indegrees as 0.
```

```
        indegree = [0] * (self.n + 1)

        # Traverse adjacency lists to fill indegrees of vertices. This step takes O(V + E) time
        for i in self.graph:
            for j in self.graph[i]:
                indegree[j] += 1

        # Array to store the time in which the job i can be done
        job = [0] * (self.n + 1)

        # Create an queue and enqueue all vertices with indegree 0
        q = []

        # Update the time of the jobs who don't require any job to be completed before this job
        for i in range(1, self.n + 1):
            if indegree[i] == 0:
                q.append(i)
                job[i] = 1

        # Iterate until queue is empty
        while q:

            # Get front element of queue
            cur = q.pop(0)

            for adj in self.graph[cur]:

                # Decrease in-degree of the current node
                indegree[adj] -= 1

                # Push its adjacent elements
                if (indegree[adj] == 0):
                    job[adj] = 1 + job[cur]
                    q.append(adj)

        # Print the time to complete the job
        for i in range(1, n + 1):
            print(job[i], end = " ")

        print()

# Given Nodes N and edges M
n = 10
m = 13
g = Graph(n, m)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(1, 5)
g.addEdge(2, 3)
g.addEdge(2, 8)
g.addEdge(2, 9)
g.addEdge(3, 6)
g.addEdge(4, 6)
g.addEdge(4, 8)
g.addEdge(5, 8)
g.addEdge(6, 7)
g.addEdge(7, 8)
g.addEdge(8, 10)
g.printOrder(n, m)
```

Find whether it is possible to finish all tasks or not from given dependencies

```
"""
There are a total of n tasks you have to pick, labelled from 0 to n-1. Some tasks may have
prerequisites, for example to pick task 0 you have to first pick task 1, which is expressed as a
pair: [0, 1]
Given the total number of tasks and a list of prerequisite pairs, is it possible for you to finish
all tasks?
Examples:



Input: 2, [[1, 0]]
Output: true
Explanation: There are a total of 2 tasks to pick. To pick task 1 you should have finished task 0.
So it is possible.
Input: 2, [[1, 0], [0, 1]]
Output: false
Explanation: There are a total of 2 tasks to pick. To pick task 1 you should have finished task 0,
and to pick task 0 you should also have finished task 1. So it is impossible.
Input: 3, [[1, 0], [2, 1], [3, 2]]
Output: true
Explanation: There are a total of 3 tasks to pick. To pick tasks 1 you should have finished task
0, and to pick task 2 you should have finished task 1 and to pick task 3 you should have finished
task 2. So it is possible.
"""



class Solution:

    arr = []
    # parameterized constructor
    def __init__(self,n):
        # Initially, everyone is their own child
        self.arr = list(range(n))

    def makeParent(self,a, b):
        # find parent of b and make it a's parent
        self.arr[a] = self.findParent(b)

    def findParent(self,c):
        # when an independent task is found
        return c if (c == self.arr) else self.findParent(self.arr)

    def isPossible(self,N , prerequisites):
        # traverse through pre-requisites array
        for i in range(len(prerequisites)):
            # check whether given pre-requisite pair already have a common pre-requisite(parent)
            if (self.findParent(prerequisites[i][0]) == self.findParent(prerequisites[i][1])):
            # tasks cannot be completed because there was a cyclic condition in the tasks
                return False
            # make parent-child relation between pre-requisite task and the task dependent on it
            self.makeParent(prerequisites[i][0], prerequisites[i][1])
        # if there was no cycle found, tasks can be completed
        return True


prerequisites = [[1, 0], [2, 1], [3, 2]]
```

```
ob = Solution(4)
if ob.isPossible(4,prerequisites ):
    print("Yes")
else:
    print("No")
```

Find the no. of Islands

```
"""
Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island.
For example, the below matrix contains 5 islands

Example:

Input : mat[][] = {{1, 1, 0, 0, 0},
                   {0, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1},
                   {0, 0, 0, 0, 0},
                   {1, 0, 1, 0, 1}}
Output : 5
"""



# Program to count islands in boolean 2D matrix
class Graph:
    def __init__(self, row, col, graph):
        self.ROW = row
        self.COL = col
        self.graph = graph

    # A utility function to do DFS for a 2D boolean matrix. It only considers the 8 neighbours as
adjacent vertices
    def DFS(self, i, j):
        if i < 0 or i >= len(self.graph) or j < 0 or j >= len(self.graph[0]) or self.graph[i][j]
!= 1:
            return

        # mark it as visited
        self.graph[i][j] = -1

        # Recur for 8 neighbours
        self.DFS(i - 1, j - 1)
        self.DFS(i - 1, j)
        self.DFS(i - 1, j + 1)
        self.DFS(i, j - 1)
        self.DFS(i, j + 1)
        self.DFS(i + 1, j - 1)
        self.DFS(i + 1, j)
        self.DFS(i + 1, j + 1)

    # The main function that returns count of islands in a given boolean 2D matrix
    def countIslands(self):
        # Initialize count as 0 and traverse through the all cells of given matrix
        count = 0
        for i in range(self.ROW):
            for j in range(self.COL):
                # If a cell with value 1 is not visited yet, then new island found
                if self.graph[i][j] == 1:
                    # Visit all cells in this island and increment island count
```

```
                    self.DFS(i, j)
                    count += 1

        return count


graph = [
    [1, 1, 0, 0, 0],
    [0, 1, 0, 0, 1],
    [1, 0, 0, 1, 1],
    [0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1]
]
row = len(graph)
col = len(graph[0])
g = Graph(row, col, graph)
print("Number of islands is:", g.countIslands())
```

Given a sorted Dictionary of an Alien Language, find order of characters

```
"""
Given a dictionary of ancient origin where the words are arranged alphabetically, find the correct order of alphabets in the ancient language.

For example,

Input:  Ancient dictionary { ¥€±, €±€, €±‰ð, ðß, ±±ð, ±ßß }
Output: The correct order of alphabets in the ancient language is {¥ € ‰ ð ± ß}.

Since the input is small, more than one ordering is possible. Another such ordering is {¥ € ð ± ß ‰}.


Input:  Ancient dictionary { ÿ€±š, €€€ß, €€‰ð, ðß, ±ß¥š }
Output: The correct order of alphabets in the ancient language is {ÿ € ‰ ð ±}.

The alphabets {š, ß, ¥} are not included in the order as they are not properly defined.
"""



class Graph:
    def __init__(self, N):
        self.adj = [[] for _ in range(N)]


def DFS(graph, v, discovered, departure, time):
    discovered[v] = True
    time = time + 1
    for u in graph.adj[v]:
        if not discovered[u]:
            time = DFS(graph, u, discovered, departure, time)
    departure[time] = v
    return time + 1


# Utility function to performs topological sort on a given DAG
```

```
def doTopologicalSort(graph, d):

    # `departure[]` stores the vertex number using departure time as an index
    departure = [-1] * (2 * N)

    ''' If we had done it the other way around, i.e., fill the array
        with departure time using vertex number as an index, we would
        need to sort it later '''

    # to keep track of whether a vertex is discovered or not
    discovered = [False] * N
    time = 0

    # perform DFS on all undiscovered connected vertices
    for i in range(N):
        if not discovered[i] and len(graph.adj[i]):
            time = DFS(graph, i, discovered, departure, time)

    print('\nThe correct order of alphabets in the ancient language is', end=' ')

    # Print the vertices in order of their decreasing
    # departure time in DFS, i.e., in topological order
    for i in reversed(range(2*N)):
        if departure[i] != -1:
            print(d[departure[i]], end=' ')


# Utility function to print adjacency list representation of a graph
def printGraph(graph, d):

    for i in range(N):
        # ignore vertices with no outgoing edges
        if graph.adj[i]:
            # print current vertex and all neighboring vertices of a vertex `i`
            print(d[i], '->', [d[v] for v in graph.adj[i]])


# Function to find the correct order of alphabets in a given dictionary of
# ancient origin. This function assumes that the input is correct.
def findAlphabetsOrder(dictionary):
    # create a dictionary to map each non-ASCII character present in the given dictionary with a
unique integer
    d = {}
    k = 0
    # do for each word
    for word in dictionary:
        # do for each non-ASCII character of the word
        for s in word:
            # if the current character is not present in the dictionary, insert it
            d.setdefault(s, k)
            k = k + 1
    # create a graph containing `N` nodes
    graph = Graph(N)

    # iterate through the complete dictionary and compare adjacent words for character mismatch
    for i in range(1, len(dictionary)):
        # previous word in the dictionary
        prev = dictionary[i - 1]

        # current word in the dictionary
```

```
        curr = dictionary[i]

        # iterate through both `prev` and `curr` simultaneously and find the first mismatching
character
        j = 0
        while j < len(prev) and j < len(curr):
            # mismatch found
            if prev[j] is not curr[j]:

                # add an edge from the current character of `prev` to the
                # current character of `curr` in the graph
                graph.adj[d[prev[j]]].append(d[curr[j]])
                break

            j += 1

    # create a reverse dict
    reverse = {v: k for k, v in d.items()}
    printGraph(graph, reverse)
    # perform a topological sort on the above graph
    doTopologicalSort(graph, reverse)


# define the maximum number of alphabets in the ancient dictionary
N = 100
dictionary = [
    ["¥", "€", "±"],
    ["€", "±", "€"],
    ["€", "±", "‰", "ð"],
    ["ð", "ß"],
    ["±", "±", "ð"],
    ["±", "ß", "ß"]
]
findAlphabetsOrder(dictionary)
```

Implement Kruksal'sAlgorithm

```
"""
Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If
cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.
"""

from collections import defaultdict
class Graph:

    def __init__(self, vertices):
        self.V = vertices # No. of vertices
        self.graph = [] # default dictionary
        # to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i (uses path compression technique)
```

```
    def find(self, parent, i):
        return i if parent[i] == i else self.find(parent, parent[i])

    # A function that does union of two sets of x and y (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's algorithm
    def KruskalMST(self):

        result = [] # This will store the resultant MST
        i = 0 # An index variable, used for sorted edges
        e = 0 # An index variable, used for result[]

        # Step 1: Sort all the edges in non-decreasing order of their
        # weight. If we are not allowed to change the given graph, we can create a copy of graph
        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge doesn't cause cycle, include it in result and increment the
indexof
            # result for next edge
            if x != y:
                e += 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
                # Else discard the edge

        minimumCost = 0
        print ("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
```

```
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree" , minimumCost)

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)
g.KruskalMST()
```

Implement Prim’s Algorithm

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph

import sys # Library for INT_MAX

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                    for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print ("Edge \tWeight")
        for i in range(1, self.V):
            print (parent[i], "-", i, "\t", self.graph[i][parent[i]])

    # A utility function to find the vertex with minimum distance value, from the set of vertices
not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize minValue value
        minValue = sys.maxsize

        for v in range(self.V):
            if key[v] < minValue and mstSet[v] == False:
                minValue = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph represented using adjacency matrix
representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1 # First node is always the root of

        for cout in range(self.V):
```

```
            # Pick the minimum distance vertex from the set of vertices not yet processed. u is
always equal to src in first iteration
            u = self.minKey(key, mstSet)

            # Put the minimum distance vertex in the shortest path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices of the picked vertex only if the current
            # distance is greater than new distance and the vertex in not in the shortest path
tree
            for v in range(self.V):

                # graph[u][v] is non zero only for adjacent vertices of m mstSet[v] is false for
vertices not yet included in MST
                # Update the key only if graph[u][v] is smaller than key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                        key[v] = self.graph[u][v]
                        parent[v] = u
        self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],
            [2, 0, 3, 8, 5],
            [0, 3, 0, 0, 7],
            [6, 8, 0, 0, 9],
            [0, 5, 7, 9, 0]]
g.primMST();
```

Total no. of Spanning tree in a graph

```
TODO
```

Implement Bellman Ford Algorithm

```
"""
We are given a directed graph. We need to compute whether the graph has a negative cycle or not. A
negative cycle is one in which the overall sum of the cycle becomes negative.
"""

# a structure to represent a weighted edge in graph
class Edge:
    def __init__(self):
        self.src = 0
        self.dest = 0
        self.weight = 0

# a structure to represent a connected, directed and weighted graph
class Graph:
    def __init__(self):
        # V. Number of vertices, E. Number of edges
        self.V = 0
        self.E = 0

        # graph is represented as an array of edges.
        self.edge = None

# Creates a graph with V vertices and E edges
```

```python
def createGraph(V, E):

    graph = Graph()
    graph.V = V;
    graph.E = E;
    graph.edge = [Edge() for _ in range(graph.E)]
    return graph;

# The main function that finds shortest distances from src to all other vertices using Bellman-
Ford algorithm. The function also detects negative weight cycle
def isNegCycleBellmanFord(graph, src):

    V = graph.V;
    E = graph.E;
    dist = [1000000 for _ in range(V)];
    dist[src] = 0;

    # Step 2: Relax all edges |V| - 1 times.
    # A simple shortest path from src to any other vertex can have at-most |V| - 1 edges
    for _ in range(1, V):
        for j in range(E):

            u = graph.edge[j].src;
            v = graph.edge[j].dest;
            weight = graph.edge[j].weight;
            if (dist[u] != 1000000 and dist[u] + weight < dist[v]):
                dist[v] = dist[u] + weight;

    # Step 3: check for negative-weight cycles.
    # The above step guarantees shortest distances if graph doesn't contain negative weight cycle.
    # If we get a shorter path, then there is a cycle.
    for i in range(E):

        u = graph.edge[i].src;
        v = graph.edge[i].dest;
        weight = graph.edge[i].weight;
        if (dist[u] != 1000000 and dist[u] + weight < dist[v]):
            return True;

    return False;


# Let us create the graph given in above example
V = 5; # Number of vertices in graph
E = 8; # Number of edges in graph
graph = createGraph(V, E)

source= [0,0,1,1,1,3,3,4]
destination= [1,2,2,3,4,2,1,3]
weight=[-1,4,3,2,2,5,1,-3]

for i in range(E):
    graph.edge[i].src=source[i]
    graph.edge[i].dest=destination[i]
    graph.edge[i].weight=weight[i]

if (isNegCycleBellmanFord(graph, 0)):
    print("Yes")
else:
    print("No")
```

