1. Deep Copy of linked list with random pointers

```
class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        node = head
        while node:
            node.next = Node(x=node.val, next=node.next)
            node = node.next.next

        node = head
        while node:
            if node.random:
                node.next.random = node.random.next
            node = node.next.next

        node = head
        copy = copyhead = Node(0)
        while node:
            copy.next = copy = node.next
            node.next = node = node.next.next

        return copyhead.next
```

https://leetcode.com/problems/copy-list-with-random-pointer/discuss/578598/iterative-%2B-recursive-%2B-hashmap-python

---

2. Merge two sorted linked lists:

```
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        head = ListNode(None)
        l3 = head
        while l1 and l2:
            head.next = ListNode(min(l1.val,l2.val))
            if l1.val < l2.val:
                l1 = l1.next
            else:
                l2 = l2.next
            head = head.next
        head.next = l1 if l1 else l2
        return l3.next
```

---

3. Subtree of another tree:

```
class Solution:
    def isSubtree(self, s: TreeNode, t: TreeNode) -> bool:
        if s is None or t is None:
            return False
        elif s is None and t is None:
            return True
        elif s.val == t.val and self.inordertraversal(s) ==
self.inordertraversal(t):
            return True
        return self.isSubtree(s.left,t) or
self.isSubtree(s.right,t)

    def inordertraversal(self,root):
        if root is None:
            return []
        else:
            res = []
            res = self.inordertraversal(root.left)
            res.append(root.val)
            res = res + self.inordertraversal(root.right)
            return res
```

---

4. Search 2D matrix

Solution 1:

```
def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        flag = False
        for item in matrix:
            if target in item:
                flag = True
                break
        return flag
```

Solution 2:

```
def searchMatrix(self, matrix: List[List[int]], target:
int) -> bool:
        if not matrix:
            return False
        rows = len(matrix)
        cols = len(matrix[0])
        if not cols:
            return False

        left = 0
        right = rows*cols-1

        while left <= right:
            mid = (left+right)//2
            i = mid //cols
            j = mid % cols

            if matrix[i][j] == target:
                return True
            elif matrix[i][j] < target:
                left = mid + 1
            else:
                right = mid - 1

        return False
```

---

5. Two sum

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
```

```
        """
        h = {}
        for i, num in enumerate(nums):
            n = target - num
            if n not in h:
                h[num] = i
            else:
                return [h[n], i]
```

6. Twosum sorted

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        m = len(numbers) // 2
        m_value = numbers[m]

        if m_value > target and numbers[0]>0:
            return self.twoSum(numbers[:m], target)
        else:
            cache = {}
            for i in range(len(numbers)):
                diff = target - numbers[i]
                if diff in cache:
                    return [cache[diff]+1, i+1]
                cache[numbers[i]] = i
```

7. Favorite song genre

```
def favouriteGenres(userSongs, songGenres):
  songGenre = {}
  output = {}

  # build mapping from song -> genre
  for genre in songGenres:
    songs = songGenres[genre]
    for song in songs:
      songGenre[song] = genre
```

```
  for user in userSongs:
    genreCount = {genre: 0 for genre in songGenres}
    songs = userSongs[user]
    maxCount = float("-inf")
    favSongs = []

    for song in songs:
      genre = songGenre[song]
      genreCount[genre] += 1

      if genreCount[genre] > maxCount:
        maxCount = genreCount[genre]
        favSongs = []
        favSongs.append(genre)

      elif genreCount[genre] == maxCount:
        favSongs.append(genre)
    output[user] = favSongs

  return output
```

---

8. TwoSum Amazon:
Given an int array nums and an int target, find how many unique pairs in the array such that their sum is equal to target. Return the number of pairs.
Input: nums = [1, 1, 2, 45, 46, 46], target = 47
Output: 2

```
def uniqueTwoSum(nums, target):
  ans, comp = set(), set()
  for n in nums:
    c = target-n
    if c in comp:
      res = (n, c) if n > c else (c, n)
```

```
      if res not in ans:
        ans.add(res)
    comp.add(n)
  return len(ans)
```

---

9. Generating spiral matrix

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) ->
List[int]:
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """

        if not matrix:
            return None

        res = []
        i, j = 0, 0
         # d[0]: moving direction along a row; d[1]: moving
direction along a column
        d = [1, 0]
        m, n = len(matrix), len(matrix[0])
         # bounds records the ranges of moves
        bounds = [0, n - 1, 1, m - 1]

         # go through each position only once
        while len(res) < m * n:
            res.append(matrix[i][j])
              # change the direction when reaching a bound
            if d[0] == 1 and j == bounds[1]:
```

