Hot 100

2022年5月21日 17:01

1. 电话号码的字母组合
   - 回溯法+start控制回溯树的层数
   - ```
     # 首先定义电话号码对应的字母列表
     # 相当于从每个字符串取出一个数字进行组合，求总的组合数
     # 回溯树的每一层都是一个新的选择列表
     class Solution:
         def letterCombinations(self, digits):
             if not digits:
                 return []
             res = []
             cur = []
             temp = ['', '', 'abc', 'def', 'ghi', 'jkl', 'mno', 'pqrs', 'tuv', 'wxyz']
             # start表示digits的索引,用于控制当前回溯树的层数，每一层都在一个新的字符串中选择一个元素
             def recur(start, l):
                 if start == l:
                     res.append(''.join(cur))
                     return
                 # 当前的选择列表
                 strs = temp[int(digits[start])]
                 for item in strs:
                     cur.append(item)
                     # 选择下一层的字母
                     recur(start+1, l)
                     cur.pop()
             recur(0, len(digits))
             return res
     ```
2. 下一个排列
   - 逆序遍历+找升序对+交换+降序反转
   - ```
     # 从后向前遍历,找到第一个相邻的升序数对(i,j)
     # 在子数组[j:end]中逆序遍历，找到第一个比nums[i]大的数字nums[k]
     # 将nums[k]与num[i]进行调换
     # 将子数组[j:end]进行升序排列
     # 返回整个数组

     class Solution:
         def nextPermutation(self, nums):
             """
             Do not return anything, modify nums in-place instead.
             """
             if len(nums) <= 1:
                 return
             right = len(nums) - 1
             while right > 0:
                 if nums[right] <= nums[right-1]:
                     right -= 1
                     continue
                 else:
                     cur = right-1
                     temp = right
                     for i in range(len(nums)-1, right, -1):
                         temp = i
                         if nums[i] > nums[cur]:
                             break
                     nums[temp], nums[cur] = nums[cur], nums[temp]
                     nums[:] = nums[:right] + nums[right:][::-1]
                     return nums
             nums[:] = nums[::-1]
             return nums
     ```
3. 字母异位词分组
   - 哈希表+每个字符串排序比较
   - ```
     # 将数组中的字符串升序排序重新,升序排序结果相同的为一组
     class Solution:
         def groupAnagrams(self, strs):
             if not strs:
                 return [['']]
             hash_dict = {}
             for item in strs:
                 temp = ''.join(sorted(item))
                 if temp in hash_dict:
                     hash_dict[temp].append(item)
                 else:
                     hash_dict[temp] = [item]
             return list(hash_dict.values())
     ```
4. 跳跃游戏
   - 贪心算法+更新最远能到达的地方
   - ```
     # 定义当前可以到达的最远位置max_idx
     # 如果能到达位置i,那么它前面的位置都可以到达
     # 根据i+num[i]更新当前能到达的最远位置
     class Solution:
         def canJump(self, nums):
             max_idx = 0
             for i in range(len(nums)):
     ```

```
            # 如果当前位置i<=max_idx,那么当前位置i一定是可以到达的
            if i <= max_idx and i + nums[i] > max_idx:
                max_idx = i + nums[i]
        return max_idx >= len(nums) - 1
```

