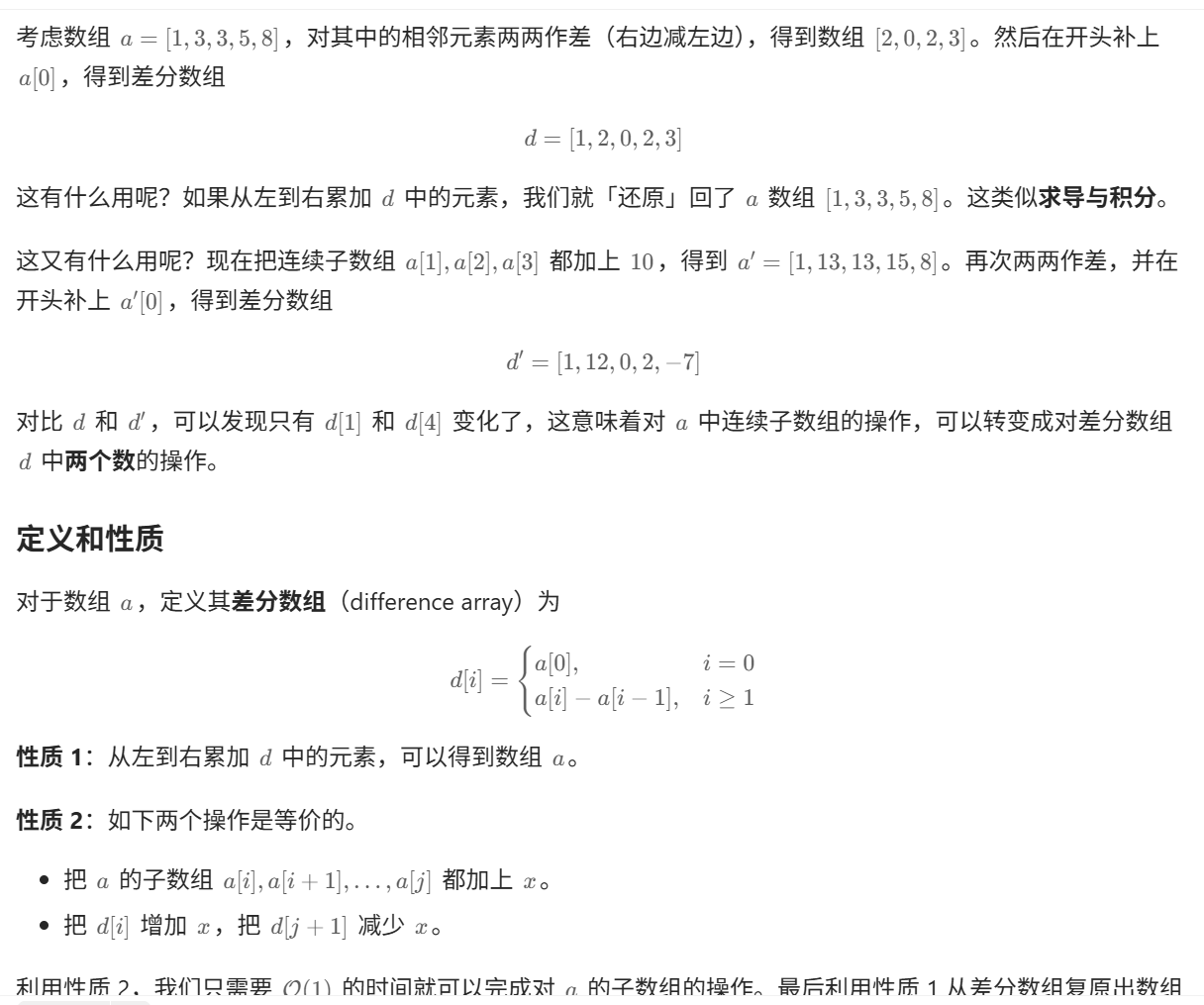
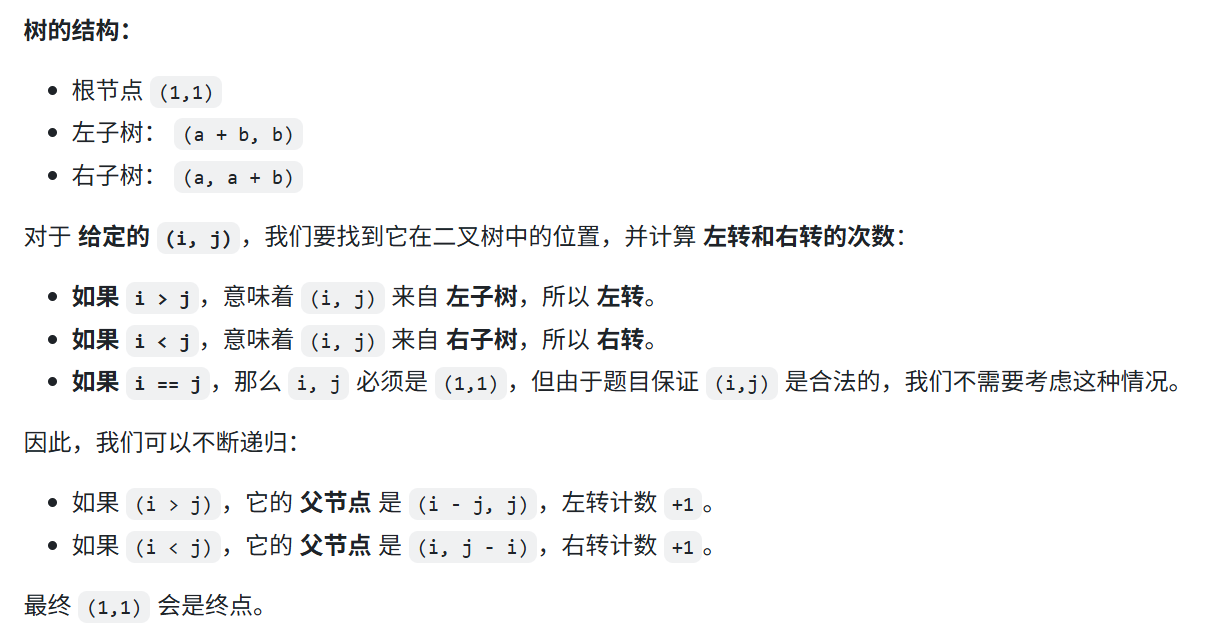
**注意：1、**质数筛法：欧拉筛