```
                d = [0, 1]
                bounds[1] = j - 1
            if d[1] == 1 and i == bounds[3]:
                d = [-1, 0]
                bounds[3] = i - 1
            if d[0] == -1 and j == bounds[0]:
                d = [0, -1]
                bounds[0] = j + 1
            if d[1] == -1 and i == bounds[2]:
                d = [1, 0]
                bounds[2] = i + 1

            i += d[1]
            j += d[0]

        return res
```

---

10. Critical Connections:

```
from collections import defaultdict

class Solution(object):
    def criticalConnections(self, n, connections):
        """
        :type n: int
        :type connections: List[List[int]]
        :rtype: List[List[int]]
        """
        graph = defaultdict(list)
        for v in connections:
            graph[v[0]].append(v[1])
            graph[v[1]].append(v[0])
```

```
        disc = [None for _ in range(n+1)]
        low = [None for _ in range(n+1)]

        res = []
        self.cur = 0

        def dfs(node, parent):
            if disc[node] is None:
                disc[node] = self.cur
                low[node] = self.cur
                self.cur += 1
                for n in graph[node]:
                    if disc[n] is None:
                        dfs(n, node)
                if parent is not None:
                    l = min([low[i] for i in graph[node] if
i!=parent]+[low[node]])
                else:
                    l = min(low[i] for i in graph[node]+
[low[node]])
                low[node] = l

        dfs(1, None)

        for v in connections:
            if low[v[0]]>disc[v[1]] or
low[v[1]]>disc[v[0]]:
                res.append(v)

        return res
```

---

11. Minimum cost to connect all nodes (Min cost to add new roads)

```
#Kruskal algorithm
def compute_min_cost(num_nodes, base_mst, poss_mst):
    uf = {}

    # create union find for the initial edges given
    def find(edge):
        uf.setdefault(edge, edge)
        if uf[edge] != edge:
            uf[edge] = find(uf[edge])
        return uf[edge]

    def union(edge1, edge2):
        uf[find(edge1)] = find(edge2)

    for e1, e2 in base_mst:
        if find(e1) != find(e2):
            union(e1, e2)

    # sort the new edges by cost
    # if an edge is not part of the minimum spanning tree,
then include it, else continue
    cost_ret = 0
    for c1, c2, cost in sorted(poss_mst, key=lambda x :
x[2]):
        if find(c1) != find(c2):
            union(c1, c2)
            cost_ret += cost

    if len({find(c) for c in uf}) == 1 and len(uf) ==
```

```
num_nodes:
        return cost_ret
    else:
        return -1


if __name__ == '__main__':
    n = 6
    edges = [[1, 4], [4, 5], [2, 3]]
    new_edges = [[1, 2, 5], [1, 3, 10], [1, 6, 2], [5, 6,
5]]
    print(compute_min_cost(n, edges, new_edges))
```

---

12. Minimum cost to repair all edges (MST)

Solution 1:

```
from collections import defaultdict
import heapq

class Solution:
   def __init__(self):
      pass

   def minCostForRepair(self, n, edges, edgesToRepair):
      graph=defaultdict(list)
      addedEdges=set()
      for edge in edgesToRepair:
         graph[edge[0]].append((edge[2], edge[1]))
         graph[edge[1]].append((edge[2], edge[0]))
         addedEdges.add((edge[0], edge[1]))
         addedEdges.add((edge[1], edge[0]))
      for edge in edges:
         if tuple(edge) not in addedEdges:
            graph[edge[0]].append((0, edge[1]))
            graph[edge[1]].append((0, edge[0]))
```

```
        res=0
        priorityQueue=[(0, 1)]
        heapq.heapify(priorityQueue)
        visited=set()

        while priorityQueue:
            minCost, node=heapq.heappop(priorityQueue)
            if node not in visited:
                visited.add(node)
                res+=minCost
                for cost, connectedNode in graph[node]:
                    if connectedNode not in visited:
                        heapq.heappush(priorityQueue, (cost, connectedNode))

        return res



s = Solution()
n = 5
edges = [[1, 2], [2, 3], [3, 4], [4, 5], [1, 5]]
edgesToRepair = [[1, 2, 12], [3, 4, 30], [1, 5, 8]]
print(s.minCostForRepair(n, edges, edgesToRepair))
```

Solution 2:

```
def minCostForRepair( n, edges,edgesToRepair):
        edgesToRepair.sort(key= lambda x:x[2])