Implement Floyd warshallAlgorithm

```
"""
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to
find shortest distances between every pair of vertices in a given edge weighted directed Graph.
Example:

Input:
       graph[][] = { {0,   5,  INF, 10},
                    {INF,  0,  3,  INF},
                    {INF, INF, 0,   1},
                    {INF, INF, INF, 0} }
which represents the following graph
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
"""



def floydWarshall(graph):
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    for k in range(V):
        # pick all vertices as source one by one
        for i in range(V):
            # Pick all vertices as destination for the above picked source
            for j in range(V):

                # If vertex k is on the shortest path from i to j, then update the value of
dist[i][j]
                dist[i][j] = min(dist[i][j],dist[i][k] + dist[k][j])
    printSolution(dist)



# A utility function to print the solution
def printSolution(dist):
    print ("Following matrix shows the shortest distances\
between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print ("%7s" % ("INF"),end=" ")
            else:
                print ("%7d\t" % (dist[i][j]),end=' ')
```

```
            if j == V-1:
                print ()



# Let us create the following weighted graph
"""
            10
    (0)------->(3)
        |         /|\
    5 |       |
        |         | 1
    \|/       |
    (1)------->(2)
            3
"""
# Number of vertices in the graph
V = 4
INF = 99999

graph = [[0, 5, INF, 10],
        [INF, 0, 3, INF],
        [INF, INF, 0, 1],
        [INF, INF, INF, 0]
        ]

floydWarshall(graph)
```

Travelling Salesman Problem

```
"""
Travelling Salesman Problem (TSP):

Given a set of cities and the distance between every pair of cities, the problem is to find the
shortest possible route that visits every city exactly once and returns to the starting point.
"""
n = 4 # there are four nodes in example graph (graph is 1-based)

# dist[i][j] represents shortest distance to go from i to  this matrix can be calculated for any
given graph usin all-pair shortest path algorithms
dist = [[0, 0, 0, 0, 0], [0, 0, 10, 15, 20], [
    0, 10, 0, 25, 25], [0, 15, 25, 0, 30], [0, 20, 25, 30, 0]]

# memoization for top down recursion
memo = [[-1]*(1 << (n+1)) for _ in range(n+1)]


def fun(i, mask):
    # base case
    # if only ith bit and 1st bit is set in our mask, it implies we have visited all other nodes
already
    if mask == ((1 << i) | 3):
        return dist[1][i]

    # memoization
    if memo[i][mask] != -1:
        return memo[i][mask]
```

```
    res = 10**9 # result of this sub-problem

    # we have to travel all nodes j in mask and end the path at ith node so for every node j in
mask, recursively calculate cost of travelling all nodes in mask except i and then travel back
from node j to node i taking
    # the shortest path take the minimum of all possible j nodes
    for j in range(1, n+1):
        if (mask & (1 << j)) != 0 and j != i and j != 1:
            res = min(res, fun(j, mask & (~(1 << i))) + dist[j][i])
    memo[i][mask] = res # storing the minimum value
    return res




ans = 10**9
for i in range(1, n+1):
    # try to go from node 1 visiting all nodes in between to i then return from i taking the
shortest route to 1
    ans = min(ans, fun(i, (1 << (n+1))-1) + dist[i][1])
print(f"The cost of most efficient tour = {str(ans)}")
```

Graph ColouringProblem

```
TODO
```

Snake and Ladders Problem

```
"""
On an N x N board, the numbers from 1 to N*N are written boustrophedonically starting from the
bottom left of the board, and alternating direction each row. For example, for a 6 x 6 board, the
numbers are written as follows:

You start on square 1 of the board (which is always in the last row and first column). Each move,
starting from square x, consists of the following:

You choose a destination square S with number x+1, x+2, x+3, x+4, x+5, or x+6, provided this
number is <= N*N.
(This choice simulates the result of a standard 6-sided die roll: ie., there are always at most 6
destinations, regardless of the size of the board.)
If S has a snake or ladder, you move to the destination of that snake or ladder. Otherwise, you
move to S. A board square on row r and column c has a “snake or ladder” if board[r][c] != -1. The
destination of that snake or ladder is board[r][c].
Note that you only take a snake or ladder at most once per move: if the destination to a snake or
ladder is the start of another snake or ladder, you do not continue moving. (For example, if the
board is [[4,-1],[-1,3]], and on the first move your destination square is 2, then you finish your
first move at 3, because you do not continue moving to 4.)

Return the least number of moves required to reach square N*N. If it is not possible, return -1.
"""
import collections
def snakesAndLadders(board):
        rows = len(board)
        total_square = rows*rows

        def next_square(step):
                quot, rem = divmod(step-1, rows)
                row = (rows - 1) - quot
                col = rem if row%2 != rows%2 else (rows - 1) - rem
```

```
            return row, col

        dist = {1: 0}#square and step
        queue = collections.deque([1])
        while queue:
            square = queue.popleft()
            if square == total_square:
                return dist[square]
            for new_square in range(square+1, min(square+6, total_square) + 1):
                r, c = next_square(new_square)
                if board[r][c] != -1:
                    new_square = board[r][c]
                if new_square not in dist:
                    dist[new_square] = dist[square] + 1
                    queue.append(new_square)
board=[
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,35,-1,-1,13,-1],
[-1,-1,-1,-1,-1,-1],
[-1,15,-1,-1,-1,-1]]
print(snakesAndLadders(board))
```

Find bridge in a graph

```
from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    '''
    A recursive function that finds and prints bridges
    using DFS traversal
    u --> The vertex to be visited next
    visited[] --> keeps track of visited vertices
    disc[] --> Stores discovery times of visited vertices
    parent[] --> Stores parent vertices in DFS tree
    '''
    def bridgeUtil(self, u, visited, parent, low, disc):

        # Mark the current node as visited and print it
        visited[u]= True

        # Initialize discovery time and low value
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1
```

```
        #Recur for all the vertices adjacent to this vertex
        for v in self.graph[u]:
            # If v is not visited yet, then make it a child of u in DFS tree and recur for it
            if visited[v] == False :
                parent[v] = u
                self.bridgeUtil(v, visited, parent, low, disc)

                # Check if the subtree rooted with v has a connection to one of the ancestors of u
                low[u] = min(low[u], low[v])


                ''' If the lowest vertex reachable from subtree
                under v is below u in DFS tree, then u-v is
                a bridge'''
                if low[v] > disc[u]:
                    print ("%d %d" %(u,v))


            elif v != parent[u]: # Update low value of u for parent function calls.
                low[u] = min(low[u], disc[v])


    # DFS based function to find all bridges. It uses recursive function bridgeUtil()
    def bridge(self):

        # Mark all the vertices as not visited and Initialize parent and visited, and
ap(articulation point) arrays
        visited = [False] * (self.V)
        disc = [float("Inf")] * (self.V)
        low = [float("Inf")] * (self.V)
        parent = [-1] * (self.V)

        # Call the recursive helper function to find bridges in DFS tree rooted with vertex 'i'
        for i in range(self.V):
            if visited[i] == False:
                self.bridgeUtil(i, visited, parent, low, disc)



g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)


print ("Bridges in first graph ")
g1.bridge()

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print ("\nBridges in second graph ")
g2.bridge()


g3 = Graph (7)
g3.addEdge(0, 1)
```

```
g3.addEdge(1, 2)
g3.addEdge(2, 0)
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print ("\nBridges in third graph ")
g3.bridge()
```

Count Strongly connected Components(Kosaraju Algo)

```
from collections import defaultdict
class Graph:
    def __init__(self,vertices):
        self.V= vertices
        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):
        # Mark the current node as visited and print it
        visited[v]= True
        print (v)
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.DFSUtil(i,visited)


    def fillOrder(self,v,visited, stack):
        # Mark the current node as visited
        visited[v]= True
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.fillOrder(i, visited, stack)
        stack = stack.append(v)


    # Function that returns reverse (or transpose) of this graph
    def getTranspose(self):
        g = Graph(self.V)

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j,i)
        return g



    # The main function that finds and prints all strongly connected components
    def printSCCs(self):

        stack = []
        # Mark all the vertices as not visited (For first DFS)
```

```
        visited =[False]*(self.V)
        # Fill vertices in stack according to their finishing times
        for i in range(self.V):
            if visited[i]==False:
                self.fillOrder(i, visited, stack)

        # Create a reversed graph
        gr = self.getTranspose()

        # Mark all the vertices as not visited (For second DFS)
        visited =[False]*(self.V)

        # Now process all vertices in order defined by Stack
        while stack:
            i = stack.pop()
            if visited[i]==False:
                gr.DFSUtil(i, visited)
                print()

# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)
print ("Following are strongly connected components " + "in given graph")
g.printSCCs()
```

Check whether a graph is Bipartite or Not

```
V = 4

def colorGraph(G, color, pos, c):
    if color[pos] not in [-1, c]:
        return False

    # color this pos as c and all its neighbours and 1-c
    color[pos] = c
    ans = True
    for i in range(V):
        if G[pos][i]:
            if color[i] == -1:
                ans &= colorGraph(G, color, i, 1-c)
            if color[i] not in [-1, 1 - c]:
                return False
        if not ans:
            return False
    return True

def isBipartite(G):
    color = [-1] * V
    #start is vertex 0
    pos = 0
    # two colors 1 and 0
    return colorGraph(G, color, pos, 1)

G = [[0, 1, 0, 1],
    [1, 0, 1, 0],
```

```
    [0, 1, 0, 1],
    [1, 0, 1, 0]]

if isBipartite(G): print("Yes")
else: print("No")
```

Longest path in a Directed Acyclic Graph

```
def topologicalSortUtil(v):
    global Stack, visited, adj
    visited[v] = True
    for i in adj[v]:
        if (not visited[i[0]]):
            topologicalSortUtil(i[0])
    Stack.append(v)

# The function to find longest distances from a given vertex. It uses recursive
topologicalSortUtil() to get topological sorting.
def longestPath(s):
    global Stack, visited, adj, V
    dist = [-10**9 for _ in range(V)]

    # Call the recursive helper function to store Topological Sort starting from all vertices one
by one
    for i in range(V):
        if (visited[i] == False):
            topologicalSortUtil(i)

    # Initialize distances to all vertices as infinite and distance to source as 0
    dist[s] = 0

    # Process vertices in topological order
    while (len(Stack) > 0):

        # Get the next vertex from topological order
        u = Stack[-1]
        del Stack[-1]

        # Update distances of all adjacent vertices
        if (dist[u] != 10**9):
            for i in adj[u]:
                if (dist[i[0]] < dist[u] + i[1]):
                    dist[i[0]] = dist[u] + i[1]

    # Print calculated longest distances print(dist)
    for i in range(V):
        print("INF ",end="") if (dist[i] == -10**9) else print(dist[i],end=" ")


V, Stack, visited = 6, [], [False for _ in range(7)]
adj = [[] for _ in range(7)]
# Create a graph given in the above diagram.
# Here vertex numbers are 0, 1, 2, 3, 4, 5 with following mappings:
# 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
adj[0].append([1, 5])
adj[0].append([2, 3])
adj[1].append([3, 6])
adj[1].append([2, 2])
adj[2].append([4, 4])
```

```
adj[2].append([5, 2])
adj[2].append([3, 7])
adj[3].append([5, 1])
adj[3].append([4, -1])
adj[4].append([5, -2])
s = 1
print("Following are longest distances from source vertex ",s)
longestPath(s)
```

Journey to the Moon

```
TODO
```

Cheapest Flights Within K Stops

```
TODO
```

Oliver and the Game

```
TODO
```

Water Jug problem using BFS

```
"""
You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2
Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)
Examples:

Input : 4 3 2
Output : {(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)}
"""

from collections import deque

def BFS(a, b, target):

    # Map is used to store the states, every state is hashed to binary value to indicate either that state is visited before or not
    m = {}
    isSolvable = False
    path = []
```

```
    # Queue to maintain states
    q = deque()

    # Initialing with initial state
    q.append((0, 0))

    while (len(q) > 0):
        # Current state
        u = q.popleft()

        #q.pop() #pop off used state

        # If this state is already visited
        if ((u[0], u[1]) in m):
            continue

        # Doesn't met jug constraints
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue

        # Filling the vector for constructing the solution path
        path.append([u[0], u[1]])

        # Marking current state as visited
        m[(u[0], u[1])] = 1

        # If we reach solution state, put ans=1
        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
                if (u[1] != 0):

                    # Fill final state
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):

                    # Fill final state
                    path.append([0, u[1]])

            # Print the solution path
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                        path[i][1], ")")
            break

        # If we have not reached final state then, start developing intermediate states to reach
solution state
        q.append([u[0], b]) # Fill Jug2
        q.append([a, u[1]]) # Fill Jug1

        for ap in range(max(a, b) + 1):

            # Pour amount ap from Jug2 to Jug1
            c = u[0] + ap
            d = u[1] - ap
```

```
            # Check if this state is possible or not
            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            # Pour amount ap from Jug 1 to Jug2
            c = u[0] - ap
            d = u[1] + ap

            # Check if this state is possible or not
            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        # Empty Jug2
        q.append([a, 0])

        # Empty Jug1
        q.append([0, b])

    # No, solution exists if ans=0
    if (not isSolvable):
        print ("No solution")


Jug1, Jug2, target = 4, 3, 2
print("Path from initial state to solution state ::")
BFS(Jug1, Jug2, target)
```

Find if there is a path of more thank length from a source

```
"""
Given a graph, a source vertex in the graph and a number k, find if there is a simple path
(without any cycle) starting from given source and ending at any other vertex such that the
distance from source to that vertex is atleast ‘k’ length.

Example:

Input  : Source s = 0, k = 58
Output : True
There exists a simple path 0 -> 7 -> 1
-> 2 -> 8 -> 6 -> 5 -> 3 -> 4
Which has a total distance of 60 km which
is more than 58.

Input  : Source s = 0, k = 62
Output : False

In the above graph, the longest simple
path has distance 61 (0 -> 7 -> 1-> 2
 -> 3 -> 4 -> 5-> 6 -> 8, so output
should be false for any input greater
than 61.
"""
# Program to find if there is a simple path with
# weight more than k

# This class represents a dipathted graph using
# adjacency list representation
class Graph:
```

```
    # Allocates memory for adjacency list
    def __init__(self, V):
        self.V = V
        self.adj = [[] for _ in range(V)]

    # Returns true if graph has path more than k length
    def pathMoreThanK(self,src, k):
        # Create a path array with nothing included in path
        path = [False]*self.V

        # Add source vertex to path
        path[src] = 1

        return self.pathMoreThanKUtil(src, k, path)

    # Prints shortest paths from src to all other vertices
    def pathMoreThanKUtil(self,src, k, path):
        # If k is 0 or negative, return true
        if (k <= 0):
            return True

        # Get all adjacent vertices of source vertex src and recursively explore all paths from
src.
        i = 0
        while i != len(self.adj[src]):
            # Get adjacent vertex and weight of edge
            v = self.adj[src][i][0]
            w = self.adj[src][i][1]
            i += 1

            # If vertex v is already there in path, then there is a cycle (we ignore this edge)
            if (path[v] == True):
                continue

            # If weight of is more than k, return true
            if (w >= k):
                return True

            # Else add this vertex to path
            path[v] = True

            # If this adjacent can provide a path longer than k, return true.
            if (self.pathMoreThanKUtil(v, k-w, path)):
                return True

            # Backtrack
            path[v] = False

        # If no adjacent could produce longer path, return false
        return False

    # Utility function to an edge (u, v) of weight w
    def addEdge(self,u, v, w):
        self.adj[u].append([v, w])
        self.adj[v].append([u, w])

# create the graph given in above figure
V = 9
g = Graph(V)
```

```
# making above shown graph
g.addEdge(0, 1, 4)
g.addEdge(0, 7, 8)
g.addEdge(1, 2, 8)
g.addEdge(1, 7, 11)
g.addEdge(2, 3, 7)
g.addEdge(2, 8, 2)
g.addEdge(2, 5, 4)
g.addEdge(3, 4, 9)
g.addEdge(3, 5, 14)
g.addEdge(4, 5, 10)
g.addEdge(5, 6, 2)
g.addEdge(6, 7, 1)
g.addEdge(6, 8, 6)
g.addEdge(7, 8, 7)

src = 0
k = 62
if g.pathMoreThanK(src, k):
    print("Yes")
else:
    print("No")

k = 60
if g.pathMoreThanK(src, k):
    print("Yes")
else:
    print("No")
```

