300. Longest Increasing Subsequence
We will go through our array and use an array to maintain our current longest increasing array. If res[-1] < current, we will append the new one to res, else we will use binary search to get the index that we want to replace the element that closest to it and larger than it at the same time. TC is O(nlogn), SC is O(n)

```
from bisect import *
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums:
          return 0
        max_length = 1
        res = [nums[0]]

        for num in nums[1:]:
          if num < res[-1]:
            idx = bisect_left(res, num)
            res[idx] = num
          elif num > res[-1]:
            res.append(num)

        return len(res)
```

547. Friend Circles
We will go through all elements in nested array and use union find to find all friends connected. We will use a hashmap to record all nodes' parent. TC is O(n^2), SC is O(n)

```
class Solution:
    def findCircleNum(self, M: List[List[int]]) -> int:
        self.parent = {}
        if not M or not M[0]:
          return 0
        count = len(M)
        rows = len(M)
        cols = len(M[0])
        for i in range(rows):
          for j in range(cols):
            if i != j and M[i][j] == 1:
              i_p = self.findParent(i)
              j_p = self.findParent(j)
              if i_p != j_p:
                self.parent[i_p] = j_p
                count -= 1
        return count
```

```
  def findParent(self, i):
    memo = i
    while i in self.parent and self.parent[i] != i:
      i = self.parent[i]
    self.parent[memo] = i
    return i
```

695. Max Area of Island

We will iterate through all elements in the nested array. We will use dfs to search all connected islands until the end of array. TC is O(n*m), SC is O(n)

```
class Solution:
  def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
    if not grid or not grid[0]:
      return 0
    rows = len(grid)
    cols = len(grid[0])
    count = 0
    max_count = 0
    for i in range(rows):
      for j in range(cols):
        if grid[i][j] == 1:
          count = 1
          grid[i][j] = 0
          cur = [[i, j]]
          while cur:
            next_ite = []
            for n_i, n_j in cur:
              for d_i, d_j in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
                new_i, new_j = n_i + d_i, n_j + d_j
                if 0 <= new_i < rows and 0 <= new_j < cols and grid[new_i][new_j] == 1:
                  grid[new_i][new_j] = 0
                  count += 1
                  next_ite.append([new_i, new_j])
            cur = next_ite
          max_count = max(max_count, count)
    return max_count
```

733. Flood Fill

We will use dfs to refill all cells connecting to these cells and equals to original color. TC is O(m*n), SC is O(1)

```
class Solution:
  def floodFill(self, image: List[List[int]], sr: int, sc: int, newColor: int) -> List[List[int]]:
    def dfs(r, c, src_color, rows, cols):
```

```
        for d_i, d_j in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
          new_i, new_j = r + d_i, c + d_j
          if 0 <= new_i < rows and 0 <= new_j < cols and image[new_i][new_j] == src_color:
            image[new_i][new_j] = newColor
            dfs(new_i, new_j, src_color, rows, cols)
      if image[sr][sc] == newColor:
        return image
      else:
        src_color = image[sr][sc]
        image[sr][sc] = newColor
        dfs(sr, sc, src_color, len(image), len(image[0]))
      return image
```

841. Keys and Rooms

We will use bfs and go through all possible nodes, in the end, we will check whether it visited all door. TC is O(n),SC is O(n)

```
class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        length = len(rooms) - 1
        cur = rooms[0]
        visited = {}
        visited[0] = True
        while cur:
          next_ite = []
          for i in cur:
            if i not in visited:
              length -= 1
              visited[i] = True
              next_ite.extend(rooms[i])
          cur = next_ite
        return length == 0
```