更注意：对于n大于10^7的数据，只需算出根号n + 1以内的质数，然后用n去除，**均无法整除即为质数**

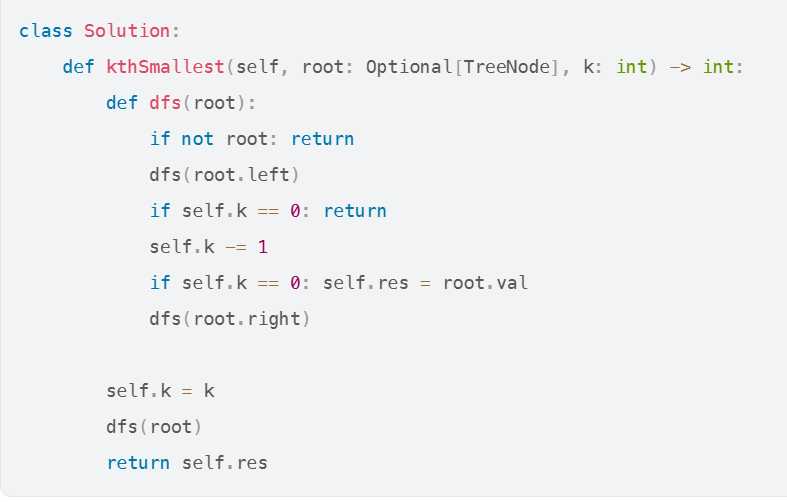
1. **善用栈的后进后出模拟中间抽出去一些元素/暂存元素**
2. **差分数组：核心即，只需要改两端的值，省时**