Minimum edges to reverse o make path from source to destination

```
"""
Given a directed graph and a source node and destination node, we need to find how many edges we
need to reverse in order to make at least 1 path from the source node to the destination node.
"""

def addEdge(u, v, w):
    global adj
    adj[u].append((v, w))

def shortestPath(src):

    # Create a set to store vertices that are being preprocessed
    setds = {}

    # Create a vector for distances and initialize all distances as infinite (INF)
    dist = [10**18 for _ in range(V)]

    # Insert source itself in Set and initialize its
    global adj
    setds[(0, src)] = 1
    dist[src] = 0

    while setds:

        # The first vertex in Set is the minimum distance vertex, extract it from set.
        tmp = list(setds.keys())[0]
```

```
        del setds[tmp]

        # vertex label is stored in second of pair (it has to be done this way to keep the
vertices sorted distance (distance must be first item in pair)
        u = tmp[1]

        # 'i' is used to get all adjacent vertices of a vertex
        # list< pair<int, int> >::iterator i;
        for i in adj[u]:

            # Get vertex label and weight of current adjacent
            # of u.
            v = i[0];
            weight = i[1]

            # If there is shorter path to v through u.
            if (dist[v] > dist[u] + weight):

                # /* If distance of v is not INF then it must be in
                #    our set, so removing it and inserting again
                #    with updated less distance.
                #    Note : We extract only those vertices from Set
                #    for which distance is finalized. So for them,
                #    we would never reach here. */
                if (dist[v] != 10**18):
                    del setds[(dist[v], v)]

                # Updating distance of v
                dist[v] = dist[u] + weight
                setds[(dist[v], v)] = 1

    return dist

# method adds reverse edge of each original edge in the graph. It gives reverse edge a weight = 1
and all original edges a weight of 0. Now, the length of the shortest path will give us the
answer. If shortest path is p: it means we used p reverse edges in the shortest path.
def modelGraphWithEdgeWeight(edge, E, V):
    global adj
    for i in range(E):
        addEdge(edge[i][0], edge[i][1], 0) # original edge : weight 0
        addEdge(edge[i][1], edge[i][0], 1) # reverse edge : weight 1

# Method returns minimum number of edges to be reversed to reach from src to dest
def getMinEdgeReversal(edge, E, V,src, dest):
    # get modified graph with edge weight
    modelGraphWithEdgeWeight(edge, E, V)

    # get shortes path vector
    dist = shortestPath(src)

    # If distance of destination is still INF, not possible
    return -1 if (dist[dest] == 10**18) else dist[dest]



V = 7
edge = [[0, 1], [2, 1], [2, 3], [5, 1],[4, 5], [6, 4], [6, 3]]
E, adj = len(edge), [[] for _ in range(V + 1)]
minEdgeToReverse = getMinEdgeReversal(edge, E, V, 0, 6)
if (minEdgeToReverse != -1):
    print(minEdgeToReverse)
```

```
else:
    print("Not possible")
```

Paths to travel each nodes using each edge(Seven Bridges)

```
TODO
```

Vertex Cover Problem

```
"""
There are n nodes and m bridges in between these nodes. Print the possible path through each node
using each edges (if possible), traveling through each edges only once.
"""

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)


    def addEdge(self, u, v):
        self.graph[u].append(v)


    def printVertexCover(self):
        # Initialize all vertices as not visited.
        visited = [False] * (self.V)

        # Consider all edges one by one
        for u in range(self.V):

            # An edge is only picked when both visited[u] and visited[v] are false
            if not visited[u]:

                # Go through all adjacents of u and pick the first not yet visited
                # vertex (We are basically picking an edge (u, v) from remaining edges.
                for v in self.graph[u]:
                    if not visited[v]:

                        # Add the vertices (u, v) to the
                        # result set. We make the vertex u and v visited so that all
                        # edges from/to them would be ignored
                        visited[v] = True
                        visited[u] = True
                        break

        # Print the vertex cover
        for j in range(self.V):
            if visited[j]:
                print(j, end = ' ')
        print()

g = Graph(7)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
```

```
g.addEdge(3, 4)
g.addEdge(4, 5)
g.addEdge(5, 6)
g.printVertexCover()
```

Chinese Postman or Route Inspection

```
TODO
```

Number of Triangles in a Directed and Undirected Graph

```
"""
Given a Graph, count number of triangles in it. The graph is can be directed or undirected.

Example:

Input: digraph[V][V] = { {0, 0, 1, 0},
                        {1, 0, 0, 1},
                        {0, 1, 0, 0},
                        {0, 0, 1, 0}
                      };
Output: 2
Give adjacency matrix represents following
directed graph.
"""

# function to calculate the number of triangles in a simple directed/undirected graph.
# isDirected is true if the graph is directed, its false otherwise
def countTriangle(g, isDirected):
    nodes = len(g)
    count_Triangle = 0
    # Consider every possible triplet of edges in graph
    for i in range(nodes):
        for j in range(nodes):
            for k in range(nodes):
                # check the triplet if it satisfies the condition
                if(i != j and i != k and j != k and g[i][j] and g[j][k] and g[k][i]):
                    count_Triangle += 1

    # If graph is directed , division is done by 3 else division by 6 is done
    if isDirected:
      return (count_Triangle//3)
    else:
      return (count_Triangle//6)


# Create adjacency matrix of an undirected graph
graph = [[0, 1, 1, 0],
        [1, 0, 1, 1],
        [1, 1, 0, 1],
        [0, 1, 1, 0]]
# Create adjacency matrix of a directed graph
digraph = [[0, 0, 1, 0],
        [1, 0, 0, 1],
        [0, 1, 0, 0],
        [0, 0, 1, 0]]

print("The Number of triangles in undirected graph : %d" %
```

```
    countTriangle(graph, False))

print("The Number of triangles in directed graph : %d" %
    countTriangle(digraph, True))
```

[Minimise the cashflow among a given set of friends who have borrowed money from each other]()

```
"""
Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.
"""

# Number of persons(or vertices in graph)
N = 3

# A utility function that returns index of minimum value in arr[]
def getMin(arr):
    minInd = 0
    for i in range(1, N):
        if (arr[i] < arr[minInd]):
            minInd = i
    return minInd

# A utility function that returns index of maximum value in arr[]
def getMax(arr):
    maxInd = 0
    for i in range(1, N):
        if (arr[i] > arr[maxInd]):
            maxInd = i
    return maxInd


def minOf2(x, y):
    return x if x < y else y

# amount[p] indicates the net amount to be credited/debited to/from person 'p' If amount[p] is positive, then i'th person will amount[i] If amount[p] is negative, then i'th person will give -amount[i]
def minCashFlowRec(amount):

    # Find the indexes of minimum and maximum values in amount[] amount[mxCredit] indicates the maximum amount to be given(or credited) to any person. And amount[mxDebit] indicates the maximum amount to be taken (or debited) from any person. So if there is a positive value in amount[], then there must be a negative value
    mxCredit = getMax(amount)
    mxDebit = getMin(amount)

    # If both amounts are 0, then all amounts are settled
    if (amount[mxCredit] == 0 and amount[mxDebit] == 0):
        return 0

    # Find the minimum of two amounts
    min = minOf2(-amount[mxDebit], amount[mxCredit])
    amount[mxCredit] -=min
    amount[mxDebit] += min

    # If minimum is the maximum amount to be
```

```
    print("Person " , mxDebit , " pays " , min
        , " to " , "Person " , mxCredit)

    # Recur for the amount array. Note that it is guaranteed that the recursion would terminate as either amount[mxCredit] or amount[mxDebit] becomes 0
    minCashFlowRec(amount)

# Given a set of persons as graph[] where graph[i][j] indicates the amount that person i needs to pay person j, this function finds and prints the minimum cash flow to settle all debts.
def minCashFlow(graph):

    # Create an array amount[], initialize all value in it as 0.
    amount = [0 for _ in range(N)]

    # Calculate the net amount to be paid to person 'p', and stores it in amount[p]. The value of amount[p] can be calculated by subtracting debts of 'p' from credits of 'p'
    for p in range(N):
        for i in range(N):
            amount[p] += (graph[i][p] - graph[p][i])

    minCashFlowRec(amount)


# graph[i][j] indicates the amount that person i needs to pay person j
graph = [ [0, 1000, 2000],
        [0, 0, 5000],
        [0, 0, 0] ]
minCashFlow(graph)
```

Two Clique Problem

```
"""
A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other. Given a Graph, find if it can be divided into two Cliques.

Examples:

Input : G[][] =   {{0, 1, 1, 0, 0},
                  {1, 0, 1, 1, 0},
                  {1, 1, 0, 0, 0},
                  {0, 1, 0, 0, 1},
                  {0, 0, 0, 1, 0}};
Output : Yes
"""

from queue import Queue

# This function returns true if subgraph reachable from src is Bipartite or not.
def isBipartiteUtil(G, src, colorArr):
    global V
    colorArr[src] = 1

    # Create a queue (FIFO) of vertex numbers and enqueue source vertex for BFS traversal
    q = Queue()
    q.put(src)

    # Run while there are vertices in queue (Similar to BFS)
    while (not q.empty()):
```

```
        # Dequeue a vertex from queue
        u = q.get()

        # Find all non-colored adjacent vertices
        for v in range(V):

            # An edge from u to v exists and destination v is not colored
            if (G[u][v] and colorArr[v] == -1):

                # Assign alternate color to this adjacent v of u
                colorArr[v] = 1 - colorArr[u]
                q.put(v)

            # An edge from u to v exists and destination v is colored with same color as u
            elif (G[u][v] and colorArr[v] == colorArr[u]):
                return False

    # If we reach here, then all adjacent vertices can be colored with alternate color
    return True

# Returns true if a Graph G[][] is Bipartite or Note that G may not be connected.
def isBipartite(G):
    global V
    # Create a color array to store colors assigned to all vertices. Vertex number is used as
index in this array. The value '-1' of colorArr[i] is used to indicate that no color is assigned
to vertex 'i'. The value 1 is used to indicate first color is assigned and value 0 indicates
second color is assigned.
    colorArr = [-1] * V

    # One by one check all not yet colored vertices.
    for i in range(V):
        if (colorArr[i] == -1):
            if (isBipartiteUtil(G, i, colorArr) == False):
                return False

    return True

# Returns true if G can be divided into
# two Cliques, else false.
def canBeDividedinTwoCliques(G):
    global V
    # Find complement of G[][] All values are complemented except diagonal ones
    GC = [[None] * V for _ in range(V)]
    for i in range(V):
        for j in range(V):
            GC[i][j] = not G[i][j] if i != j else 0

    # Return true if complement is Bipartite else false.
    return isBipartite(GC)



V = 5
G = [[0, 1, 1, 1, 0],
    [1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0],
    [0, 1, 0, 0, 1],
    [0, 0, 0, 1, 0]]
if canBeDividedinTwoCliques(G):
    print("Yes")
else:
```

```
    print("No")
```

Greedy

Activity Selection Problem

```
"""
There is one meeting room in a firm. There are N meetings in the form of (start[i], end[i]) where
start[i] is start time of meeting i and end[i] is finish time of meeting i.
What is the maximum number of meetings that can be accommodated in the meeting room when only one
meeting can be held in the meeting room at a particular time?

Note: Start time of one chosen meeting can't be equal to the end time of the other chosen meeting.


Example 1:

Input:
N = 6
start[] = {1,3,0,5,8,5}
end[] =  {2,4,6,7,9,9}
Output:
4
Explanation:
Maximum four meetings can be held with
given start and end timings.
The meetings are - (1, 2),(3, 4), (5,7) and (8,9)
"""

class meeting:
    def __init__(self, start, end, pos):
        self.start = start
        self.end = end
        self.pos = pos


def maxMeeting(l, n):

    # Sorting of meeting according to heir finish time.
    l.sort(key = lambda x: x.end)

    ans = [l[0].pos]
    # time_limit to check whether new meeting can be conducted or not.
    time_limit = l[0].end

    # Check for all meeting whether it can be selected or not.
    for i in range(1, n):
        if l[i].start > time_limit:
            ans.append(l[i].pos)
            time_limit = l[i].end

    # Print final selected meetings
    for i in ans:
        print(i + 1, end = "")
    print()
```

```
s = [ 1, 3, 0, 5, 8, 5 ]    # Starting time
f = [ 2, 4, 6, 7, 9, 9 ]    # Finish time
n = len(s)
l = [meeting(s[i], f[i], i) for i in range(n)]
maxMeeting(l, n)
```

Huffman Coding

```
# A Huffman Tree Node
class node:
    def __init__(self, freq, symbol, left=None, right=None):
        # frequency of symbol
        self.freq = freq

        # symbol name (character)
        self.symbol = symbol

        # node left of current node
        self.left = left

        # node right of current node
        self.right = right

        # tree direction (0/1)
        self.huff = ''


def printNodes(node, val=''):
    # huffman code for current node
    newVal = val + str(node.huff)

    # if node is not an edge node then traverse inside it
    if(node.left):
        printNodes(node.left, newVal)
    if(node.right):
        printNodes(node.right, newVal)

        # if node is edge node then display its huffman code
    if(not node.left and not node.right):
        print(f"{node.symbol} -> {newVal}")


# characters for huffman tree
chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters
freq = [ 5, 9, 12, 13, 16, 45]

# list containing unused nodes
nodes = [node(freq[x], chars[x]) for x in range(len(chars))]

while len(nodes) > 1:
    # sort all the nodes in ascending order based on theri frequency
    nodes = sorted(nodes, key=lambda x: x.freq)

    # pick 2 smallest nodes
    left = nodes[0]
    right = nodes[1]
```

```
    # assign directional value to these nodes
    left.huff = 0
    right.huff = 1

    # combine the 2 smallest nodes to create new node as their parent
    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

    # remove the 2 nodes and add their parent as new node among others
    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)

# Huffman Tree is ready!
printNodes(nodes[0])
```