5. 不同的二叉搜索树
   - 区间[low, high]可以构造的BST数的个数+以每个元素作为根节点
   - ```
     # 以val为根节点，则[low, val-1]属于左子树,[val+1, high]属于右子树
     # 以val为根节点总共可以构造的BST个数=左子树个数*右子树个数
     # 可以递归求解
     class Solution:
         def numTrees(self, n):
             # 备忘录
             memo = [[0 for _ in range(n+1)] for _ in range(n+1)]
             # 区间[low, high]可以构造的BST个数
             def recur(low, high):
                 if low >= high:
                     return 1
                 if memo[low][high] != 0:
                     return memo[low][high]
                 res = 0
                 # 以[low, high]中的每个元素作为根节点构造BST
                 for i in range(low, high+1):
                     left_num = recur(low, i-1)
                     right_num = recur(i+1, high)
                     res += right_num * left_num
                 memo[low][high] = res
                 return res
             res = recur(1, n)
             return res
     ```
6. 不同的二叉搜索树II
   - 返回构造的每个BST树
   - ```
     class Solution:
         def generateTrees(self, n: int) -> List[TreeNode]:
             if n == 0:
                 return []
             # 构造区间[low, high]中所有合法的BST数
             def recur(low, high):
                 res = []
                 if low > high:
                     res.append(None)
                     return res
                 # 以区间中的每一个点最为根节点
                 for i in range(low, high+1):
                     left_list = recur(low, i-1)
                     right_list = recur(i+1, high)
                     for left in left_list:
                         for right in right_list:
                             root = TreeNode(i)
                             root.left = left
                             root.right = right
                             res.append(root)
                 return res
             rel = recur(1, n)
             return rel
     ```
7. 二叉树展开为链表
   - 左右子树先展平+根节点的右指针指向左子链表+左子链表的末尾链接右子链表的头结点
   - ```
     class Solution:
         def flatten(self, root: TreeNode) -> None:
             """
             Do not return anything, modify root in-place instead.
             """
             if not root:
                 return
             # 将左右子树展平为链表
             self.flatten(root.left)
             self.flatten(root.right)
             # 左右子链表分配指针
             left = root.left
             right = root.right
             # 根节点右指针指向左子链表
             root.right = left
             root.left = None
             # 找左子链表的末尾节点
             p = root
             while p.right:
                 p = p.right
             p.right = right
     ```
8. 最长连续序列
   - 去重+排序+滑动窗口
   - ```
     # 去重之后排序+滑动窗口判断
     class Solution:
         def longestConsecutive(self, nums):
             if not nums:
                 return 0
             nums = list(set(nums))
             nums.sort()
             left = 0
             right = 0
             res = 0
             while right < len(nums):
     ```

```
            if nums[right] - nums[left] == right - left:
                res = max(res, right-left+1)
                right += 1
            else:
                left = right
                right += 1
        return res
```

9. 单词拆分
   - 动态规划+背包问题+dp[i]表示第i个字符之前的子字符串是否都能匹配
   - ```
     class Solution:
         def wordBreak(self, s: str, wordDict: List[str]) -> bool:
             dp = [False] * (len(s) + 1)
             dp[0] = True
             for i in range(len(s)):
                 for j in range(i, len(s)+1):
                     if dp[i] and s[i:j] in wordDict:
                         dp[j] = True
             return dp[-1]
     ```
10. 乘积最大子数组
    - 动归+维护最大值+维护最小值
    - ```
      # 由于存在负数，因此不能像子数组和那样只考虑当前元素大小和前一个元素的最大子数组和
      # 以负数结尾的乘积可以负负得正
      # 如果当前是正数，就希望之前的乘积尽可能大
      # 如果当前为负数，就希望之前的乘积尽可能小
      # 保存每个位置的最大乘积和最小乘积
      class Solution:
          def maxProduct(self, nums):
              dp = [[0, 0] for _ in range(len(nums))]
              # 最大值
              dp[0][0] = nums[0]
              # 最小值
              dp[0][1] = nums[0]
              for i in range(1, len(nums)):
                  dp[i][0] = max(dp[i-1][0]*nums[i], dp[i-1][1]*nums[i], nums[i])
                  dp[i][1] = min(dp[i-1][0]*nums[i], dp[i-1][1]*nums[i], nums[i])
              return max(dp)[0]
      ```
11. 所有可能的路径
    - 图遍历
    - ```
      class Solution:
          def allPathsSourceTarget(self, graph):
              res = []
              cur = []
              # 访问当前节点并判断当前节点是不是目标节点
              def recur(item, target):
                  cur.append(item)
                  if item == target:
                      res.append(cur[:])
                      # 一次访问item的相邻节点
                  for v in graph[item]:
                      recur(v, target)
                  cur.pop()
              recur(0, len(graph)-1)
              return res
      ```
12. 课程表
    - 构建图+图的遍历+判断是否有环
    - ```
      class Solution:
          def canFinish(self, numCourses, prerequisites):
              # 构建图
              graph = self.build_graph(numCourses, prerequisites)
              # 图的遍历
              # 如果当前访问的节点已经在访问路径上，说明出现了环
              vis = [0] * numCourses
              onPath = [0] * numCourses
              self.flag = False    # 是否有环
              # 访问当前顶点
              def recur(cur):
                  # 已经在路径上
                  if onPath[cur] == 1:
                      self.flag = True
                      return
                  # 没有在路径上，但是已经被访问过了或者已经找到环了
                  if vis[cur] == 1 or self.flag:
                      return
                  vis[cur] = 1
                  onPath[cur] = 1
                  # 遍历cur的邻接节点
                  for v in graph[cur]:
                      recur(v)
                  # 离开cur时撤销
                  onPath[cur] = 0
              # 以每个点为起点进行访问
              for i in range(numCourses):
                  recur(i)
              return not self.flag


          # 构建图的邻接表形式
      ```