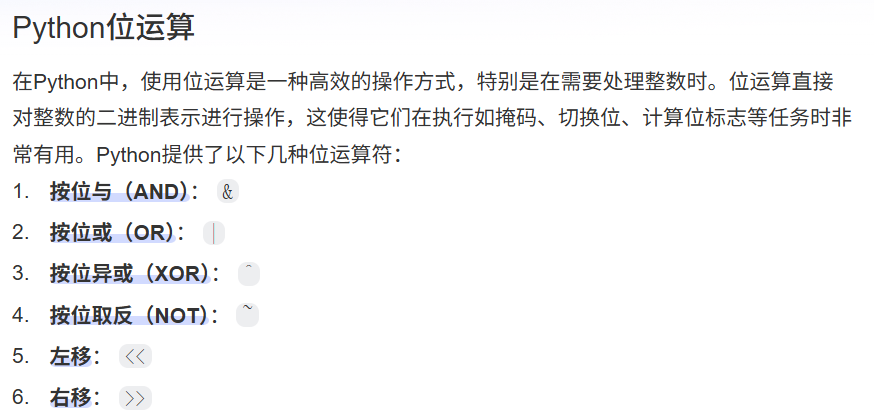
1. **贝尔曼松弛算法：增益环（好像不考？）/k次中转的最小价格（几次松弛就是最多几段路径）**
2. **水淹七军，注意后放的水币先放的水的更高的情形：即，visited列表要慎用**
3. **骑士周游，优先选下一个节点可去位置少的**
4. **边不太多的时候迪杰斯特拉也挺快的，不一定要用松弛/全节点最小距离做**
5. **很多节点找边的时候可以很多“桶”将相邻节点（具有相同特征）放在一起，只需遍历节点一次（如，单词梯那道题）**
6. **伪满二叉树：右节点和自己同级，左节点为下一级，注意观察变形之后的树和原来的树的关系**
7. **二叉树充分利用左右递归关系（如统计节点数目/前序/后序/中序等等）；以及，左右子树大小关系等等根据现有节点进行回溯：即，通过给出的特征，判断当前节点是上一节点的左还是右，不断回溯直到根节点**



1. **千万注意后序遍历是先左后右再根节点**
2. **双链表合并**
3. **在寻找最大（最长路径的时候一定注意负数权值的边）**
4. **二叉搜索树构建，考虑递归（即，比当前根节点小则考虑左节点，为空则插入，不为空则递归下一层），二叉搜索树的“落叶”在“重构”等价于按落叶顺序的倒序直接建树**
5. **公共祖先（根节点）：dfs，二者路径中最后一个相同的节点**
6. **树结合前缀和（找特定的区间和特别好用），dfs，用一个字典记录，回溯时对应的value减一**
7. **二叉搜索树验证：中序遍历形成的列表是单调的**
8. **整理文件那题见最后，有代码**
9. **中序过程中的操作**