Water Connection Problem

```
"""Every house in the colony has at most one pipe going into it and at most one pipe going out of
it. Tanks and taps are to be installed in a manner such that every house with one outgoing pipe
but no incoming pipe gets a tank installed on its roof and every house with only an incoming pipe
and no outgoing pipe gets a tap.

Given two integers n and p denoting the number of houses and the number of pipes. The connections
of pipe among the houses contain three input values: a_i, b_i, d_i denoting the pipe of diameter
d_i from house a_i to house b_i, find out the efficient solution for the network.

The output will contain the number of pairs of tanks and taps t installed in first line and the
next t lines contain three integers: house number of tank, house number of tap and the minimum
diameter of pipe between them.

Examples:


Input:  4 2
        1 2 60
        3 4 50
Output: 2
        1 2 60
        3 4 50
Explanation:
Connected components are: 1->2 and 3->4
Therefore, our answer is 2 followed by 1 2 60 and 3 4 50.

Input: 9 6
       7 4 98
       5 9 72
       4 6 10
       2 8 22
       9 7 17
       3 1 66
Output: 3
        2 8 22
        3 1 66
        5 6 10
Explanation:
Connected components are 3->1, 5->9->7->4->6 and 2->8.
Therefore, our answer is 3 followed by 2 8 22, 3 1 66, 5 6 10
"""
```

```python
# number of houses and number of pipes
n = 0
p = 0

# Array rd stores the ending vertex of pipe
rd = [0]*1100

# Array wd stores the value of diameters between two pipes
wt = [0]*1100

# Array cd stores the starting end of pipe
cd = [0]*1100

# List a, b, c are used to store the final output
a = []
b = []
c = []

ans = 0

def dfs(w):
    global ans
    if (cd[w] == 0):
        return w
    if (wt[w] < ans):
        ans = wt[w]
    return dfs(cd[w])

# Function performing calculations.
def solve(arr):
    global ans
    i = 0
    while (i < p):
        q = arr[i][0]
        h = arr[i][1]
        t = arr[i][2]
        cd[q] = h
        wt[q] = t
        rd[h] = q
        i += 1
    a = []
    b = []
    c = []

    # If a pipe has no ending vertex but has starting vertex i.e is an outgoing pipe then we need
to start DFS with this vertex.
    for j in range(1, n + 1):
        if (rd[j] == 0 and cd[j]):
            ans = 1000000000
            w = dfs(j)
            # We put the details of component in final output array
            a.append(j)
            b.append(w)
            c.append(ans)
    print(len(a))
    for j in range(len(a)):
        print(a[j], b[j], c[j])


n = 9 # number of houses
```

```
p = 6 # number of pipes
arr = [[7, 4, 98], [5, 9, 72], [4, 6, 10 ], [2, 8, 22 ], [9, 7, 17], [3, 1, 66]]
solve(arr)
```

Fractional Knapsack Problem

```

```

Greedy Algorithm to find Minimum number of Coins

```
"""
Given the weights and values of n items, we need to put these items in a knapsack of capacity W to
get the maximum total value in the knapsack.
In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This
problem in which we can break an item is also called the fractional knapsack problem.
Input:
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;

Output:
Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240
"""

class ItemValue:
    """Item Value DataClass"""
    def __init__(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost


def getMaxValue(wt, val, capacity):
    """function to get maximum value """
    iVal = [ItemValue(wt[i], val[i], i) for i in range(len(wt))]
    # sorting items by value
    iVal.sort(reverse=True)
    totalValue = 0
    for i in iVal:
        curWt = int(i.wt)
        curVal = int(i.val)
        if capacity - curWt >= 0:
            capacity -= curWt
            totalValue += curVal
        else:
            fraction = capacity / curWt
            totalValue += curVal * fraction
            capacity = int(capacity - (curWt * fraction))
            break
    return totalValue
```

```
wt = [10, 40, 20, 30]
val = [60, 40, 100, 120]
capacity = 50
maxValue = getMaxValue(wt, val, capacity)
print("Maximum value in Knapsack =", maxValue)
```

Maximum trains for which stoppage can be provided

```
TODO
```

Minimum Platforms Problem

```
"""
Given the arrival and departure times of all trains that reach a railway station, the task is to
find the minimum number of platforms required for the railway station so that no train waits.
We are given two arrays that represent the arrival and departure times of trains that stop.

Examples:

Input: arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}
dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}
Output: 3
Explanation: There are at-most three trains at a time (time between 9:40 to 12:00)

Input: arr[] = {9:00, 9:40}
dep[] = {9:10, 12:00}
Output: 1
Explanation: Only one platform is needed.
"""

def findPlatform(arr, dep, n):
    # Sort arrival and departure arrays
    arr.sort()
    dep.sort()

    # plat_needed indicates number of platforms needed at a time
    plat_needed = 1
    result = 1
    i = 1
    j = 0

    # Similar to merge in merge sort to process all events in sorted order
    while (i < n and j < n):
        # If next event in sorted order is arrival, increment count of platforms needed
        if (arr[i] <= dep[j]):
            plat_needed += 1
            i += 1
        else:
            plat_needed -= 1
            j += 1

        # Update result if needed
        if (plat_needed > result):
            result = plat_needed
    return result

arr = [900, 940, 950, 1100, 1500, 1800]
dep = [910, 1200, 1120, 1130, 1900, 2000]
```

```
n = len(arr)
print("Minimum Number of Platforms Required = ",
    findPlatform(arr, dep, n))
```

Buy Maximum Stocks if i stocks can be bought on i-th day

```
"""
In a stock market, there is a product with its infinite stocks. The stock prices are given for N
days, where arr[i] denotes the price of the stock on the ith day. There is a rule that a customer
can buy at most i stock on the ith day. If the customer has an amount of k amount of money
initially, find out the maximum number of stocks a customer can buy.

For example, for 3 days the price of a stock is given as 7, 10, 4. You can buy 1 stock worth 7 rs
on day 1, 2 stocks worth 10 rs each on day 2 and 3 stock worth 4 rs each on day 3.

Examples:

Input : price[] = { 10, 7, 19 },
              k = 45.
Output : 4
A customer purchases 1 stock on day 1,
2 stocks on day 2 and 1 stock on day 3 for
10, 7 * 2 = 14 and 19 respectively. Hence,
total amount is 10 + 14 + 19 = 43 and number
of stocks purchased is 4.

Input  : price[] = { 7, 10, 4 },
               k = 100.
Output : 6
"""

def buyMaximumProducts(n, k, price):
    # Making pair of stock cost and day number
    arr = [[i + 1, price[i]] for i in range(n)]

    # Sort based on the price of stock
    arr.sort(key = lambda x: x[1])

    # Calculating the max stocks purchased
    total_purchase = 0
    for i in range(n):
        P = min(arr[i][0], k//arr[i][1])
        total_purchase += P
        k -= (P * arr[i][1])
    return total_purchase

price = [ 10, 7, 19 ]
n = len(price)
k = 45
print(buyMaximumProducts(n, k, price))
```

Find the minimum and maximum amount to buy all N candies

```
"""
In a candy store, there are N different types of candies available and the prices of all the N
different types of candies are provided. There is also an attractive offer by the candy store. We
can buy a single candy from the store and get at most K other candies (all are different types)
for free.
```

```
Find the minimum amount of money we have to spend to buy all the N different candies.
Find the maximum amount of money we have to spend to buy all the N different candies.
In both cases, we must utilize the offer and get the maximum possible candies back. If k or more candies are available, we must take k candies for every candy purchase. If less than k candies are available, we must take all candies for a candy purchase.

Examples:

    Input :
    price[] = {3, 2, 1, 4}
    k = 2
    Output :
    Min = 3, Max = 7
    Explanation :
    Since k is 2, if we buy one candy we can take  atmost two more for free. So in the first case we buy the candy which  costs 1 and take candies worth 3 and 4 for  free, also you buy candy worth 2 as well. So min cost = 1 + 2 = 3. In the second case we buy the candy which  costs 4 and take candies worth 1 and 2 for  free, also We buy candy worth 3 as well. So max cost = 3 + 4 = 7.

One important thing to note is, we must use the offer and get maximum candies back for every candy purchase. So if we want to minimize the money, we must buy candies at minimum cost and get candies of maximum costs for free
"""

from math import ceil

# function to find the maximum and the minimum cost required
def find(arr,n,k):

    # Sort the array
    arr.sort()
    b = int(ceil(n/k))

    # print the minimum cost
    print("minimum ",sum(arr[:b]))

    # print the maximum cost
    print("maximum ", sum(arr[-b:]))

arr = [3, 2, 1, 4]
n = len(arr)
k = 2
find(arr,n,k)
```

Minimum Cost to cut a board into squares

```
"""
A board of length m and width n is given, we need to break this board into m*n squares such that cost of breaking is minimum. cutting cost for each edge will be given for the board. In short, we need to choose such a sequence of cutting such that cost is minimized.
https://media.geeksforgeeks.org/wp-content/cdn-uploads/board.png
For above board optimal way to cut into square is:
Total minimum cost in above case is 42. It is
evaluated using following steps.

Initial Value : Total_cost = 0
Total_cost = Total_cost + edge_cost * total_pieces
```

```
Cost 4 Horizontal cut         Cost = 0 + 4*1 = 4
Cost 4 Vertical cut        Cost = 4 + 4*2 = 12
Cost 3 Vertical cut        Cost = 12 + 3*2 = 18
Cost 2 Horizontal cut        Cost = 18 + 2*3 = 24
Cost 2 Vertical cut        Cost = 24 + 2*3 = 30
Cost 1 Horizontal cut        Cost = 30 + 1*4 = 34
Cost 1 Vertical cut        Cost = 34 + 1*4 = 38
Cost 1 Vertical cut        Cost = 38 + 1*4 = 42
"""



def minimumCostOfBreaking(X, Y, m, n):
    res = 0
    # sort the horizontal cost in reverse order
    X.sort(reverse = True)
    # sort the vertical cost in reverse order
    Y.sort(reverse = True)
    # initialize current width as 1
    hzntl = 1; vert = 1

    # loop until one or both cost array are processed
    i = 0; j = 0
    while (i < m and j < n):
        if (X[i] > Y[j]):
            res += X[i] * vert
            # increase current horizontal part count by 1
            hzntl += 1
            i += 1

        else:
            res += Y[j] * hzntl
            # increase current vertical part count by 1
            vert += 1
            j += 1

    # loop for horizontal array, if remains
    total = 0
    while (i < m):
        total += X[i]
        i += 1
    res += total * vert

    #loop for vertical array, if remains
    total = 0
    while (j < n):
        total += Y[j]
        j += 1
    res += total * hzntl
    return res

m = 6; n = 4
X = [2, 1, 3, 1, 4]
Y = [4, 1, 2]
print(minimumCostOfBreaking(X, Y, m-1, n-1))
```

Check if it is possible to survive on Island

```
"""
```

```
You are a poor person in an island. There is only one shop in this island, this shop is open on
all days of the week except for Sunday. Consider following constraints:

N – Maximum unit of food you can buy each day.
S – Number of days you are required to survive.
M – Unit of food required each day to survive.
Currently, it's Monday, and you need to survive for the next S days.
Find the minimum number of days on which you need to buy food from the shop so that you can
survive the next S days, or determine that it isn't possible to survive.
Examples:

Input : S = 10 N = 16 M = 2
Output : Yes 2
Explanation 1: One possible solution is to buy a box on the first day (Monday), it's sufficient to
eat from this box up to 8th day (Monday) inclusive. Now, on the 9th day (Tuesday), you buy another
box and use the chocolates in it to survive the 9th and 10th day.
Input : 10 20 30
Output : No
Explanation 2: You can't survive even if you buy food because the maximum number of units you can
buy in one day is less the required food for one day.
"""



def survival(S, N, M):
# If we can not buy at least a week supply of food during the first week OR We can not buy a day
supply of food on the first day then we can't survive.
    if (((N * 6) < (M * 7) and S > 6) or M > N):
        print("No")
    else:
    # If we can survive then we can buy ceil(A / N) times where A is total units of food required.
        days = (M * S) / N
        if (((M * S) % N) != 0):
            days += 1
        print("Yes "),
        print(days)


S = 10; N = 16; M = 2
survival(S, N, M)
```

Maximum product subset of an array

```
"""
Given an array a, we have to find maximum product possible with the subset of elements present in
the array. The maximum product can be single element also.
Examples:

Input: a[] = { -1, -1, -2, 4, 3 }
Output: 24
Explanation : Maximum product will be ( -2 * -1 * 4 * 3 ) = 24

Input: a[] = { -1, 0 }
Output: 0
Explanation: 0(single element) is maximum product possible

Input: a[] = { 0, 0, 0 }
Output: 0
"""
```

```
def maxProductSubset(a, n):
    if n == 1:
        return a[0]

    # Find count of negative numbers, count of zeros, negative number with least absolute value
and product of non-zero numbers
    max_neg = -999999999999
    count_neg = 0
    count_zero = 0
    prod = 1
    for i in range(n):

        # If number is 0, we don't multiply it with product.
        if a[i] == 0:
            count_zero += 1
            continue

        # Count negatives and keep track of negative number with least absolute value.
        if a[i] < 0:
            count_neg += 1
            max_neg = max(max_neg, a[i])
        prod = prod * a[i]

    # If there are all zeros
    if count_zero == n:
        return 0

    # If there are odd number of negative numbers
    if count_neg & 1:
        # Exceptional case: There is only negative and all other are zeros
        if (count_neg == 1 and count_zero > 0 and
            count_zero + count_neg == n):
            return 0
        # Otherwise result is product of all non-zeros divided by negative number with least
absolute value
        prod = int(prod / max_neg)
    return prod

a = [ -1, -1, -2, 4, 3 ]
n = len(a)
print(maxProductSubset(a, n))
```

Maximize array sum after K negations

```
"""
Given an array of size n and a number k. We must modify array K a number of times. Here modify
array means in each operation we can replace any array element arr[i] by -arr[i]. We need to
perform this operation in such a way that after K operations, the sum of the array must be
maximum?

Examples :

Input : arr[] = {-2, 0, 5, -1, 2}, K = 4
Output: 10
Explanation:
1. Replace (-2) by -(-2), array becomes {2, 0, 5, -1, 2}
2. Replace (-1) by -(-1), array becomes {2, 0, 5, 1, 2}
3. Replace (0) by -(0), array becomes {2, 0, 5, 1, 2}
4. Replace (0) by -(0), array becomes {2, 0, 5, 1, 2}
```

```
Input : arr[] = {9, 8, 8, 5}, K = 3
Output: 20
"""

def sol(arr, k):
    # Sorting given array using in-built java sort function
    arr.sort()
    i = 0
    while (k > 0):
        # If we find a 0 in our sorted array, we stop
        if (arr[i] >= 0):
            k = 0
        else:
            arr[i] = (-1) * arr[i]
            k = k - 1
        i += 1
    return sum(arr[j] for j in range(len(arr)))

arr = [-2, 0, 5, -1, 2]
print(sol(arr, 4))
```