        # Build hash table with broken edges in order to skip them
        # when processing the edges list
        hm = {(u, v) for u, v, _ in edgesToRepair}

        total_cost = 0

        # every node is a parent of itself
        parent = [i for i in range(n+1)]
        rank = [0 for _ in range(n+1)]

        # finds parent
        def find(p):
                if parent[p] != p:
                        parent[p] = find(parent[p])
                return parent[p]

        # make "a" as parent of "b"
        def union(a, b):
```

```
            pa = find(a)
            pb = find(b)

            # a and b are already in the same set
            if pa == pb:
                return False

            # Otherwise: the roots are distinct, then the trees are combined
            # by attaching the root of one to the root of the other.
            if rank[pa] < rank[pb]:
                # swap parents since union by rank always attaches
                # the shorter tree to the root of the taller tree.
                pa, pb = pb, pa

            parent[pb] = pa
            if rank[pa] == rank[pb]:
                rank[pa] += 1
            return True

        # connect the given edges which have no cost
        for u, v in edges:
            # check both directions since it is an undirected graph
            if (u,v) not in hm and (v, u) not in hm:
                if find(u) != find(v):
                    union(u, v)

        # connect if they have different different parents
        for u, v, cost in edgesToRepair:
            if find(u) != find(v):
                total_cost += cost
                union(u, v)

        # find one parent and check if all nodes have the same parent else its not
connected
        group = find(1)
        for i in range(1,n+1):
            if find(i) != group:
                return -1

        return total_cost


n = 5
edges = [[1, 2], [2, 3], [3, 4], [4, 5], [1, 5]]
edgesToRepair = [[1, 2, 12], [3, 4, 30], [1, 5, 8]]
print(minCostForRepair(n, edges, edgesToRepair))
```

——-

13. Maximum average of subtree

Solution 1:

```
def maximumAverageSubtree(root):
      def dfs(root):
         if not root:
            return 0, 0, (0, root)
         leftNum, leftAverage, leftMaxAverage = dfs(root.left)
         rightNum, rightAverage, RightMaxAverage = dfs(root.right)
         currNum = leftNum + rightNum + 1
         currAverage = ((leftNum * leftAverage) + root.val + (rightNum *
rightAverage))                /(1+ leftNum + rightNum)
         currMaxAverage = (currAverage, root)
         if leftMaxAverage[0] > currMaxAverage[0]:
            currMaxAverage = leftMaxAverage
         if RightMaxAverage[0] > currMaxAverage[0]:
            currMaxAverage = RightMaxAverage
         return currNum, currAverage, currMaxAverage
      return dfs(root)[2][1]
```

O(n)

Solution 2:

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



class Solution:
    def maximumAverageSubtree(self, root: TreeNode) ->
float:
        self.dic = {}
        self.inordersum(root)
        return max(self.dic.items(), key = lambda x: x[1])
```

```
[0]

    def inordersum(self,root):
        if root:
            total = root.val
            nodeCount = 1

            for child in root.children:
                childSum,childCount =
self.inordersum(child)
                total += childSum
                nodeCount += childCount



            avg = (total)/(nodeCount)

            if nodeCount != 1:
                self.dic[root.val] = avg
            return [total,nodeCount]
        else:
            return [0,0]
```

```
14. Longest string made up of only vowels

def longestVowelsOnlySubstring(S):
    S = S.lower()
    temp, aux, vowels = 0, [], set('aeiou')
    # Count the length of each vowel substring
    for c in S + 'z':
        if c in vowels:
            temp += 1
```

```
        elif temp:
            aux.append(temp)
            temp = 0
    # If the first letter is not vowel, you must cut the
head
    if S[0] not in vowels: aux = [0] + aux
    # If the last letter is not vowel, you must cut the
tail
    if S[-1] not in vowels: aux += [0]
    # Max length = max head + max tail + max middle
    return aux[0] + aux[-1] + max(aux[1:-1]) if len(aux) >=
3 else sum(aux)
```

---

15. Battleship

```
def battleship(N, s, t):
    matrix = [[0] * N for _ in range(N)]

    ships = s.split(",")
    hits = t.split(" ")
    for i in range(len(ships)):
        ships[i] = ships[i].split(" ")

    original = set()
    for i in range(len(ships)):
        top_left = ships[i][0]
        bottom_right = ships[i][1]
        top_x = int(top_left[:-1])-1
        top_y = ord(top_left[-1])-65
        bottom_x = int(bottom_right[:-1])-1
        bottom_y = ord(bottom_right[-1])-65
        vertical = bottom_x - top_x + 1
        horizonal = bottom_y - top_y + 1
        for m in range(top_x, top_x + vertical):
            for n in range(top_y, top_y + horizonal):
                matrix[m][n] = i+1
        original.add(i+1)

    hitted = set()
    for hit in hits:
        x = int(hit[:-1])-1
```

```
        y = ord(hit[-1])-65
        if matrix[x][y] != 0:
            hitted.add(matrix[x][y])
            matrix[x][y] = 0

    updated = set()
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] != 0:
                updated.add(matrix[i][j])

    sunk = len(original - updated)
    hitted_but_not_sunk = len(hitted & updated)

    return sunk, hitted_but_not_sunk
```