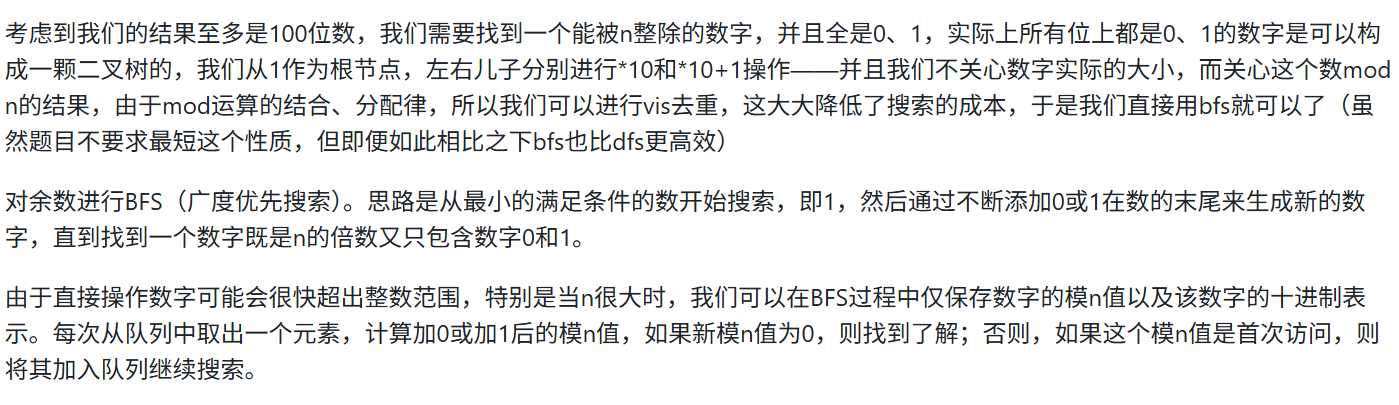
1. **位运算**





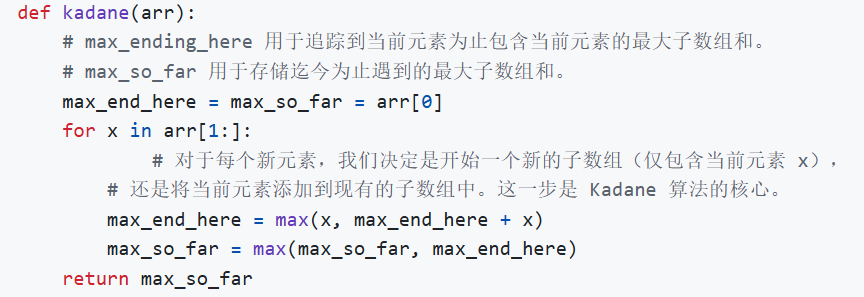
**21、跳跃游戏，贪心，还是在相同数量的桥的情况下尽可能走得远**

**22、找到一个只含01的数，是N的倍数：**

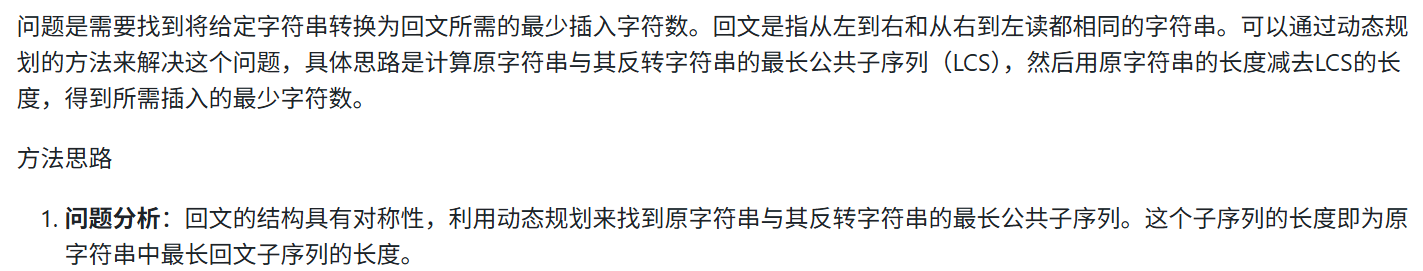


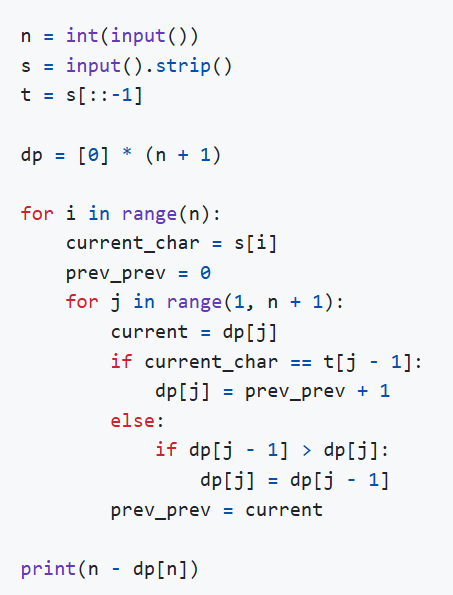


1. **最大矩形：核心想法——只在“下山”的时候讨论上一个“山峰”的最大矩形，单调栈记录“上山”过程**
2. **最大子矩阵：多维压缩为一维，然后kanada**