Maximize the sum of arr[i]*i

```
"""
Given an array of N integers. You are allowed to rearrange the elements of the array. The task is
to find the maximum value of Σarr[i]*i, where i = 0, 1, 2,…., n – 1.

Examples:

Input : N = 4, arr[] = { 3, 5, 6, 1 }
Output : 31
If we arrange arr[] as { 1, 3, 5, 6 }.
Sum of arr[i]*i is 1*0 + 3*1 + 5*2 + 6*3
= 31, which is maximum

Input : N = 2, arr[] = { 19, 20 }
Output : 20
"""
def maxSum(arr,n):
    arr.sort()
    return sum(arr[i] * i for i in range(n))

arr = [3,5,6,1]
n = len(arr)
print(maxSum(arr,n))
```

Maximum sum of absolute difference of an array

```
"""
Given an array, we need to find the maximum sum of absolute difference of any permutation of the
given array.
Examples:


Input : { 1, 2, 4, 8 }
Output : 18
Explanation : For the given array there are
```

```
several sequence possible
like : {2, 1, 4, 8}
       {4, 2, 1, 8} and some more.
Now, the absolute difference of an array sequence will be
like for this array sequence {1, 2, 4, 8}, the absolute
difference sum is
= |1-2| + |2-4| + |4-8| + |8-1|
= 14
For the given array, we get the maximum value for
the sequence {1, 8, 2, 4}
= |1-8| + |8-2| + |2-4| + |4-1|
= 18
"""
import numpy as np

def MaxSumDifference(a,n):
    # sort the original array so that we can retrieve the large elements from the end of array
elements
    np.sort(a);

    # In this loop first we will insert one smallest element not entered till that time in final
sequence and then enter a highest element(not entered till that time) in final sequence so that we
have large difference value. This process is repeated till all array has completely entered in
sequence. Here, we have loop till n/2 because we are inserting two elements at a time in loop.
    j = 0
    finalSequence = [0 for _ in range(n)]
    for i in range(int(n / 2)):
        finalSequence[j] = a[i]
        finalSequence[j + 1] = a[n - i - 1]
        j = j + 2

    # If there are odd elements, push the  middle element at the end.
    if (n % 2 != 0):
        finalSequence[n-1] = a[n//2 + 1]
    # variable to store the maximum sum of absolute difference
    MaximumSum = 0

    # In this loop absolute difference of elements for the final sequence is calculated.
    for i in range(n - 1):
        MaximumSum = (MaximumSum + abs(finalSequence[i] - finalSequence[i + 1]))

    # absolute difference of last element and 1st element
    MaximumSum = (MaximumSum + abs(finalSequence[n - 1] -   finalSequence[0]));
    print (MaximumSum)

a = [ 1, 2, 4, 8 ]
n = len(a)
MaxSumDifference(a, n);
```

Maximize sum of consecutive differences in a circular array

```
"""
Given an array of n elements. Consider array as circular array i.e element after an is a1. The
task is to find maximum mysum of the difference between consecutive elements with rearrangement of
array element allowed i.e after rearrangement of element find |a1 – a2| + |a2 – a3| + …… + |an – 1
– an| + |an – a1|.

Examples:
```

```
Input : arr[] = { 4, 2, 1, 8 }
Output : 18
Rearrange given array as : { 1, 8, 2, 4 }
mysum of difference between consecutive element
= |1 - 8| + |8 - 2| + |2 - 4| + |4 - 1|
= 7 + 6 + 2 + 3
= 18.

Input : arr[] = { 10, 12, 15 }
Output : 10
"""

def maxSum(arr, n):
    mysum = 0
    arr.sort()
    # Subtracting a1, a2, a3,....., a(n/2)-1, an/2 twice and adding a(n/2)+1, a(n/2)+2, a(n/2)+3,.
....., an - 1, an twice.
    for i in range(int(n / 2)):
        mysum -= (2 * arr[i])
        mysum += (2 * arr[n - i - 1])
    return mysum

arr = [4, 2, 1, 8]
n = len(arr)
print (maxSum(arr, n))
```

Minimum sum of absolute difference of pairs of two arrays

```
"""
Given two arrays a[] and b[] of equal length n. The task is to pair each element of array a to an
element in array b, such that mysum S of absolute differences of all the pairs is minimum.
Suppose, two elements a[i] and a[j] (i != j) of a are paired with elements b[p] and b[q] of b
respectively,
then p should not be equal to q.

Examples:

Input :  a[] = {3, 2, 1}
         b[] = {2, 1, 3}
Output : 0
Explanation :
 1st pairing: |3 - 2| + |2 - 1| + |1 - 3|
         = 1 + 1 + 2 = 4
 2nd pairing: |3 - 2| + |1 - 1| + |2 - 3|
         = 1 + 0 + 1 = 2
 3rd pairing: |2 - 2| + |3 - 1| + |1 - 3|
         = 0 + 2 + 2 = 4
 4th pairing: |1 - 2| + |2 - 1| + |3 - 3|
         = 1 + 1 + 0 = 2
 5th pairing: |2 - 2| + |1 - 1| + |3 - 3|
         = 0 + 0 + 0 = 0
 6th pairing: |1 - 2| + |3 - 1| + |2 - 3|
         = 1 + 2 + 1 = 4
 Therefore, 5th pairing has minimum mysum of
 absolute difference.

Input :  n = 4
         a[] = {4, 1, 8, 7}
         b[] = {2, 3, 6, 5}
```

```
Output : 6
"""

def findMinSum(a, b, n):
    # Sort both arrays
    a.sort()
    b.sort()
    # Find mysum of absolute differences
    mysum = 0

    for i in range(n):
        mysum = mysum + abs(a[i] - b[i])
    return mysum

# Both a[] and b[] must be of same size.
a = [4, 1, 8, 7]
b = [2, 3, 6, 5]
n = len(a)
print(findMinSum(a, b, n))
```

Program for Shortest Job First (or SJF) CPU Scheduling

```
TODO
```

Program for Least Recently Used (LRU) Page Replacement algorithm

```
TODO
```

Smallest subset with sum greater than all other elements

```
"""
Given an array of non-negative integers. Our task is to find minimum number of elements such that
their sum should be greater than the sum of rest of the elements of the array.
Examples :


Input : arr[] = {3, 1, 7, 1}
Output : 1
Smallest subset is {7}. Sum of
this subset is greater than all
other elements {3, 1, 1}
"""

def minElements(arr , n):
    # calculating HALF of array sum
    halfSum = 0
    for i in range(n):
        halfSum = halfSum + arr[i]
    halfSum = int(halfSum / 2)
    # sort the array in descending order.
    arr.sort(reverse = True)

    res = 0
    curr_sum = 0
```

```
    for i in range(n):

        curr_sum += arr[i]
        res += 1

        # current sum greater than sum
        if curr_sum > halfSum:
            return res
    return res


arr = [3, 1, 7, 1]
n = len(arr)
print(minElements(arr, n) )
```

Chocolate Distribution Problem

```
"""
Given an array of n integers where each value represents the number of chocolates in a packet.
Each packet can have a variable number of chocolates. There are m students, the task is to
distribute chocolate packets such that:

Each student gets one packet.
The difference between the number of chocolates in the packet with maximum chocolates and packet
with minimum chocolates given to the students is minimum.
Examples:

Input : arr[] = {7, 3, 2, 4, 9, 12, 56} , m = 3
Output: Minimum Difference is 2
Explanation:
We have seven packets of chocolates and
we need to pick three packets for 3 students
If we pick 2, 3 and 4, we get the minimum
difference between maximum and minimum packet
sizes.
"""

# arr[0..n-1] represents sizes of packets m is number of students. Returns minimum difference
between maximum and minimum values of distribution.
def findMinDiff(arr, n, m):

    # if there are no chocolates or number of students is 0
    if (m==0 or n==0):
        return 0

    # Sort the given packets
    arr.sort()

    # Number of students cannot be more than number of packets
    if (n < m):
        return -1

    # Largest number of chocolates
    min_diff = arr[n-1] - arr[0]

    # Find the subarray of size m such that difference between last (maximum in case of sorted)
and first (minimum in case of sorted) elements of subarray is minimum.
    for i in range(len(arr) - m + 1):
        min_diff = min(min_diff , arr[i + m - 1] - arr[i])
```

```
    return min_diff

arr = [12, 4, 7, 9, 2, 23, 25, 41, 30, 40, 28, 42, 30, 44, 48, 43, 50]
m = 7 # Number of students
n = len(arr)
print("Minimum difference is", findMinDiff(arr, n, m))
```

DEFKIN -Defense of a Kingdom

```
TODO
```

DIEHARD -DIE HARD

```
TODO
```

GERGOVIA -Wine trading in Gergovia

```
TODO
```

Picking Up Chicks

```
TODO
```

CHOCOLA –Chocolate

```
TODO
```

ARRANGE -Arranging Amplifiers

```
TODO
```

K Centers Problem

```
def maxindex(dist, n):
    mi = 0
    for i in range(n):
        if (dist[i] > dist[mi]):
            mi = i
    return mi

def selectKcities(n, weights, k):
    dist = [0]*n
    centers = []

    for i in range(n):
        dist[i] = 10**9
    # index of city having the maximum distance to it's closest center
    mymax = 0
    for i in range(k):
        centers.append(mymax)
        for j in range(n):
```

```
            # updating the distance of the cities to their closest centers
            dist[j] = min(dist[j], weights[mymax][j])

        # updating the index of the city with the maximum distance to it's closest center
        mymax = maxindex(dist, n)

    # Printing the maximum distance of a city to a center that is our answer print()
    print(dist[mymax])

    # Printing the cities that were chosen to be made centers
    for i in centers:
        print(i, end = " ")


n = 4
weights = [ [ 0, 4, 8, 5 ],
        [ 4, 0, 10, 7 ],
        [ 8, 10, 0, 9 ],
        [ 5, 7, 9, 0 ] ]
k = 2
selectKcities(n, weights, k)
```

Minimum Cost of ropes

```
"""There are given n ropes of different lengths, we need to connect these ropes into one rope. The
cost to connect two ropes is equal to the sum of their lengths. We need to connect the ropes with
minimum cost.

For example, if we are given 4 ropes of lengths 4, 3, 2, and 6. We can connect the ropes in the
following ways.

First, connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6, and 5.
Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
Finally connect the two ropes and all ropes have connected.
Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting
ropes. Other ways of connecting ropes would always have same or more cost. For example, if we
connect 4 and 6 first (we get three strings of 3, 2, and 10), then connect 10 and 3 (we get two
strings of 13 and 2). Finally, we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.
"""

import heapq
def minCost(arr, n):

    # Create a priority queue out of the given list
    heapq.heapify(arr)

    # Initialize result
    res = 0

    # While size of priority queue is more than 1
    while(len(arr) > 1):

        # Extract shortest two ropes from arr
        first = heapq.heappop(arr)
        second = heapq.heappop(arr)

        #Connect the ropes: update result and insert the new rope to arr
        res += first + second
        heapq.heappush(arr, first + second)
```

```
    return res

lengths = [ 4, 3, 2, 6 ]
size = len(lengths)
print(f"Total cost for connecting ropes is {str(minCost(lengths, size))}")
```

Find smallest number with given number of digits and sum of digits

```
"""
How to find the smallest number with given digit sum s and number of digits d?
Examples :

Input  : s = 9, d = 2
Output : 18
There are many other possible numbers like 45, 54, 90, etc with sum of digit as 9 and number of
digits as 2. The smallest of them is 18.

Input  : s = 20, d = 3
Output : 299
"""

def findSmallest(m,s):

    # If sum of digits is 0, then a number is possible only if number of digits is 1.
    if (s == 0):
        if(m == 1) :
            print("Smallest number is 0")
        else :
            print("Not possible")
        return

    # Sum greater than the maximum possible sum.
    if (s > 9*m):
        print("Not possible")
        return

    # Create an array to store digits of result
    res = [0 for _ in range(m+1)]

    # deduct sum by one to account for cases later (There must be 1 left for the most significant
digit)
    s -= 1

    # Fill last m-1 digits (from right to left)
    for i in range(m-1,0,-1):

        # If sum is still greater than 9, digit must be 9.
        if (s > 9):

            res[i] = 9
            s -= 9

        else:

            res[i] = s
            s = 0
```

```python
    # Whatever is left should be the most significant digit. The initially subtracted 1 is
incorporated here.
    res[0] = s + 1
    print("Smallest number is ",end="")
    for i in range(m):
        print(res[i],end="")

s = 9
m = 2
findSmallest(m, s)
```

Rearrange characters in a string such that no two adjacent are same

```python
"""
Given a string with repeated characters, the task is to rearrange characters in a string so that
no two adjacent characters are same.
Note : It may be assumed that the string has only lowercase English alphabets.

Examples:

Input: aaabc
Output: abaca

Input: aaabb
Output: ababa

Input: aa
Output: Not Possible
"""

def getMaxCountChar(count):
  maxCount = 0
  for i in range(26):
    if count[i] > maxCount:
        maxCount = count[i]
        maxChar = chr(i + ord('a'))

  return maxCount, maxChar

# Main function for rearranging the characters
def rearrangeString(S):
  n = len(S)

  # if length of string is None return False
  if not n:
      return False

  # create a hashmap for the alphabets
  count = [0] * 26
  for char in S:
      count[ord(char) - ord('a')] += 1


  maxCount, maxChar = getMaxCountChar(count)

  # if the char with maximum frequency is more than the half of the total length of the string
than return False
```

```
  if maxCount > (n + 1) // 2:
      return False

  # create a list for storing the result
  res = [None] * n
  ind = 0

  # place all occurrences of the char with maximum frequency in even positions
  while maxCount:
      res[ind] = maxChar
      ind += 2
      maxCount -= 1

  # replace the count of the char with maximum frequency to zero as all the maxChar are already
placed in the result
  count[ord(maxChar) - ord('a')] = 0

  # place all other char in the result starting from remaining even positions and then place in
the odd positions
  for i in range(26):
      while count[i] > 0:
          if ind >= n:
              ind = 1
          res[ind] = chr(i + ord('a') )
          ind += 2
          count[i] -= 1


  # convert the result list to string and return
  return ''.join(res)


myStr = 'bbbaa'
if res := rearrangeString(myStr):
  print(res)
else:
  print('Not valid string')
```