---

16. Longest string without 3 consecutive characters

```
import heapq
class Solution:
    def longestStringNo3repeats(self, A, B, C):
        pq = [] # initialize priority queue as array

        res = "" # initialize final answer as empty string

        for count, letter in (A, 'a'), (B, 'b'), (C, 'c'):
            heapq.heappush(pq, (-count, letter)) # setup
negative count tuple so that it is a max heap

        while len(pq)>=2:
            cur_count1, cur_letter1 = heapq.heappop(pq) #
pop the largest count
            cur_count2, cur_letter2 = heapq.heappop(pq) #
pop the 2nd largest
```

```
            if len(res) < 2:
                res += cur_letter1
                cur_count1 += 1

            elif len(res) >= 2 and res[-2:] !=
cur_letter1*2: # if the last 2 letters are the same as
current letter
                res += cur_letter1
                cur_count1 += 1

            else:
                res += cur_letter2
                cur_count2 += 1

            if cur_count1 < 0: # if there is count left,
push the tuple back to pq
                heapq.heappush(pq, (cur_count1,
cur_letter1))
            if cur_count2 < 0: # same for 2nd largest, if
there is count left, push the tuple back to pq
                heapq.heappush(pq, (cur_count2,
cur_letter2))

        # try squeezing in more of the last letter if
possible
        if pq and res[-1] != pq[0][1]:
          last_letter = pq[0][1]*min(abs(pq[0][0]), 2) #
squeeze in 2 letters
        elif pq and res[-2] != pq[0][1] and res[-1] ==
pq[0][1]:
          last_letter = pq[0][1] # squeeze in 1 letter
        else: last_letter = "")
```

```
        return res+last_letter

S = Solution()
print(S.longestStringNo3repeats(0, 1, 1))
```

---

17. Substring of size k with k distinct characters

```
def substringk(s, k):
    if not s or k == 0:
        return []

    letter, res = {}, set()
    start = 0
    for i in range(len(s)):
        if s[i] in letter and letter[s[i]] >= start:
            start = letter[s[i]]+1
        letter[s[i]] = i
        if i-start+1 == k:
            res.add(s[start:i+1])
            start += 1
    return list(res)
```

---

18. Count substrings with exactly k distinct characters

```
def subStringsWithKDistinctCharacters(s, k):
   s = list(s)

   def atMost(k):
      count = collections.defaultdict(int)
      left = 0
      ans = 0
      for right, x in enumerate(s):
         count[x] += 1
```

```
            while len(count) > k:
                count[s[left]] -= 1
                if count[s[left]] == 0:
                    del count[s[left]]
                left += 1
            ans += right - left + 1
        return ans
    return atMost(k) - atMost(k-1)
```

---

19. Zombie matrix / Min hours to send file to all available servers / "rotting oranges"

```
from collections import deque
def minHours(rows: int, columns: int, grid:
List[List[int]]) -> int:
    if not grid: return -1
    RLEN, CLEN = len(grid), len(grid[0])
    q = deque()
    hours = -1

    # get adjacents
    def neighbors(pos):
        r,c = pos
        for nr,nc in ((r-1,c), (r+1,c), (r,c-1), (r,c+1)):
            if 0 <= nr < RLEN and 0 <= nc < CLEN:
                yield nr,nc

    # get all zombies!!!
    for r, row in enumerate(grid):
        for c, val in enumerate(row):
            if val == 1:
                q.append((r,c))
    while q:
        for _ in range(len(q)):
            r,c = q.popleft()
```

```
            for nr,nc in neighbors((r,c)):
                if grid[nr][nc] == 0:
                    grid[nr][nc] = 1
                else: continue
                q.append((nr,nc))
        hours += 1
    return hours
```

20.Min Cost to Connect Ropes / Min Time to Merge Files [Experienced]

```
from heapq import heappop, heappush, heapify
def minCost(ropes):
  if not ropes: return 0
  if len(ropes) == 1: return ropes[0]
  heapify(ropes)
  cost = 0
  while len(ropes) > 1:
    a, b = heappop(ropes), heappop(ropes)
    cost += a+b
    if ropes:
      heappush(ropes, a+b)
  return cost
```