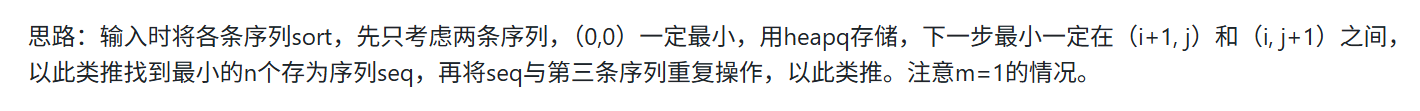


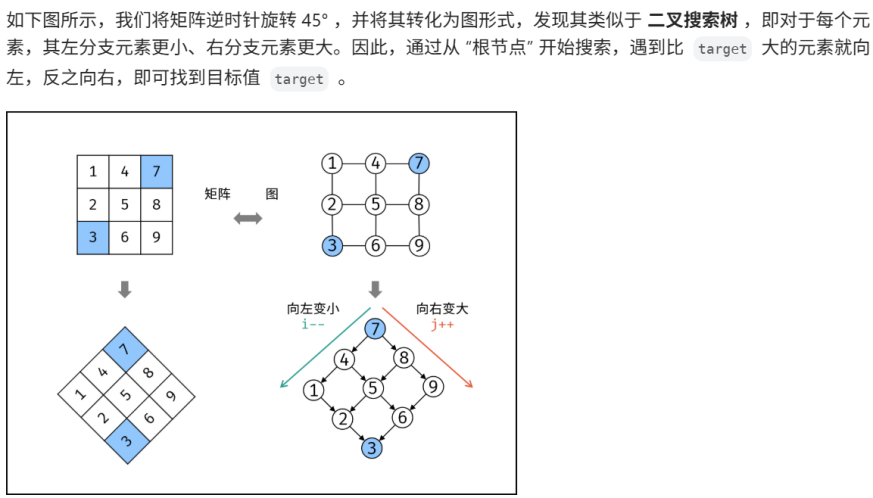
1. **dp找最长公共子串的基本功，拓展到找某序列的最长回文子序列，等价于找该序列与反转序列的最长公共子序列**