Find maximum sum possible equal sum of three stacks

```
"""
Given three stacks of the positive numbers, the task is to find the possible equal maximum sum of
the stacks with the removal of top elements allowed. Stacks are represented as an array, and the
first index of the array represent the top element of the stack.

Examples:

Input : stack1[] = { 3, 10}
  stack2[] = { 4, 5 }
  stack3[] = { 2, 1 }
Output : 0
Sum can only be equal after removing all elements
from all stacks.
"""
def maxSum(stack1, stack2, stack3, n1, n2, n3):
    sum1, sum2, sum3 = 0, 0, 0

    # Finding the initial sum of stack1.
    for i in range(n1):
```

```
        sum1 += stack1[i]

    # Finding the initial sum of stack2.
    for i in range(n2):
        sum2 += stack2[i]

    # Finding the initial sum of stack3.
    for i in range(n3):
        sum3 += stack3[i]

# As given in question, first element is top of stack..
    top1, top2, top3 = 0, 0, 0
    ans = 0

    while True:
    # If any stack is empty
        if (top1 == n1 or top2 == n2 or top3 == n3):
            return 0
    # If sum of all three stack are equal.
        if sum1 == sum2 == sum3:
            return sum1

    # Finding the stack with maximum sum and removing its top element.
        if (sum1 >= sum2 and sum1 >= sum3):
            sum1 -= stack1[top1]
            top1=top1+1
        elif (sum2 >= sum1 and sum2 >= sum3):
            sum2 -= stack2[top2]
            top2=top2+1
        elif (sum3 >= sum2 and sum3 >= sum1):
            sum3 -= stack3[top3]
            top3=top3+1


stack1 = [ 3, 2, 1, 1, 1 ]
stack2 = [ 4, 3, 2 ]
stack3 = [ 1, 1, 4, 1 ]
n1 = len(stack1)
n2 = len(stack2)
n3 = len(stack3)
print (maxSum(stack1, stack2, stack3, n1, n2, n3))
```

Heap

[Implement a Maxheap/MinHeap using arrays and recursion.]()

```
```

[Sort an Array using heap. (HeapSort)]()

```
```

[Maximum of all subarrays of size k.]()

```
```

“k” largest element in an array

Kth smallest and largest element in an unsorted array

Merge “K” sorted arrays. [IMP]

Merge 2 Binary Max Heaps

Kth largest sum continuous subarrays

Leetcode- reorganize strings

Merge “K” Sorted Linked Lists [V.IMP]

Smallest range in “K” Lists

Median in a stream of Integers

Check if a Binary Tree is Heap

Connect “n” ropes with minimum cost

Convert BST to Min Heap

Convert min heap to max heap

Rearrange characters in a string such that no two adjacent are same.

Minimum sum of two numbers formed from digits of an array

Linked List

Write a Program to reverse the Linked List. (Both Iterative and recursive)

Reverse a Linked List in group of Given Size. [Very Imp]

Write a program to Detect loop in a linked list.

Write a program to Delete loop in a linked list.

Find the starting point of the loop.

Remove Duplicates in a sorted Linked List.

Remove Duplicates in a Un-sorted Linked List.

Write a Program to Move the last element to Front in a Linked List.

Add “1” to a number represented as a Linked List.

Add two numbers represented by linked lists.

Intersection of two Sorted Linked List.

Intersection Point of two Linked Lists.

Merge Sort For Linked lists.[Very Important]

Quicksort for Linked Lists.[Very Important]

Find the middle Element of a linked list.

Check if a linked list is a circular linked list.

Split a Circular linked list into two halves.

Write a Program to check whether the Singly Linked list is a palindrome or not.

Deletion from a Circular Linked List.

Reverse a Doubly Linked list.

Find pairs with a given sum in a DLL.

Count triplets in a sorted DLL whose sum is equal to given value “X”.

Sort a “k”sorted Doubly Linked list.[Very IMP]

Rotate DoublyLinked list by N nodes.

Rotate a Doubly Linked list in group of Given Size.[Very IMP]

Can we reverse a linked list in less than O(n) ?

Why Quicksort is preferred for. Arrays and Merge Sort for LinkedLists ?

Flatten a Linked List

Sort a LL of 0's, 1's and 2's

Clone a linked list with next and random pointer

Merge K sorted Linked list

Multiply 2 no. represented by LL

Delete nodes which have a greater value on right side

Segregate even and odd nodes in a Linked List

Program for n’th node from the end of a Linked List

Find the first non-repeating character from a stream of characters

Matrix

Spiral traversal on a Matrix

Search an element in a matriix

Find median in a row wise sorted matrix

Find row with maximum no. of 1's

Print elements in sorted order using row-column wise sorted matrix

```
```

Maximum size rectangle

```
```

Find a specific pair in matrix

```
```

Rotate matrix by 90 degrees

```
```

Kth smallest element in a row-column wise sorted matrix

```
```

Common elements in all rows of a given matrix

```
```

Searching & Sorting

Bubble Sort

```
def bubble_sort(array):
    n=len(array)
    for i in range(n):
        for j in range(n-i-1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]


array=[5,2,3,1,4, -99, 0]
bubble_sort(array)
print(array)
```

Selection Sort

```
def selection_sort(array):
    global iterations
    iterations = 0
    for i in range(len(array)):
        minimum_index = i
        for j in range(i + 1, len(array)):
            iterations += 1
```

```
            if array[minimum_index] > array[j]:
                minimum_index = j

        # Swap the found minimum element with  the first element
        if minimum_index != i:
            array[i], array[minimum_index] = array[minimum_index], array[i]

array=[5,2,3,1,4, -99, 0]
selection_sort(array)
print(array)
```

Insertion Sort

```
def insertion_sort(array):
    global iterations
    iterations = 0
    for i in range(1, len(array)):
        current_value = array[i]
        for j in range(i - 1, -1, -1):
            iterations += 1
            if array[j] > current_value:
                array[j], array[j + 1] = array[j + 1], array[j] # swap
            else:
                array[j + 1] = current_value
                break

array=[5,2,3,1,4, -99, 0]
insertion_sort(array)
print(array)
```

Merge Sort

```
def merge_sort(array):
    if len(array) < 2:
        return array
    mid = len(array) // 2
    left = merge_sort(array[:mid])
    right = merge_sort(array[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) or j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
        if i == len(left) or j == len(right):
            result.extend(left[i:] or right[j:])
            break
    return result

array=[5,2,3,1,4, -99, 0]
print(merge_sort(array))
```

Quick Sort

```
def partition(array, low, high):
    i = low - 1              # index of smaller element
    pivot = array[high]      # pivot

    for j in range(low, high):
        # If current element is smaller than the pivot

        if array[j] < pivot:
        # increment index of smaller element

            i += 1
            array[i], array[j] = array[j], array[i]

    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1

def quick_sort(array, low, high):
    if low < high:
        # pi is partitioning index, arr[p] is now at right place
        temp = partition(array, low, high)

        # Separately sort elements before partition and after partition
        quick_sort(array, low, temp - 1)
        quick_sort(array, temp + 1, high)

array=[5,2,3,1,4, -99, 0]
quick_sort(array, 0, len(array)-1)
print(array)
```

Counting Sort

```
# Counting sort in Python programming


def countingSort(array):
    size = len(array)
    output = [0] * size

    # Initialize count array
    count = [0] * 10

    # Store the count of each elements in count array
    for i in range(size):
        count[array[i]] += 1

    # Store the cummulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Find the index of each element of the original array in count array
    # place the elements in output array
    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1
```

```
    # Copy the sorted elements into original array
    for i in range(size):
        array[i] = output[i]


array = [4,0,2, 2, 8, 3, 3, 1]
countingSort(array)
print(array)
```

Heap Sort

```
def heapify(nums, heap_size, root_index):
    # Assume the index of the largest element is the root index
    largest = root_index
    left_child = (2 * root_index) + 1
    right_child = (2 * root_index) + 2

    if left_child < heap_size and nums[left_child] > nums[largest]:
        largest = left_child

    if right_child < heap_size and nums[right_child] > nums[largest]:
        largest = right_child

    if largest != root_index:
        nums[root_index], nums[largest] = nums[largest], nums[root_index]
        # Heapify the new root element to ensure it's the largest
        heapify(nums, heap_size, largest)


def heap_sort(nums):
    n = len(nums)

    for i in range(n, -1, -1):
        heapify(nums, n, i)

    # Move the root of the max heap to the end of
    for i in range(n - 1, 0, -1):
        nums[i], nums[0] = nums[0], nums[i]
        heapify(nums, i, 0)

random_list_of_nums = [35, 12, 43, 8, 51]
heap_sort(random_list_of_nums)
print(random_list_of_nums)
```

Radix Sort

```
from math import log10
from random import randint

def get_num(num, base, pos):
  return (num // base ** pos) % base

def prefix_sum(array):
  for i in range(1, len(array)):
    array[i] = array[i] + array[i-1]
  return array
```

```
def radixsort(l, base=10):
  passes = int(log10(max(l))+1)
  output = [0] * len(l)

  for pos in range(passes):
    count = [0] * base

    for i in l:
      digit = get_num(i, base, pos)
      count[digit] +=1

    count = prefix_sum(count)

    for i in reversed(l):
      digit = get_num(i, base, pos)
      count[digit] -= 1
      new_pos = count[digit]
      output[new_pos] = i

    l = list(output)
  return output

l = [randint(1, 99999) for _ in range(100)]
sortedarr = radixsort(l)
print(sortedarr)
```

Linear Search

```
def linearSearch(array, n, x):
    for i in range(n):
        if (array[i] == x):
            return i
    return -1

array = [2, 4, 0, 1, 9]
x = 1
n = len(array)
result = linearSearch(array, n, x)
if(result == -1):
    print("Element not found")
else:
    print("Element found at index: ", result)
```

Binary Search

```
def binarySearch(array, x, low, high):
    while low <= high:
        mid = low + (high - low)//2
        if array[mid] == x:
            return mid
        elif array[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1

array = [3, 4, 5, 6, 7, 8, 9]
x = 4
```

```
result = binarySearch(array, x, 0, len(array)-1)
if result != -1:
    print(f"Element is present at index {str(result)}")
else:
    print("Not found")
```

Interpolation Search

```
# Function to determine if target exists in the sorted list `A` or not
# using an interpolation search algorithm
def interpolationSearch(A, target):
    if not A:
        return -1
    (left, right) = (0, len(A) - 1)
    while A[right] != A[left] and A[left] <= target <= A[right]:
        mid = left + (target - A[left]) * (right - left) // (A[right] - A[left])
        if target == A[mid]:
            return mid
        elif target < A[mid]:
            right = mid - 1
        else:
            left = mid + 1
    if target == A[left]:
        return left
    return -1

A = [2, 5, 6, 8, 9, 10]
key = 5
index = interpolationSearch(A, key)
if index != -1:
    print('Element found at index', index)
else:
    print('Element found not in the list')
```

Find first and last positions of an element in a sorted array

```
def first(arr, x, n):
    low = 0
    high = n - 1
    res = -1

    while (low <= high):
        mid = (low + high) // 2
        if arr[mid] > x:
            high = mid - 1
        elif arr[mid] < x:
            low = mid + 1
        else:
            res = mid
            high = mid - 1

    return res

# If x is present in arr[] then returns the index of FIRST occurrence of x in arr[0..n-1],
otherwise returns -1
def last(arr, x, n):
    low = 0
```

```
    high = n - 1
    res = -1
    while(low <= high):
        mid = (low + high) // 2
        if arr[mid] > x:
            high = mid - 1
        elif arr[mid] < x:
            low = mid + 1
        else:
            res = mid
            low = mid + 1
    return res

arr = [ 1, 2, 2, 2, 2, 3, 4, 7, 8, 8 ]
n = len(arr)
x = 8
print("First Occurrence =", first(arr, x, n))
print("Last Occurrence =", last(arr, x, n))
```

Find a Fixed Point (Value equal to index) in a given array

```
"""
  Input: arr[] = {-10, -5, 0, 3, 7}
  Output: 3  // arr[3] == 3

  Input: arr[] = {0, 2, 5, 8, 17}
  Output: 0  // arr[0] == 0


  Input: arr[] = {-10, -5, 3, 4, 7, 9}
  Output: -1  // No Fixed Point
"""

def binarySearch(arr, low, high):
    if high >= low :

        mid = low + (high - low)//2
        if mid == arr[mid]:
            return mid
        res = -1
        if mid + 1 <= arr[high]:
            res = binarySearch(arr, (mid + 1), high)
        if res !=-1:
            return res
        if mid-1 >= arr[low]:
            return binarySearch(arr, low, (mid -1))
    return -1

arr = [-10, -1, 0, 3, 10, 11, 30, 50, 100] # NOTE: ARRAY WILL BE SORTED
n = len(arr)
print(f"Fixed Point is {str(binarySearch(arr, 0, n-1))}")
```

Search in a rotated sorted array

```
def search(nums, target):
    low, high = 0, len(nums)-1
    while low<=high:
```

```
        mid = low + ((high - low))//2
        if nums[mid]==target:
            return mid
        elif nums[low] <= nums[mid]:
            if nums[low] <= target < nums[mid]:
                high = mid-1
            else:
                low = mid+1
        elif nums[mid] < target <= nums[high]:
            low = mid+1
        else:
            high = mid-1
    return -1

target=5
nums=[5,6,1,2,3,4]
print(target, "found at index: ",search(nums,target))
```

square root of an integer

```
def floorSqrt(x):
    if x in [0, 1]:
        return x
    i = 1
    result = 1
    while (result <= x):
        i += 1
        result = i**2
    return i - 1
x = 11
print(floorSqrt(x))
```

Find the repeating and the missing

```
def missandrepeat():
    arr = [ 4, 3, 6, 2, 1, 1 ]
    numberMap = {}
    max = len(arr)
    for i in arr:
        if i not in numberMap:
            numberMap[i] = True
        else:
            print("Repeating =", i)
    for i in range(1, max + 1):
        if i not in numberMap:
            print("Missing =", i)
missandrepeat()
```

