85. Maximal Rectangle

We would use dp to solve this question. We will iterate row by row. In each row, we will calculate each height for each column. If it’s ‘1’, we will add the previous height[j] by 1, if not, we will set it to zero. For left boundary, if it’s 1, we will set it max of previous left[j] and left continous ‘1’’s left index on this row. The same as right boundary. After each row, we will calculate the max area at this time. TC is O(N * 2)

```
class Solution:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        if not matrix or not matrix[0]:
            return 0
        rows = len(matrix)
        cols = len(matrix[0])
        height = [0] * cols
        left = [0] * cols
        right = [cols] * cols
        area = 0
        cur_left, cur_right = 0, cols

        for i in range(rows):
            cur_left, cur_right = 0, cols
            for j in range(cols):
                if matrix[i][j] == '1':
                    height[j] += 1
                    left[j] = max(cur_left, left[j])
                else:
                    height[j] = 0
                    left[j] = 0
                    cur_left = j + 1



            for j in range(cols - 1, -1, -1):
                if matrix[i][j] == '1':
                    right[j] = min(cur_right, right[j])
                else:
                    right[j] = cols
                    cur_right = j

            for j in range(cols):
                area = max(area, (right[j] - left[j]) * height[j])
        return area
```

924. Minimize Malware Spread

We need to use bfs to find the whole network where initial element is in and if there are only one common node between initial and this network, we will push(-nodes_number, node) to record for further sorting. In the end, if result is not empty, we will get the minimum one. If not, we will get the min from initial. TC is O(n*n)

```
class Solution:
    def minMalwareSpread(self, graph: List[List[int]], initial: List[int]) -> int:
        visited = set()
        result = []

        for node in initial:
            if node in visited:
                continue
            visited.add(node)
            cur = set([node])
            net_work = set()
            net_work.add(node)

            while cur:
                next_ite = set()
                for ite_node in cur:
                    for idx, connected in enumerate(graph[ite_node]):
                        if idx != node and idx not in visited and connected == 1:
                            next_ite.add(idx)
                            net_work.add(idx)
                            visited.add(idx)
                cur = next_ite

            if len(set(net_work) & set(initial)) == 1:
                result.append([-len(net_work), node])
        if result:
            return min(result)[1]
        else:
            return min(initial)
```

45. Jump Game II

We will use greedy algorithm to find next index that could take us to the longest distance. Then we will pick that one. And repeat it, until we could reach last position in the next step. TC is O(n):

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        cur_idx, count = 0, 0
        length = len(nums)

        if length <= 1:
            return 0

        while cur_idx < length:
            next_index = cur_idx
            max_distance = 0
            if cur_idx + nums[cur_idx] >= length - 1:
                return count + 1
            for i in range(cur_idx + 1, cur_idx + nums[cur_idx] + 1):
                if i + nums[i] > max_distance:
                    max_distance = i + nums[i]
                    next_index = i
            cur_idx = next_index
            count += 1
        return count
```

975. Odd Even Jump
We will take advantage of stack to find next higher index for current element, and next lower index for current element. We will use two arraies higher and lower to mark from current position , whether it will succeed jumping to next lower or next higher position. Then we will iterate from end to start.current higher's success depends on the next lower one's success. And vice versa. Higher's sum is the result we want. TC is O(nlogn).

```
class Solution:
    def oddEvenJumps(self, A: List[int]) -> int:
        length = len(A)

        next_higher = [0] * length
        next_lower = [0] * length

        stack = []
        for _, i in sorted([a, i] for i, a in enumerate(A)):
            while stack and stack[-1] < i:
                next_higher[stack.pop()] = i
            stack.append(i)

        stack = []
        for _, i in sorted([-a, i] for i, a in enumerate(A)):
            while stack and stack[-1] < i:
                next_lower[stack.pop()] = i
            stack.append(i)

        higher = [0] * length
        lower = [0] * length
        higher[-1], lower[-1] = 1, 1

        for i in range(length - 1)[::-1]:
            higher[i] = lower[next_higher[i]]
            lower[i] = higher[next_lower[i]]
        return sum(higher)
```

126. Word Ladder II

We use bfs to check the distance from start word to end word. Also we will record each word’s neighbors for bfs in the future. In DFS, we will append path that ends up with endword and within the got distance to res.

```
WORDTABLE = 'abcdefghijklmnopqrstuvwxyz'
from collections import defaultdict
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        wordList = set(wordList)
        res = []
        self.len = None
        self.nebor = defaultdict(set)

        self.len = self.bfs([beginWord], set(), endWord, wordList)

        if self.len > 0:
            self.dfs([beginWord], res, endWord)
        return res

    def dfs(self, cur, res, endWord):
        if len(cur) >= self.len:
            return
        for newWord in self.nebor[cur[-1]]:
            cur.append(newWord)
            if newWord == endWord:
                res.append(cur[:])
            self.dfs(cur, res, endWord)
            cur.pop()

    def bfs(self, cur, visited, endWord, wordList):
        count = 1
```

```
mark = False
while cur:
    next_ite = set()
    for word in cur:
        for i in range(len(word)):
            for l in WORDTABLE:
                newWord = word[:i] + l + word[i + 1:]
                if newWord in wordList and newWord not in visited:
                    self.nebor[word].add(newWord)
                    if newWord == endWord:
                        mark = True
                    next_ite.add(newWord)
    for w in next_ite:
        visited.add(w)
    if mark:
        return count + 1
    cur = next_ite
    count += 1
return -1
```