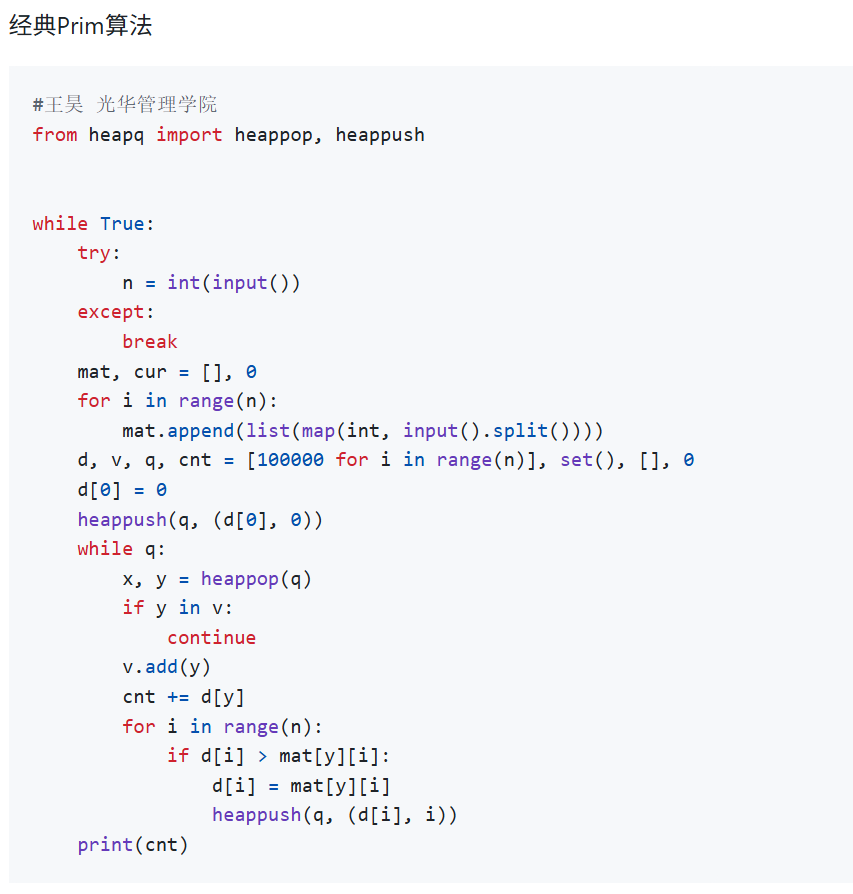


1. **直接找算法想不到的话，不妨按题目描述模拟一下，如，合法的出栈序列**





**模板：**



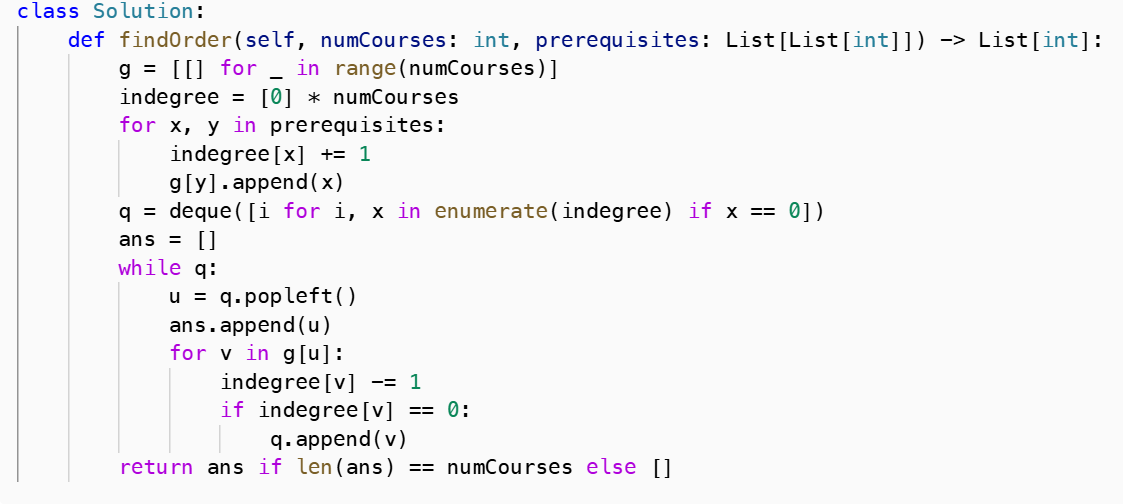
**Dp树**



**二分查找：**



**拓扑排序：**



**松弛（好像不考）**





**并查集：**



**前缀树：**





**哈夫曼编码（唯一） ：**

**import heapq**

**class TreeNode :**

**def \_\_init\_\_(self , val , weight) :**

**self.val = val**

**self.weight = weight**

**self.left = None**

**self.right = None**

**def huffman\_code(root , str , step) :**

**if root.val == str and root.left == None and root.right == None :**

**return step**

**elif root.left == None and root.right == None :**

**return ""**

**else :**

**return(huffman\_code(root.left , str , step + "0") + huffman\_code(root.right , str , step + "1"))**

**n = int(input())**

**nodes = []**

**for \_ in range(n) :**

**val , weight = input().split()**

**weight = int(weight)**

**heapq.heappush(nodes , (weight , val , TreeNode(val , weight)))**

**while len(nodes) > 1 :**

**weight1 , val1 , node1 = heapq.heappop(nodes)**

**weight2 , val2 , node2 = heapq.heappop(nodes)**

**node = TreeNode(min(val1 , val2) , weight1 + weight2)**

**node.left = node1**

**node.right = node2**

**heapq.heappush(nodes , (weight1 + weight2 , node.val , node))**

**root = nodes[0][2]**

**#print(root.val)**

**#print(root.right.weight)**

**#print(root.right.left.val)**

**while True :**

**try :**

**a = input()**

**if a[0] not in ["1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9"] :**

**answer = ""**

**for i in range(len(a)) :**

**answer += huffman\_code(root , a[i] , "")**

**print(answer)**

**else :**

**ind = 0**

**now = root**

**answer = ""**

**while ind < len(a) :**

**if now.left == None and now.right == None :**

**answer += str(now.val)**

**now = root**

**if a[ind] == "0" :**

**now = now.left**

**else :**

**now = now.right**

**ind += 1**

**answer += str(now.val)**

**print(answer)**

**except :**

**break**

**前缀和：**



**归并排序：**

**def** **hebing**(lista , listb) :

answerlist = []

answer = 0

i = 0

j = 0

**while** True :

**if** i == **len**(lista) **and** **len**(listb) == j :

**break**

**elif** j == **len**(listb) :

answerlist.**append**(lista[i])

i += 1

**elif** i == **len**(lista) :

answerlist.**append**(listb[j])

j += 1

answer += i

**elif** lista[i] < listb[j] :

answerlist.**append**(lista[i])

i += 1

**else** :

answer += i

answerlist.**append**(listb[j])

j += 1

**return** (answerlist , answer)

**def** **bingchaji**(list0) :

**if** **len**(list0) >= 2 :

a , ana = **bingchaji**(list0[:(**len**(list0) // 2)])

b , anb = **bingchaji**(list0[(**len**(list0) // 2) : **len**(list0)])

new , ans = **hebing**(a , b)

**return** new , ana + anb + ans

**else** :

**return** (list0 , 0)

n = **int**(**input**())

nums = [**int**(**input**()) **for** \_ **in** **range**(n)]

