218. The Skyline Problem
We will scan each line and once height changed, we will append our edge point with new height into our array. TC is O(nlog(n))

```
from heapq import *
class Solution:
    def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        events = sorted([[L, -H, R] for L, R, H in buildings] + [[R, 0, None] for _, R, _ in buildings])
        hp, res = [(0, float('inf'))], [[0, 0]]
        for x, negH, R in events:
            while x >= hp[0][1]:
                heappop(hp)

            if R:
                heappush(hp, (negH, R))

            if res[-1][-1] + hp[0][0]:
                res.append([x, -hp[0][0]])
        return res[1:]
```

312. Burst Balloons
We will use dp to get maximum dp[left][right] from bottom up. We will assume i is the last balloon to burn within (left, right). We will start from every three elements until n elements. TC is O(n^3)

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        nums = [1] + nums + [1]
        length = len(nums)
        dp = [[0 for _ in range(length)] for _ in range(length)]

        for k in range(1, length):
            for left in range(0, length - k):
                right = left + k
                for i in range(left + 1, right):
                    dp[left][right] = max(dp[left][right], nums[left] * nums[i] * nums[right] + dp[left][i] +
dp[i][right])
        return dp[0][length - 1]
```

51. N-Queens
We will use dfs to find our queen row by row. Because no two queens could locate on one line, we will check the next queen doesn't have the same y-value, and x - y and x + y cannot exist in the previous rows. We will use the array to store the first one, because we don't want to lose its order, and will use set for the latter two for the speedness. TC is O(n*n)

```
class Solution:
```

```
    def solveNQueens(self, n: int) -> List[List[str]]:
        def dfs(cor, t_dif, t_sum):
          p = len(cor)
          if p == n:
            result.append(cor[:])
            return
          for q in range(n):
            if q not in cor and p + q not in t_sum and p - q not in t_dif:
              dfs(cor + [q], t_dif | set([p - q]), t_sum | set([p + q]))

        result = []
        dfs([], set(), set())
        return [['.' * i + 'Q' + '.' * (n - i - 1) for i in cor] for cor in result]
```

52. N-Queens II

It's similar to the previous question. We will use the same way. But since we don't need to output the graph, we could also use set to record previous columns. TC is O(n ** 2)

```
class Solution:
    def totalNQueens(self, n: int) -> int:
        self.result = 0
        def dfs(cor, t_dif, t_sum):
          p = len(cor)
          if p == n:
            self.result += 1
            return
          for q in range(n):
            if q not in cor and p + q not in t_sum and p - q not in t_dif:
              dfs(cor | set([q]), t_dif | set([p - q]), t_sum | set([p + q]))

        dfs(set(), set(), set())
        return self.result
```

63. Unique Paths II

We will use dp and traverse row by row. If that cell is an obstacle, we will set dp[x][y] to zero. If it's not, we will add bottom and left number to this one. In the end, dp[-1][-1] is what we want. TC is O(m * n)

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        if not obstacleGrid or not obstacleGrid[0]:
          return 0
        if obstacleGrid[0][0] == 1 or obstacleGrid[-1][-1] == 1:
          return 0
        rows = len(obstacleGrid)
```

```
cols = len(obstacleGrid[0])

dp = [[0 for j in range(cols)] for i in range(rows)]
dp[0][0] = 1
for j in range(1, cols):
  if obstacleGrid[0][j] == 1:
    dp[0][j] = 0
  else:
    dp[0][j] = dp[0][j - 1]

for i in range(1, rows):
  if obstacleGrid[i][0] == 1:
    dp[i][0] = 0
  else:
    dp[i][0] = dp[i - 1][0]

for i in range(1, rows):
  for j in range(1, cols):
    if obstacleGrid[i][j] == 1:
      dp[i][j] = 0
    else:
      dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
return dp[-1][-1]
```