```
    def build_graph(self, numCourses, prerequisites):
        graph = [[] for _ in range(numCourses)]  # 下标表示图中的顶点
        for item in prerequisites:
            start = item[1]
            end = item[0]
            graph[start].append(end)
        return graph
```

13. 实现Trie前缀树

```
# 定义前缀树结构Node
# 前缀树包括:一个标记(是否是结束字符)+指向子节点位置的列表
# 前缀树结构
class Node():
    def __init__(self):
        # 是否是结束字符
        self.isEnd = False
        # 保存26个英文字母
        self.child = [None for _ in range(26)]
    # 是否包含key
    def contain_key(self, key):
        return self.child[ord(key)-ord('a')]
    # 返回key的值
    def get(self, key):
        return self.child[ord(key)-ord('a')]
    # 插入key
    def put(self, key):
        self.child[ord(key)-ord('a')] = Node()

class Trie:
    def __init__(self):
        self.root = Node()

    def insert(self, word: str) -> None:
        p = self.root
        # 从word的第一个字符开始匹配
        for item in word:
            if not p.contain_key(item):
                p.put(item)
            p = p.get(item)
        # 表示一个单词插入结束
        p.isEnd = True

    def search(self, word: str) -> bool:
        p = self.root
        for item in word:
            if not p.contain_key(item):
                return False
            else:
                p = p.get(item)
        # 查找单词需要检查最后一个字符是否是结束符
        return p.isEnd

    def startsWith(self, prefix: str) -> bool:
        p = self.root
        for item in prefix:
            if not p.contain_key(item):
                return False
            else:
                p = p.get(item)
        # 检查前缀不需要
        return True
```

14. 最大正方形

```
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        # 以当前位置为正方形右下角的正方形的最大边长
        dp = [[0 for _ in range(len(matrix[0]))] for _ in range(len(matrix))]
        # 初始化第一列
        for i in range(len(matrix)):
            if matrix[i][0] == '1':
                dp[i][0] = 1
        # 初始化第一列
        for i in range(len(matrix[0])):
            if matrix[0][i] == '1':
                dp[0][i] = 1
        # 转换转移
        for i in range(1, len(matrix)):
            for j in range(1, len(matrix[0])):
                if matrix[i][j] == '1':
                    dp[i][j] = min(dp[i-1][j-1], dp[i][j-1], dp[i-1][j]) + 1
                else:
                    dp[i][j] = 0
        max_len = 0
        for i in range(len(dp)):
            for j in range(len(dp[0])):
                if dp[i][j] > max_len:
                    max_len = dp[i][j]
        return max_len * max_len
```

15. 完全平方数
    - 背包问题