\_ , answer = **bingchaji**(nums)**print**(answer)

答案代码：

import sys

def merge\_sort(arr, temp, left, right):

if left >= right:

return 0

mid = (left + right) // 2

inv\_count = merge\_sort(arr, temp, left, mid) + merge\_sort(arr, temp, mid + 1, right)

# 归并过程，同时计算逆序数

i, j, k = left, mid + 1, left

while i <= mid and j <= right:

if arr[i] >= arr[j]: # 注意这里是 >=，保证稳定性

temp[k] = arr[i]

i += 1

else:

temp[k] = arr[j]

inv\_count += (mid - i + 1) # 统计逆序对

j += 1

k += 1

while i <= mid:

temp[k] = arr[i]

i += 1

k += 1

while j <= right:

temp[k] = arr[j]

j += 1

k += 1

# 拷贝回原数组

for i in range(left, right + 1):

arr[i] = temp[i]

return inv\_count

if \_\_name\_\_ == "\_\_main\_\_":

n = int(sys.stdin.readline().strip())

arr = [int(sys.stdin.readline().strip()) for \_ in range(n)]

temp = [0] \* n

result = merge\_sort(arr, temp, 0, n - 1)

print(result)

**解数独：**

class Solution:

def solveSudoku(self, board: List[List[str]]) -> None:

"""

Do not return anything, modify board in-place instead.

"""

self.backtracking(board)

def backtracking(self, board: List[List[str]]) -> bool:

# 若有解，返回True；若无解，返回False

for i in range(len(board)): # 遍历行

for j in range(len(board[0])): # 遍历列

# 若空格内已有数字，跳过

if board[i][j] != '.': continue

for k in range(1, 10):

if self.is\_valid(i, j, k, board):

board[i][j] = str(k)

if self.backtracking(board): return True

board[i][j] = '.'

# 若数字1-9都不能成功填入空格，返回False无解

return False

return True # 有解

def is\_valid(self, row: int, col: int, val: int, board: List[List[str]]) -> bool:

# 判断同一行是否冲突

for i in range(9):

if board[row][i] == str(val):

return False

# 判断同一列是否冲突

for j in range(9):

if board[j][col] == str(val):

return False

# 判断同一九宫格是否有冲突

start\_row = (row // 3) \* 3

start\_col = (col // 3) \* 3

for i in range(start\_row, start\_row + 3):

for j in range(start\_col, start\_col + 3):

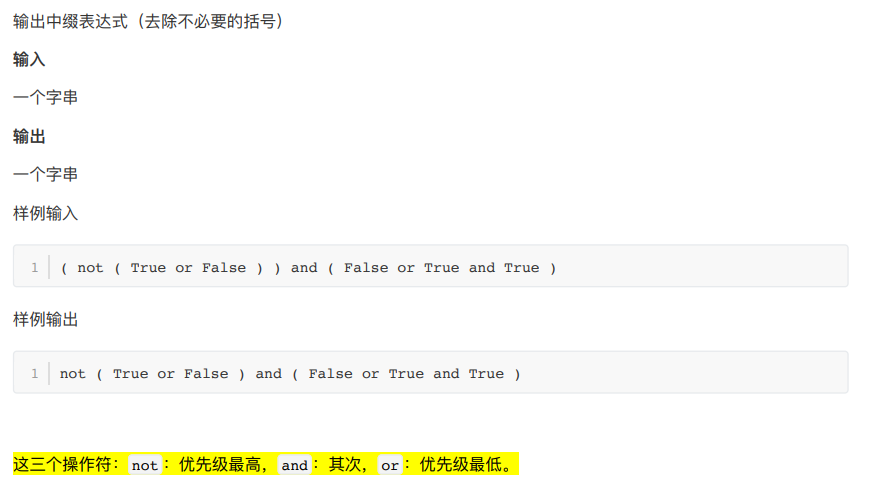
if board[i][j] == str(val):