Searching in an array where adjacent differ by at most k

```
"""
Input : arr[] = {4, 5, 6, 7, 6}
           k = 1
           x = 6
Output : 2
The first index of 6 is 2.
```

```
Input : arr[] = {20, 40, 50, 70, 70, 60}
          k = 20
          x = 60
Output : 5
The index of 60 is 5
"""

def search(arr, n, x, k):
    # Traverse the given array starting from leftmost element
    i = 0
    while (i < n):
        if (arr[i] == x):
            return i
        # Jump the difference between current array element and x divided by k
        # We use max here to make sure that i moves at-least one step ahead.
        i += max(1, int(abs(arr[i] - x) / k))
    print("number is not present!")
    return -1

arr = [2, 4, 5, 7, 7, 6]
x = 6
k = 2
n = len(arr)
print("Element", x, "is present at index",search(arr, n, x, k))
```

find a pair with a given difference

```
"""
Input : arr[] = {4, 5, 6, 7, 6}
           k = 1
           x = 6
Output : 2
The first index of 6 is 2.

Input : arr[] = {20, 40, 50, 70, 70, 60}
          k = 20
          x = 60
Output : 5
The index of 60 is 5
"""
def findPair(arr,n):
    size = len(arr)
    i,j = 0,1
    while i < size and j < size:
        if i != j and arr[j]-arr[i] == n:
            print (f"Pair found ({arr[i]} ,{arr[j]})")
            return True

        elif arr[j] - arr[i] < n:
            j+=1
        else:
            i+=1
    print ("No pair found")
    return False

arr = [1, 8, 30, 40, 100]
n = 60
findPair(arr, n)
```

find two elements that sum to a given value - TwoSum

```
def findPair(nums, target):
    d = {}
    for i, e in enumerate(nums):
        if target - e in d:
            print('Pair found', (nums[d.get(target - e)], nums[i]))
            return
        d[e] = i
    print('Pair not found')

nums = [8, 7, 2, 5, 3, 1]
target = 10
findPair(nums, target)
```

find four elements that sum to a given value - ThreeSum

```
def isTripletExist(nums, target):
    d = {e: i for i, e in enumerate(nums)}
    for i in range(len(nums) - 1):
        for j in range(i + 1, len(nums)):
            val = target - (nums[i] + nums[j])
            if val in d and d[val] not in [i, j]:
                return True
    return False

nums = [2, 7, 4, 0, 9, 5, 1, 3]
target = 6
if isTripletExist(nums, target):
    print('Triplet exists')
else:
    print('Triplet doesn\'t exist')
```

find four elements that sum to a given value - - FourSum

```
def hasQuadruplet(nums, target):
    # create an empty dictionary
    # key -> target of a pair in the list
    # value -> list storing an index of every pair having that sum
    d = {}
    for i in range(len(nums) - 1):
        for j in range(i + 1, len(nums)):
            val = target - (nums[i] + nums[j])
            if val in d:
                for pair in d[val]:
                    x, y = pair
                    if x not in [i, j] and y not in [i, j]:
                        print('Quadruplet Found', (nums[i], nums[j], nums[x], nums[y]))
                        return True
            d.setdefault(nums[i] + nums[j], []).append((i, j))
    return False

nums = [2, 7, 4, 0, 9, 5, 1, 3]
target = 20
if not hasQuadruplet(nums, target):
    print('Quadruplet doesn\'t exist')
```

maximum sum such that no 2 elements are adjacent

```
"""
Input: arr[] = {5, 5, 10, 100, 10, 5}
Output: 110
Explanation: Pick the subsequence {5, 100, 5}.
The sum is 110 and no two elements are adjacent. This is the highest possible sum.

Input: arr[] = {3, 2, 7, 10}
Output: 13
Explanation: The subsequence is {3, 10}. This gives sum = 13.
This is the highest possible sum of a subsequence following the given criteria

Input: arr[] = {3, 2, 5, 10, 7}
Output: 15
Explanation: Pick the subsequence {3, 5, 7}. The sum is 15.
"""

def findMaxSum(arr, n):
    incl = 0
    excl = 0
    for i in arr:
        new_excl = max (excl, incl)
        incl = excl + i
        excl = new_excl
    return max(excl, incl)

arr = [5, 5, 10, 100, 10, 5]
N = 6
print (findMaxSum(arr, N))
```

Count triplet with sum smaller than a given value

```
def countTriplets(arr,n,sum):
    arr.sort()
    ans = 0
    # Every iteration of loop counts triplet with first element as arr[i].
    for i in range(n-2):

        # Initialize other two elements as corner elements of subarray arr[j+1..k]
        j = i + 1
        k = n-1

        while(j < k):
            # If sum of current triplet is more or equal, move right corner to look for smaller
values
            if (arr[i]+arr[j]+arr[k] >=sum):
                k = k-1
            # Else move left corner
            else:
                # This is important. For current i and j, there can be total k-j third elements.
                ans += (k - j)
                j = j+1
    return ans

arr = [5, 1, 3, 4, 7]
```

```
n = len(arr)
target = 12
print(countTriplets(arr, n, target))
```

print all subarrays with 0 sum

```
def findSubArrays(arr,n):
    # create a python dict
    hashMap = {}
    # create a python list equivalent to ArrayList
    out = []
    # tracker for sum of elements
    sum1 = 0
    for i in range(n):
        sum1 += arr[i]
        if sum1 == 0:
            out.append((0, i))
        al = []
        if sum1 in hashMap:
            al = hashMap.get(sum1)
            for it in range(len(al)):
                out.append((al[it] + 1, i))
        al.append(i)
        hashMap[sum1] = al
    return out

def printOutput(output):
    for i in output:
        print(f"Subarray found from Index {str(i[0])} to {str(i[1])}")

arr = [6, 3, -1, -3, 4, -2,
        2, 4, 6, -12, -7]
n = len(arr)
out = findSubArrays(arr, n)
# if we did not find any subarray with 0 sum, then subarray does not exists
if (len(out) == 0):
    print ("No subarray exists")
else:
    printOutput (out)
```

Product array Puzzle

```
"""
Given an array arr[] of n integers, construct a Product Array prod[] (of same size) such that
prod[i] is equal to the product of all the elements of arr[] except arr[i]. Solve it without
division operator and in O(n).
Example:

Input: arr[] = {10, 3, 5, 6, 2}
Output: prod[] = {180, 600, 360, 300, 900}
The elements of output array are
{3*5*6*2, 10*5*6*2, 10*3*6*2,
10*3*5*2, 10*3*5*6}

Input: arr[] = {1, 2, 1, 3, 4}
Output: prod[] = {24, 12, 24, 8, 6}
The elements of output array are
{3*4*1*2, 1*1*3*4, 4*3*2*1,
```

```
1*1*4*2, 1*1*3*2}
"""
def solve(arr, n):
    # Initialize a variable to store the total product of the array elements
    prod = 1
    for i in arr:
        prod *= i

    # we know x / y mathematically is same as x*(y to power -1)
    for i in arr:
        print(int(prod*(i**-1)), end =" ")

arr = [10, 3, 5, 6, 2]
n = len(arr)
solve(arr, n)
```

Sort array according to count of set bits

```
"""
Input: arr[] = {5, 2, 3, 9, 4, 6, 7, 15, 32};
Output: 15 7 5 3 9 6 2 4 32
Explanation:
The integers in their binary representation are:
    15 -1111
    7  -0111
    5  -0101
    3  -0011
    9  -1001
    6  -0110
    2  -0010
    4- -0100
    32 -10000
hence the non-increasing sorted order is:
{15}, {7}, {5, 3, 9, 6}, {2, 4, 32}
"""

def countSetBits(val):
    cnt = 0
    while val:
        cnt += val % 2
        val = val//2
    return cnt

# Using custom comparator lambda function
arr = [1, 2, 3, 4, 5, 6]

# form a tuple with val, index
n = len(arr)
arr = [(arr[i], i) for i in range(n)]

# first criteria to sort is number of set bits, then the index
sorted_arr = sorted(arr, key=lambda val: (
    countSetBits(val[0]), n-val[1]), reverse=True)
sorted_arr = [val[0] for val in sorted_arr]
print(sorted_arr)
```

minimum no. of swaps required to sort the array

```
"""
Given an array of n distinct elements, find the minimum number of swaps required to sort the
array.
Input: {4, 3, 2, 1}
Output: 2
Explanation: Swap index 0 with 3 and 1 with 2 to
              form the sorted array {1, 2, 3, 4}.
"""

def minSwaps(arr, N):
    ans = 0
    temp = arr.copy()
    temp.sort()
    for i in range(N):
        # This is checking whether the current element is at the right place or not
        if (arr[i] != temp[i]):
            ans += 1
            # Swap the current element with the right index so that arr[0] to arr[i] is sorted
            swap(arr, i,
                indexOf(arr, temp[i]))
    return ans

def swap(arr, i, j):
    arr[i], arr[j]= arr[j], arr[i]


def indexOf(arr, ele):
    for i in range(len(arr)):
        if (arr[i] == ele):
                return i
    return -1

a = [101, 758, 315, 730, 472, 619, 460, 479]
n = len(a)
print(minSwaps(a, n))
```

Find pivot element in a sorted array

```
def findPivot(arr, left, right):
    if right< left:
        return -1
    if right == left:
        return left
    mid = (left+right)//2
    if mid<right and arr[mid]>arr[mid+1]:
        return mid
    if mid>left and arr[mid]<arr[mid-1]:
        return mid-1
    if arr[left]<arr[mid]:
        return findPivot(arr, mid+1, right)
    else:
        return findPivot(arr, left, mid-1)

arr=[14, 23, 7, 9, 3, 6, 18, 22, 16, 36]
start=0
n=len(arr)-1
pivot=findPivot(arr, start, n)
pivot+=1 # pivot is the index of the first element in the right subarray
print(f'Pivot is {arr[pivot]}')
```

K-th Element of Two Sorted Arrays

```
"""Given two sorted arrays of size m and n respectively, you are tasked with finding the element
that would be at the k’th position of the final sorted array.

Examples:

Input : Array 1 - 2 3 6 7 9
        Array 2 - 1 4 8 10
        k = 5
Output : 6
Explanation: The final sorted array would be -
1, 2, 3, 4, 6, 7, 8, 9, 10
The 5th element of this array is 6.
"""

def find(A, B, m, n, k_req):
    i, j, k = 0, 0, 0
    # Keep taking smaller of the current elements of two sorted arrays and keep incrementing k
    while i < len(A) and j < len(B):
        k += 1
        if A[i] < B[j]:
            if k == k_req:
                return A[i]
            i += 1
        elif k == k_req:
            return B[j]
        else:
            j += 1

    # If array B[] is completely traversed
    while i < len(A):
        k += 1
        if k == k_req:
                return A[i]
        i += 1

    # If array A[] is completely traversed
    while j < len(B):
        k += 1
        if k == k_req:
                return B[j]
        j += 1

A = [2, 3, 6, 7, 9]
B = [1, 4, 8, 10]
k = 5;
print(find(A, B, 5, 4, k))
```

Aggressive cows

```
"""
Problem Statement: There is a new barn with N stalls and C cows. The stalls are located on a
straight line at positions x1,….,xN (0 <= xi <= 1,000,000,000). We want to assign the cows to the
stalls, such that the minimum distance between any two of them is as large as possible. What is
the largest minimum distance?
```

```
Examples:

Input: No of stalls = 5
       Array: {1,2,8,4,9}
       And number of cows: 3

Output: One integer, the largest minimum distance 3
"""

def isPossible(a, n, cows, mid):
    CntCows = 1
    lastPlacedCow=a[0]
    for i in range(1, n):
        if a[i] -lastPlacedCow >= mid:
            CntCows+=1
            lastPlacedCow=a[i]
    return CntCows >= cows

n = 5
cows = 3;
a=[1,2,8,4,9]
a.sort()
low = 1
high = a[n - 1] - a[0]
while low< high:
    mid = (low + high)//2
    if isPossible(a, n, cows, mid):
        low=mid+1
    else:
        high=mid-1
print(f'Largest minimum distance is: {high}')
```

Book Allocation Problem

```
"""
Given number of pages in n different books and m students. The books are arranged in ascending
order of number of pages. Every student is assigned to read some consecutive books. The task is to
assign books in such a way that the maximum number of pages assigned to a student is minimum.
Example :


Input : pages[] = {12, 34, 67, 90} , m = 2
Output : 113
Explanation:
There are 2 number of students. Books can be distributed
in following fashion :
  1) [12] and [34, 67, 90]
      Max number of pages is allocated to student
      '2' with 34 + 67 + 90 = 191 pages
  2) [12, 34] and [67, 90]
      Max number of pages is allocated to student
      '2' with 67 + 90 = 157 pages
  3) [12, 34, 67] and [90]
      Max number of pages is allocated to student
      '1' with 12 + 34 + 67 = 113 pages

Of the 3 cases, Option 3 has the minimum pages = 113.
"""
```

```python
def isPossible(arr, n, m, curr_min):
    studentsRequired = 1
    curr_sum = 0
    for i in range(n):

        # check if current number of pages are greater than curr_min that means we will get the
result after mid no. of pages
        if (arr[i] > curr_min):
            return False

        # count how many students are required to distribute curr_min pages
        if (curr_sum + arr[i] > curr_min):

            # increment student count
            studentsRequired += 1

            # update curr_sum
            curr_sum = arr[i]

            # if students required becomes greater than given no. of students, return False
            if (studentsRequired > m):
                return False

        # else update curr_sum
        else:
            curr_sum += arr[i]

    return True

# function to find minimum pages
def findPages(arr, n, m):
    # return -1 if no. of books is less than no. of students
    if (n < m):
        return -1

    mysum = sum(arr[i] for i in range(n))
    # initialize start as 0 pages and end as total pages
    start, end = 0, mysum
    result = 10**9

    # traverse until start <= end
    while (start <= end):

        # check if it is possible to distribute books by using mid as current minimum
        mid = (start + end) // 2
        if (isPossible(arr, n, m, mid)):

            # update result to current distribution as it's the best we have found till now.
            result = mid

            # as we are finding minimum and books are sorted so reduce end = mid -1 that means
            end = mid - 1

        else:
            # if not possible means pages should be increased so update start = mid + 1
            start = mid + 1

    # at-last return minimum no. of pages
    return result
```

```
arr = [12, 34, 67, 90]  # Number of pages in books
n = len(arr)
m = 2 # No. of students
print("Minimum number of pages = ",findPages(arr, n, m))
```

EKOSPOJ:

```
TODO
```

Missing Number in AP

```
"""
Given an array that represents elements of arithmetic progression in order. One element is missing
in the progression, find the missing number.

Examples:

Input: arr[]  = {2, 4, 8, 10, 12, 14}
Output: 6

Input: arr[]  = {1, 6, 11, 16, 21, 31};
Output: 26
"""

def find_missing(arr, n):
    first = arr[0]
    last = arr[-1]

    if (first + last) % 2:
        s = (n + 1) / 2
        s *= (first + last)
    else:
        s = (first + last) / 2
        s *= (n + 1)

    return s - sum(arr)

arr = [2, 4, 8, 10, 12, 14]
n = len(arr)
missing = find_missing(arr, n)
print(missing)
```