```
# 背包问题
```

```
# 用完全平方数将容量为n的背包装满，最少需要几个
class Solution:
    def numSquares(self, n):
        # 最坏情况下n用1表示，共需要n个
        dp = [i for i in range(n+1)]
        for i in range(1, n+1):
            j = 1
            while i - j*j >= 0:
                dp[i] = min(dp[i], dp[i-j*j]+1)
                j += 1
        return dp[-1]
```

16. 移动零
    - 双指针+找不为0的与0交换
    - ```
      class Solution:
          def moveZeroes(self, nums: List[int]) -> None:
              """
              Do not return anything, modify nums in-place instead.
              """
              if len(nums) <= 1:
                  return
              left = 0
              right = 0
              while right < len(nums):
                  if nums[right] != 0:
                      nums[right], nums[left] = nums[left], nums[right]
                      left += 1
                  right += 1
      ```
17. 寻找重复数
    - 快慢指针+相当于寻找链表环入口
    - ```
      # 题目已知一定存在重复数
      # 相当于链表中一定存在环，需要找到换入口
      # 根据索引->值的映射，就可以得到一条链表
      # 思想与求链表环入口一致
      class Solution:
          def findDuplicate(self, nums):
              slow = 0
              fast = 0
              # slow走一步
              slow = nums[slow]
              # fast走两步
              fast = nums[nums[fast]]
              # 寻找相遇点
              while slow != fast:
                  slow = nums[slow]
                  fast = nums[nums[fast]]
              # 退回链表头部
              slow = 0
              while slow != fast:
                  slow = nums[slow]
                  fast = nums[fast]
              return slow
      ```
18. 比特位计数
    - 如果 I 的最后一位是 1，那么 i&(i-1) 会消掉最后一个1，因此利用 i&(i-1) 的结果 +1 即可得到i的结果
    - ```
      class Solution:
          def countBits(self, n: int) -> List[int]:
              res = [0] * (n + 1)
              for i in range(1, n+1):
                  res[i] = res[i & (i-1)] + 1
              return res
      ```
19. 字符串解码
    - ```
      # 当遇到'['时，就将它前面已经形成的(字符串,数组)压入栈
      # 遇到']'时,弹出栈顶的元祖,给已经形成的字符串后面添加子字符串
      class Solution:
          def decodeString(self, s):
              res = ''
              num = 0
              stack = []
              for item in s:
                  if item.isdigit():
                      num = num * 10 + int(item)
                  elif item == '[':
                      stack.append((res, num))
                      res = ''
                      num = 0
                  elif item == ']':
                      pop_item = stack.pop()
                      res = pop_item[0] + res * pop_item[1]
                  else:
                      res += item
              return res
      ```
20. 除法求值
    - 构建有向顶点图+构建有向权值图+图的深度优先遍历
    - ```
      # 构建顶点有向图：每个字符当做顶点
      # 构建权重有向图：存储已知的两个顶点的商
      from collections import defaultdict
      class Solution:
          def calcEquation(self, equations, values, queries):
              # 构建图
      ```
      ```

```
        graph = defaultdict(set)  # graph[item]=set()
        weight = defaultdict(lambda :0)  # weight[item] = 0
        for idx, item in enumerate(equations):
            graph[item[0]].add(item[1])
            graph[item[1]].add(item[0])
            weight[(item[0], item[1])] = values[idx]
            weight[(item[1], item[0])] = float(1 / values[idx])
        # 深度优先搜索
        def recur(start, end, vis):
            # 顶点不在图中
            if not start in graph or not end in graph:
                return 0
            # 边已经存在于图中
            if end in graph[start]:
                return weight[(start, end)]
            # 顶点已经访问过了
            if start in vis:
                return 0
            # 标记顶点
            vis.add(start)
            res = 0
            for item in graph[start]:
                res = recur(item, end, vis) * weight[(start, item)]
                # 备忘录
                if res != 0:
                    weight[(start, end)] = res
                    break
            # 撤销标记
            vis.remove(start)
            return res

        res = []
        for item in queries:
            temp = recur(item[0], item[1], set())
            if temp == 0:
                res.append(-1.0)
            else:
                res.append(temp)
        return res
```

21. 根据身高重建队列
    - 贪心+先确定身高降序排列+以k作为索引插入
    - ```
      # 贪心算法+先确定一个维度+再确定另一个维度
      # 先按照身高降序排列+遇到身高相同的按照个数k升序排列
      # 遍历一遍已排序好的people+以k为索引进行插入
      class Solution:
          def reconstructQueue(self, people):
              people.sort(key=lambda x:(-x[0], x[1]))
              res = []
              for item in people:
                  res.insert(item[1], item)
              return res
      ```
22. 分割等和子集
    - 背包问题+dp[i][j]表示num的前j个元素中是否存在一种选择方法将容量为i的背包装满
    - ```
      class Solution:
          def canPartition(self, nums):
              # 可以看给容量为summ//2的背包中装nums中的数值，nums中是否存在一个组合可以满足这个要求
              if len(nums) <= 1:
                  return False
              summ = 0
              for item in nums:
                  summ += item
              if summ % 2 == 1:
                  return False
              summ = summ // 2
              # dp[i][j]表示用nums中的前i个元素装容量为j的背包，能否恰好装满
              dp = [[False for _ in range(summ+1)] for _ in range(len(nums)+1)]
              # base case
              for i in range(len(nums)+1):
                  dp[i][0] = True
              for i in range(1, len(nums)+1):
                  for j in range(1, summ+1):
                      if j - nums[i-1] < 0:
                          dp[i][j] = dp[i-1][j]
                      else:
                          dp[i][j] = dp[i-1][j] | dp[i-1][j-nums[i-1]]
              return dp[-1][-1]
      ```
23. 路径总和III
    - 深度优先遍历
    - ```
      class Solution:
          def pathSum1(self, root, targetSum):
              if not root:
                  return 0
              res = 0
              def recur(root, diff):
                  if not root:
                      return 0
                  cur = 0
                  if root.val == diff:
                      cur += 1
                  cur += recur(root.left, diff - root.val)
                  cur += recur(root.right, diff - root.val)
                  return cur
      ```
      ```