Smallest number with atleastn trailing zeroes infactorial

```
"""
Given a number n. The task is to find the smallest number whose factorial contains at least n
trailing zeroes.
Examples :


Input : n = 1
Output : 5
1!, 2!, 3!, 4! does not contain trailing zero.
5! = 120, which contains one trailing zero.

Input : n = 6
Output : 25
```

```
"""

def check(p,n):
    temp = p
    count = 0
    f = 5
    while (f <= temp):
        count += temp//f
        f *= 5
    return (count >= n)

# Return smallest number whose factorial contains at least n trailing zeroes
def findNum(n):
    # If n equal to 1, return 5. since 5! = 120.
    if (n==1):
        return 5
    # Initializing low and high for binary search.
    low = 0
    high = 5*n

    while (low <high):
        mid = (low + high) >> 1
        # Checking if mid's factorial contains n trailing zeroes.
        if (check(mid, n)):
            high = mid
        else:
            low = mid+1
    return low

n = 6
print(findNum(n))
```

ROTI-Prata SPOJ

```
TODO
```

DoubleHelix SPOJ

```
TODO
```

Subset Sums

```
"""
Given an array of integers, print sums of all subsets in it. Output sums can be printed in any
order.

Examples :
Input : arr[] = {2, 3}
Output: 0 2 3 5

Input : arr[] = {2, 4, 5}
Output : 0 2 4 5 6 7 9 11
"""

def subsetSums(arr, l, r, sum=0):
    # Print current subset
    if l > r:
```

```
        print(sum, end=" ")
        return
    # Subset including arr[l]
    subsetSums(arr, l + 1, r, sum + arr[l])
    # Subset excluding arr[l]
    subsetSums(arr, l + 1, r, sum)

arr = [5, 4, 3]
n = len(arr)
subsetSums(arr, 0, n - 1)
```

Implement Merge-sort in-place

```
"""
NOTE: TIME COMPLEXITY IS HIGHER THAN STANDARD MERGE SORT BUT SPACE COMPLEXITY IS O(1)
"""
def merge(arr, start, mid, end):
    start2 = mid + 1
    # If the direct merge is already sorted
    if (arr[mid] <= arr[start2]):
        return
    # Two pointers to maintain start of both arrays to merge
    while (start <= mid and start2 <= end):
        # If element 1 is in right place
        if arr[start] > arr[start2]:
            value = arr[start2]
            index = start2
            # Shift all the elements between element 1 element 2, right by 1.
            while (index != start):
                arr[index] = arr[index - 1]
                index -= 1
            arr[start] = value
            mid += 1
            start2 += 1
        start += 1


def mergeSort(arr, l, r):
    if (l < r):
        # Same as (l + r) / 2, but avoids overflow for large l and r
        m = l + (r - l) // 2
        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)

arr = [12, 11, 13, 5, 6, 7]
arr_size = len(arr)
mergeSort(arr, 0, arr_size - 1)
print(arr)
```

Stacks & Queues

Implement Stack from Scratch

```
```

Implement Queue from Scratch

Implement 2 stack in an array

find the middle element of a stack

Implement "N" stacks in an Array

Check the expression has valid or Balanced parenthesis or not.

Reverse a String using Stack

Design a Stack that supports getMin() in O(1) time and O(1) extra space.

Find the next Greater element

The celebrity Problem

Arithmetic Expression evaluation

Evaluation of Postfix expression

Implement a method to insert an element at its bottom without using any other data structure.

Reverse a stack using recursion

Sort a Stack using recursion

Merge Overlapping Intervals

```
"""
Consider an event where a log register is maintained containing the guest's arrival and departure
times. Given an array of arrival and departure times from entries in the log register, find the
point when there were maximum guests present in the event.

Note that if an arrival and departure event coincides, the arrival time is preferred over the
departure time.


For example,

Input:

arrival = { 1, 2, 4, 7, 8, 12 }
departure = { 2, 7, 8, 12, 10, 15 }

Output: Maximum number of guests is 3, present at time 7
"""

def findMaxGuests(arrival, departure):

    # Find the time when the last guest leaves the event
    t = max(departure)

    # create a count array initialized by 0
    count = [0] * (t + 2)

    for i in range(len(arrival)):
        count[arrival[i]] += 1
        count[departure[i] + 1] -= 1

    # keep track of the time when there are maximum guests
    max_event_tm = count[0]

    # perform a prefix sum computation to determine the guest count at each point
    for i in range(1, t + 1):
        count[i] += count[i - 1]
        if count[max_event_tm] < count[i]:
            max_event_tm = i
```

```
    print("Event Time:", max_event_tm)
    print("The maximum number of guests is", count[max_event_tm])


arrival = [1, 2, 4, 7, 8, 12]
departure = [2, 7, 8, 12, 10, 15]
findMaxGuests(arrival, departure)
```

Largest rectangular Area in Histogram

Length of the Longest Valid Substring

Expression contains redundant bracket or not

Implement Stack using Queue

Implement Stack using Deque

Stack Permutations (Check if an array is stack permutation of other)

Implement Queue using Stack

Implement "n" queue in an array

Implement a Circular queue

LRU Cache Implementationa

Reverse a Queue using recursion

Reverse the first "K" elements of a queue

Interleave the first half of the queue with second half

Find the first circular tour that visits all Petrol Pumps

Minimum time required to rot all oranges

Distance of nearest cell having 1 in a binary matrix

First negative integer in every window of size "k"

Check if all levels of two trees are anagrams or not.

Sum of minimum and maximum elements of all subarrays of size "k".

Minimum sum of squares of character counts in a given string after removing "k" characters.

Queue based approach or first non-repeating character in a stream.

Next Smaller Element

String

Reverse a String

Check whether a String is Palindrome or not

Find Duplicate characters in a string

Why strings are immutable in Java?

Write a Code to check whether one string is a rotation of another

Write a Program to check whether a string is a valid shuffle of two strings or not

Count and Say problem

Write a program to find the longest Palindrome in a string.[Longest palindromic Substring]

Find Longest Recurring Subsequence in String

Print all Subsequences of a string.

Print all the permutations of the given string

Split the Binary string into two substring with equal 0’s and 1’s

Find next greater number with same set of digits. [Very Very IMP]

Balanced Parenthesis problem.[Imp]

Rabin Karp Algo

KMP Algo

Convert a Sentence into its equivalent mobile numeric keypad sequence.

Minimum number of bracket reversals needed to make an expression balanced.

Count All Palindromic Subsequence in a given String.

Count of number of given string in 2D character array

Search a Word in a 2D Grid of characters.

Boyer Moore Algorithm for Pattern Searching.

Converting Roman Numerals to Decimal

Longest Common Prefix

Number of flips to make binary string alternate

Find the first repeated word in string.

Minimum number of swaps for bracket balancing.

Find the longest common subsequence between two strings.

Program to generate all possible valid IP addresses from given string.

Write a program to find the smallest window that contains all characters of string itself.

Rearrange characters in a string such that no two adjacent are same

Minimum characters to be added at front to make string palindrome

Given a sequence of words, print all anagrams together

Find the smallest window in a string containing all characters of another string

Recursively remove all adjacent duplicates

String matching where one string contains wildcard characters

Function to find Number of customers who could not get a computer

Transform One String to Another using Minimum Number of Given Operation

Check if two given strings are isomorphic to each other

Recursively print all sentences that can be formed from list of word lists

Trie

Construct a trie from scratch

```
class TrieNode:
```

```python
    def __init__(self):
        self.children = [None]*26
        # isEndOfWord is True if node represent the end of the word
        self.isEndOfWord = False

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

    def getNode(self):
        # Returns new trie node (initialized to NULLs)
        return TrieNode()

    def _charToIndex(self,ch):
        # private helper function.
        # Converts key current character into index use only 'a' through 'z' and lower case
        return ord(ch)-ord('a')


    def insert(self,key):

        # If not present, inserts key into trie
        # If the key is prefix of trie node, just marks leaf node
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])

            # if current character is not present
            if not pCrawl.children[index]:
                pCrawl.children[index] = self.getNode()
            pCrawl = pCrawl.children[index]

        # mark last node as leaf
        pCrawl.isEndOfWord = True

    def search(self, key):

        # Search key in the trie
        # Returns true if key presents in trie, else false
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])
            if not pCrawl.children[index]:
                return False
            pCrawl = pCrawl.children[index]

        return pCrawl.isEndOfWord


def constructTrie():

    # Input keys (use only 'a' through 'z' and lower case)
    keys = ["the","a","there","answer","any", "by","their", "these"]
    output = ["Not present in trie", "Present in trie"]

    # Trie object
    t = Trie()
```

```
    # Construct trie
    for key in keys:
        t.insert(key)

    # Search for different keys
    print("the->",output[t.search("the")])
    print("these->",output[t.search("these")])
    print("their->",output[t.search("their")])
    print("thaw->",output[t.search("thaw")])

if __name__ == '__main__':
    constructTrie()
```

Find shortest unique prefix for every word in a given list

```
"""
Input:  [AND, BONFIRE, BOOL, CASE, CATCH, CHAR]
Output: [A, BON, BOO, CAS, CAT, CH]
Explanation:
A can uniquely identify AND
BON can uniquely identify BONFIRE
BOO can uniquely identify BOOL
CAS can uniquely identify CASE
CAT can uniquely identify CATCH
CH can uniquely identify CHAR
"""
# A class to store a Trie node
class TrieNode:
    def __init__(self):
        # each node stores a dictionary to its child nodes
        self.child = {}

        # keep track of the total number of times the current node is visited
        # while inserting data in Trie
        self.freq = 0


# Function to insert a given string into a Trie
def insert(root, word):

    # start from the root node
    curr = root
    for c in word:
        # create a new node if the path doesn't exist
        curr.child.setdefault(c, TrieNode())

        # increment frequency
        curr.child[c].freq += 1

        # go to the next node
        curr = curr.child[c]


# Function to recursively traverse the Trie in a preorder fashion and
# print the shortest unique prefix for each word in the Trie
def printShortestPrefix(root, word_so_far):
    if root is None:
        return
```

```
    # print `word_so_far` if the current Trie node is visited only once
    if root.freq == 1:
        print(word_so_far)
        return

    # recur for all child nodes
    for k, v in root.child.items():
        printShortestPrefix(v, word_so_far + k)


# Find the shortest unique prefix for every word in a given array
def findShortestPrefix(words):

    # construct a Trie from the given items
    root = TrieNode()
    for s in words:
        insert(root, s)

    # Recursively traverse the Trie in a preorder fashion to list all prefixes
    printShortestPrefix(root, '')


if __name__ == '__main__':
    words = ['AND', 'BONFIRE', 'BOOL', 'CASE', 'CATCH', 'CHAR']
    findShortestPrefix(words)
```

Word Break Problem | (Trie solution)

```
# Currently, Trie supports lowercase English characters. So, the character size is 26.
CHAR_SIZE = 26


# A class to store a Trie node
class Node:
    next = [None] * CHAR_SIZE
    exist = False       # true when the node is a leaf node


# Iterative function to insert a string into a Trie
def insertTrie(head, s):

    # start from the root node
    node = head

    # do for each character in the string
    for c in s:

        index = ord(c) - ord('a')

        # create a new node if the path doesn't exist
        if node.next[index] is None:
            node.next[index] = Node()

        # go to the next node
        node = node.next[index]

    # mark the last node as a leaf
    node.exist = True
```

```
# Function to determine if a string can be segmented into space-separated
# sequence of one or more dictionary words
def wordBreak(head, s):

    # get the length of the string
    n = len(s)

    # `good[i]` is true if the first `i` characters of `s` can be segmented
    good = [None] * (n + 1)
    good[0] = True      # base case

    for i in range(n):
        if good[i]:
            node = head
            for j in range(i, n):
                if node is None:
                    break

                index = ord(s[j]) - ord('a')
                node = node.next[index]

                # we can make [0, i] using our known decomposition
                # and [i+1, j] using this string in a Trie
                if node and node.exist:
                    good[j + 1] = True

    # `good[n]` would be true if all characters of `s` can be segmented
    return good[n]


if __name__ == '__main__':

    # List of strings to represent a dictionary
    words = ['self', 'th', 'is', 'famous', 'word', 'break', 'b', 'r', 'e', 'a', 'k', 'br', 'bre',
'brea', 'ak', 'prob', 'lem']

    # given string
    s = 'wordbreakproblem'

    # create a Trie to store the dictionary
    t = Node()
    for word in words:
        insertTrie(t, word)

    # check if the string can be segmented or not
    if wordBreak(t, s):
        print('The string can be segmented')
    else:
        print('The string can\'t be segmented')
```

Given a sequence of words, print all anagrams together

```
class TrieNode:
    def __init__(self):
        # each node stores a dictionary to its child nodes
        self.child = {}

        # stores anagrams in the leaf node
```

```
        self.words = []


# Function to insert a string into a Trie
def insert(root, word, originalWord):

    # start from the root node
    curr = root
    for c in word:
        # create a new node if the path doesn't exist
        curr.child.setdefault(c, TrieNode())
        # go to the next node
        curr = curr.child[c]

    # anagrams will end up at the same leaf node
    curr.words.append(originalWord)


# A recursive function that traverses a Trie in preorder fashion and
# prints all anagrams together
def printAnagrams(root):

    # base case
    if root is None:
        return

    # print the current word
    if len(root.words) > 1:
        print(root.words)

    # recur for all child nodes
    for child in root.child.values():
        printAnagrams(child)


# Function to group anagrams from a given list of words
def groupAnagrams(words):

    # construct an empty trie
    root = TrieNode()

    # do for each word
    for word in words:
        # Sort the characters of the current word and insert it into the Trie.
        # Note that the original word gets stored on the leaf
        insert(root, ''.join(sorted(word)), word)

    # print all anagrams together
    printAnagrams(root)


if __name__ == '__main__':

    words = ['auctioned', 'actors', 'altered', 'streaming', 'related', 'education', 'aspired',
'costar', 'despair', 'mastering', 'act','cat','tac'
    ]
    groupAnagrams(words)
```

Print unique rows in a given boolean matrix

```python
# Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
ROW = 4
COL = 5

def findUniqueRows(M):

    # Traverse through the matrix
    for i in range(ROW):
        flag = 0

        # Check if there is similar column is already printed, i.e if i and jth column match.
        for j in range(i):
            flag = 1

            for k in range(COL):
                if (M[i][k] != M[j][k]):
                    flag = 0

            if (flag == 1):
                break

        # If no row is similar
        if (flag == 0):

            # Print the row
            for j in range(COL):
                print(M[i][j], end = " ")
            print()


M = [ [ 0, 1, 0, 0, 1 ],
      [ 1, 0, 1, 1, 0 ],
      [ 0, 1, 0, 0, 1 ],
      [ 1, 0, 1, 0, 0 ] ]
findUniqueRows(M)
```