```
        res += recur(root, targetSum)
        res += self.pathSum1(root.left, targetSum)
        res += self.pathSum1(root.right, targetSum)
        return res
```

- 前缀和
- ```
  class Solution:
      # 前缀和
      def pathSum2(self, root, targetSum):
          # preSum[item]表示前缀和为item的路径组合个数
          import collections
          preSum = collections.defaultdict(int)
          preSum[0] = 1
          # root节点之前的前缀和为cur
          def recur(root, cur):
              if not root:
                  return 0
              temp = 0
              cur += root.val
              temp += preSum[cur - targetSum]
              preSum[cur] += 1
              temp += recur(root.left, cur)
              temp += recur(root.right, cur)
              preSum[cur] -= 1
              return temp
          return recur(root, 0)
  ```

24. 找到数组中消失的数字
    - 无序数组+以数组中存在的数字为索引，将其对应元素改为负值+数组中正值对应的索引idx+1为缺失值
    - ```
      # nums中的值为1-n
      # 将以nums中的值val为下标的对应的值改为负值
      # 遍历数组，数组中不是负值的数对应的索引idx+1就为缺失的值
      class Solution:
          def findDisappearedNumbers(self, nums):
              for item in nums:
                  nums[abs(item)-1] = -abs(nums[abs(item)-1])
              res = []
              for idx, val in enumerate(nums):
                  if val > 0:
                      res.append(idx+1)
              return res
      ```
25. 目标和
    - 递归+备忘录
    - ```
      class Solution:
          def findTargetSumWays(self, nums, target):
              memo = {}
              # 以num[cur]开始的子数组组合，得到目标值diff的不同表达式的数目
              def recur(cur, diff):
                  # base case
                  if cur == len(nums):
                      if diff == 0:
                          return 1
                      return 0
                  key = str(cur) + '_' + str(diff)
                  if key in memo:
                      return memo[key]
                  rel = recur(cur+1, diff-nums[cur]) + recur(cur+1, diff+nums[cur])
                  memo[key] = rel
                  return rel
              return recur(0, target)
      ```
26. 把二叉搜索树转换为累加树
    - 利用全局变量记录比当前节点node的值大的元素的累加和+遍历顺序为右中左
    - ```
      class Solution:
          def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
              if not root:
                  return
              self.sum = 0
              def recur(root):
                  if not root:
                      return
                  recur(root.right)
                  self.sum += root.val
                  root.val = self.sum
                  recur(root.left)
              recur(root)
              return root
      ```
27. 回文子串
    - 双指针+中心扩散法
    - ```
      class Solution:
          def countSubstrings(self, s):
              res = 0
              def recur(i, j):
                  cur = 0
                  # 记录每一个子串
                  while i >= 0 and j < len(s) and s[i] == s[j]:
                      i -= 1
                      j += 1
                      cur += 1
                  return cur
              for i in range(len(s)):
                  res += recur(i, i)
                  res += recur(i, i+1)
              return res
      ```
      ```

28. 最短无序连续子数组
    - 原始数组排序+双指针对照确定左右边界
    - ```
      class Solution:
          def findUnsortedSubarray(self, nums):
              temp = sorted(nums)
              left = 0
              right = len(nums) - 1
              while left <= right and temp[left] == nums[left]:
                  left += 1
              while left <= right and temp[right] == nums[right]:
                  right -= 1
              return right - left + 1
      ```
29. 和为K的子数组
    - 前缀和+哈希表存储累积和temp出现过的次数
    - ```
      # 前缀和
      # 用哈希表记录和为sum的子数组的个数
      # temp表示以第一个元素到当前元素的和
      # 如果temp-K在哈希表中，那么表示存在一个中间子数组的和为K
      # a0-ai的累和为temp-K,a0-aj的累和为temp,那么ai+1-aj的和为K
      class Solution:
          def subarraySum(self, nums, k):
              # hash_dict[item]=val:累计和为item的子数组出现过val次
              hash_dict = {}
              temp = 0
              res = 0
              hash_dict[0] = 1
              for i in range(len(nums)):
                  temp += nums[i]
                  if temp - k in hash_dict:
                      res += hash_dict[temp - k]
                  if temp in hash_dict:
                      hash_dict[temp] += 1
                  else:
                      hash_dict[temp] = 1
              return res
      ```
30. 下一个更大元素I
    - ```
      # 逆序遍历+维护降序单调栈
      class Solution:
          def nextGreaterElement(self, nums1, nums2):
              res = [0] * len(nums1)
              que = []
              for i in range(len(nums2)-1, -1, -1):
                  while que and que[-1] < nums2[i]:
                      que.pop()
                  if nums2[i] in nums1:
                      if que:
                          res[nums1.index(nums2[i])] = que[-1]
                      else:
                          res[nums1.index(nums2[i])] = -1
                  que.append(nums2[i])
              return res
      ```
31. 下一个更大元素II
    - ```
      # 环形数组+数组扩展为2倍+取余
      # 逆序遍历+维护降序单调栈
      class Solution:
          def nextGreaterElements(self, nums):
              l = len(nums)
              res = [0] * l
              que = []
              for i in range(2*l-1, -1, -1):
                  while que and que[-1] <= nums[i % l]:
                      que.pop()
                  res[i % l] = que[-1] if que else -1
                  que.append(nums[i%l])
              return res
      ```
32. 每日温度
    - 单调栈
    - ```
      class Solution:
          def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
              stack = []
              res = [0] * len(temperatures)
              for i in range(len(temperatures)):
                  while stack and temperatures[stack[-1]] < temperatures[i]:
                      cur = stack.pop()
                      res[cur] = i - cur
                  stack.append(i)
              return res
      ```
33. 任务调度器
    - ```
      # 记录出现最多次的任务的次数max_val
      # 需要的最少时间为:(max_val-1)*(n+1)
      # 如果最多的任务次数出现不止一次，再加上最多任务出现的次数
      # T =  (max_val-1)*(n+1)+count
      class Solution:
          def leastInterval(self, tasks, n):
              # 记录每个任务出现的总次数
              hash_dict = {}
              for item in tasks:
                  if item in hash_dict:
                      hash_dict[item] += 1
      ```

```
            else:
                hash_dict[item] = 1
        val = list(hash_dict.values())
        max_val = max(val)
        # 时间
        T = (max_val - 1) * (n + 1) + val.count(max_val)
        if T > len(tasks):
            return T
        else:
            return len(tasks)
```

34. 柱状图中最大的矩形
    - 单调递增栈+存储索引+寻找左右边界
    - # 寻找以heights[i]为高度的矩形的最大面积:向左向右两个方向寻找严格比heights[i]小的元素，作为矩形的左右边界
      # 维护一个单调栈：栈中元素严格单调递增
      # 如果当前元素比栈顶元素大，证明找到了以栈顶元素为高的矩形的右边界
      # 栈中存储高度的索引
      class Solution:
          def largestRectangleArea(self, heights):
              res = 0   # 存储最大值
              heights.append(-1)   # 避免栈中最小元素无法弹出找
              stack = []   # 单调递增栈
              for i in range(len(heights)):
                  while stack and heights[stack[-1]] > heights[i]:
                      # 找到了stack[-1]的右边界
                      cur = stack.pop()
                      # 如果不为空
                      if stack:
                          # 因为栈是严格递增的，所以左边界就是弹出cur之后的栈顶元素
                          cur_s = (i - stack[-1] - 1) * heights[cur]
                      else:
                          # 栈为空，相当于弹出了栈中的最后一个元素，它的左边界就是-1
                          cur_s = (i - (-1) -1) * heights[cur]
                      res = max((res, cur_s))
                  stack.append(i)
              return res

35. 最大矩形
    - # 从上到下:&运算确定最大高度
      # 从左往右:&+>>运算确定最大宽度
      class Solution:
          def maximalRectangle(self, matrix):
              if not matrix or not matrix[0]:
                  return
              # 将每一行变成二进制的数字
              # [20, 23, 31, 18]
              nums = [int(''.join(row), 2) for row in matrix]
              # ans:最大面积
              # N:行数
              res, N = 0, len(nums)
              # 遍历每一行，找以第i行为起点，可以延伸的最大高度(竖直方向都为1的地方，高度+1)
              for i in range(N):
                  cur = i   # 从当前行i开始向下延伸
                  cur_num = nums[i]
                  while cur < N:
                      # num中为1的位置，说明上下两行该位置都是1，相当于求矩阵的高，高度为j - i + 1
                      cur_num = cur_num & nums[cur]
                      # 没有1，说明没有直达i-j行的竖直矩形
                      if not cur_num:
                          break
                      width = 0
                      width_num = cur_num
                      # 已经确定了目前达到的最大高度，确定可以达到的最大宽度，也就是相邻1的个数
                      while width_num:
                          width += 1
                          width_num = width_num & (width_num >> 1)
                      # 更新最大值
                      res = max(res, width*(cur-i+1))
                      cur += 1
              return res

36. 删除无效的括号
    - BFS+对于新生成的一层字符串，检查是否有合法字符串
    - class Solution:
          def removeInvalidParentheses(self, s):
              def isvalid(string): # 判断括号串是否合法
                  count = 0
                  for c in string:
                      if c == '(':
                          count += 1
                      elif c == ')':
                          count -= 1
                          if count < 0:
                              return False
                  return count == 0
              # level:初始化第一层的字符串
              level = {s}
              while True: # BFS
                  # 从当前层中挑选出合法的字符串

```
            valid = list(filter(isvalid, level))
            if valid:
                return valid
            # 新的层中的字符串
            new_level = set()
            for s in level:
                for i in range(len(s)):
                    if s[i] in '()':
                        # 每次删除当前字符s[i]构成新的字符串
                        new_level.add(s[:i]+s[i+1:])
            # 检查新生成的一层字符串是否包含合法字符串
            level = new_level
```

37. 戳气球

```
# 动态规划
# 先给数组的首尾添加元素1
# dp[i][j]表示戳破开区间(i,j)之间的所有气球，可以获得的最大分数
# 最终的答案为dp[0][n+1]
# 对于区间(i,j):需要选择区间内最后一个戳破的气球k
# i:从下往上遍历,其实元素就是nums的最后一个元素的位置
# j:从左向右遍历,一定比i大
class Solution:
    def maxCoins(self, nums):
        # 给 nums收尾添加1
        temp = [0] * (len(nums) + 2)
        temp[0] = 1
        temp[-1] = 1
        for i in range(1, len(temp) - 1):
            temp[i] = nums[i - 1]
        # dp[i][j]表示戳破开区间 (i,j) 子数组中的所有气球，可以获得的最大分数
        dp = [[0 for _ in range(len(temp))] for _ in range(len(temp))]
        # i 从下往上
        for i in range(len(nums), -1, -1):
            # j 从左向右
            for j in range(i + 1, len(temp)):
                # 选择最后戳破的气球
                for k in range(i + 1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + temp[i] * temp[k] * temp[j])
        return dp[0][-1]
```

38. 寻找两个正序数组的中位数

- 二分查找法
- 相当于寻找合并数组的第K小元素
- 比较两数组的第k/2小元素的大小：
  - 如果arr1[k/2-1] < arr2[k/2-1]：那么至多有（k/2-1+k/2-1）= k-2个数<arr1[k/2-1]，arr[k/2-1]肯定不是第k小数，可以排除arr1[k/2-1]及其之前的数
  - 如果arr1[k/2-1] > arr2[k/2-1]：那么arr2[k/2-1]及其之前的数可以被排除
  - 如果arr1[k/2-1] = arr2[k/2-1]：处理方式与之前相同
- 更新k：k = k - k/2 (相当于每次都找到了前k/2小的数)
- Base case
  - 如果arr1为空：返回arr2第K元素arr2[k-1]
  - 如果k=1：返回两数组的首元素中的较小者

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        # 寻找arr1和arr2中第K小元素
        def findK(arr1, arr2, k):
            len1 = len(arr1)
            len2 = len(arr2)
            # arr1总是长度较短的元素，避免冗余判断
            if len1 > len2:
                return findK(arr2, arr1, k)
            # 其中一个数组为空，饭返回另一个数组的第K小元素
            if not arr1:
                return arr2[k-1]
            # k=1时：返回两数组中较小的首元素
            if k == 1:
                return min(arr1[0], arr2[0])
            # 比较arr1[k/2-1]与arr2[k/2-1],排除点较小的一部分
            idx1 = min(k//2, len1) - 1
            idx2 = min(k//2, len2) - 1
            if arr1[idx1] < arr2[idx2]:
                # 更新k = k - k/2
                return findK(arr1[idx1+1:], arr2, k-(idx1+1))
            else:
                return findK(arr1, arr2[idx2+1:], k-(idx2+1))

        l1 = len(nums1)
        l2 = len(nums2)
        mid = (l1 + l2) // 2
        # 中位数位置
        # 长度为偶数
        if  (l1 + l2) % 2 == 0:
            res = (findK(nums1, nums2, mid) + findK(nums1, nums2, mid+1)) / 2
        # 长度为奇数
        else:
```

```
            res = findK(nums1, nums2, mid+1)
        return res
```

39. 图的遍历

```
# 图的构建
# 存储为字典形式
graph={
        '1': ['2','3'],
        '2': ['4','5'],
        '3': ['6','7'],
        '4': ['8'],
        '5': ['8'],
        '6': ['7'],
        '7': [],
        '8': [],
        }
res = []
vis = []
# DFS
# 基于栈的深度优先遍历
def DFS1(graph, s):
    stack = []
    stack.append(s)
    vis = []   # 记录已经遍历过的点
    vis.append(s)
    while stack:
        cur = stack.pop()
        neibor = graph[cur]
        for item in neibor[::-1]:
            if item not in vis:
                vis.append(item)
                stack.append(item)
        res.append(cur)

# 递归的深度优先遍历
def DFS2(graph, s):
    if len(res) == len(graph):
        return
    res.append(s)
    vis.append(s)
    neibor = graph[s]
    for item in neibor:
        if item not in vis:
            DFS2(graph, item)

# 基于队列的广度优先遍历
def BFS(graph, s):
    queue = []
    queue.append(s)
    vis = []
    vis.append(s)
    while queue:
        cur = queue.pop(0)
        neibor = graph[cur]
        for item in neibor:
            if item not in vis:
                vis.append(item)
                queue.append(item)
        res.append(cur)
```