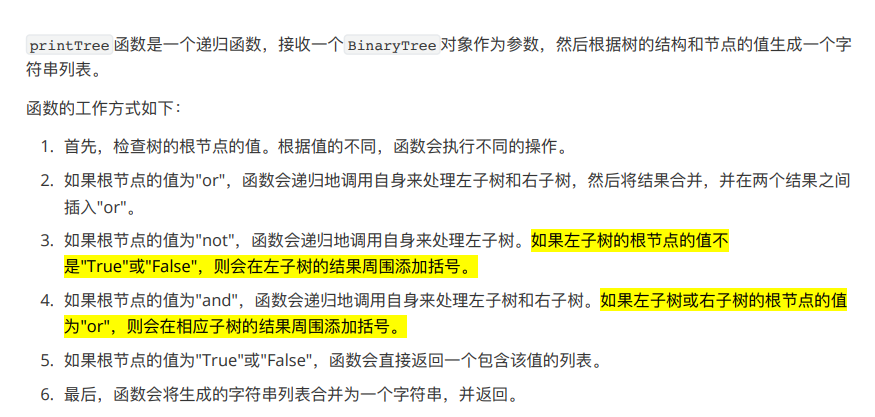
return False

return True

练习T20576: printExp（逆波兰表达式建树）

http://cs101.openjudge.cn/practice/20576/





class BinaryTree:

def \_\_init\_\_(self, root, left=None, right=None):

self.root = root

self.leftChild = left

self.rightChild = right

def getrightchild(self):

return self.rightChild

def getleftchild(self):

return self.leftChild

def getroot(self):

return self.root

def postorder(string): #中缀改后缀 Shunting yard algorightm

opStack = []

postList = []

inList = string.split()

prec = { '(': 0, 'or': 1,'and': 2,'not': 3}

for word in inList:

if word == '(':

opStack.append(word)

elif word == ')':

topWord = opStack.pop()

while topWord != '(':

postList.append(topWord)

topWord = opStack.pop()

elif word == 'True' or word == 'False':

postList.append(word)

else:

while opStack and prec[word] <= prec[opStack[-1]]:

postList.append(opStack.pop())

opStack.append(word)

while opStack:

postList.append(opStack.pop())

return postList

def buildParseTree(infix): #以后缀表达式为基础建树

postList = postorder(infix)

stack = []

for word in postList:

if word == 'not':

newTree = BinaryTree(word)

newTree.leftChild = stack.pop()

stack.append(newTree)

elif word == 'True' or word == 'False':

stack.append(BinaryTree(word))

else:

right = stack.pop()

left = stack.pop()

newTree = BinaryTree(word)

newTree.leftChild = left

newTree.rightChild = right

stack.append(newTree)

currentTree = stack[-1]

return currentTree

def printTree(parsetree: BinaryTree):

if parsetree.getroot() == 'or':

return printTree(parsetree.getleftchild()) + ['or'] + printTree(parsetree.getrightchild())

elif parsetree.getroot() == 'not':

return ['not'] + (

['('] + printTree(parsetree.getleftchild()) + [')']

if parsetree.leftChild.getroot() not in ['True', 'False']

else printTree(parsetree.getleftchild())

)

elif parsetree.getroot() == 'and':

leftpart = (

['('] + printTree(parsetree.getleftchild()) + [')']

if parsetree.leftChild.getroot() == 'or'

else printTree(parsetree.getleftchild())

)

rightpart = (

['('] + printTree(parsetree.getrightchild()) + [')']

if parsetree.rightChild.getroot() == 'or'

else printTree(parsetree.getrightchild())

)

return leftpart + ['and'] + rightpart

else:

return [str(parsetree.getroot())]

def main():

infix = input()

Tree = buildParseTree(infix)

print(' '.join(printTree(Tree)))

main()

**整理文件：**

# 夏天明，元培学院

from sys import exit

class dir:

def \_\_init\_\_(self, dname):

self.name = dname

self.dirs = []

self.files = []

def getGraph(self):

g = [self.name]

for d in self.dirs:

subg = d.getGraph()

g.extend(["| " + s for s in subg])

for f in sorted(self.files):

g.append(f)

return g

n = 0

while True:

n += 1

stack = [dir("ROOT")]

while (s := input()) != "\*":

if s == "#": exit(0)

if s[0] == 'f':

stack[-1].files.append(s)

elif s[0] == 'd':

stack.append(dir(s))

stack[-2].dirs.append(stack[-1])

else:

stack.pop()

print(f"DATA SET {n}:")

print(\*stack[0].getGraph(), sep='\n')

print()

**2048：**

# pylint: skip-file

def move\_left(board):

m, n = len(board), len(board[0])

new\_board = []

for row in board:

# 压缩：去除0，保留非0数值

filtered = [x for x in row if x != 0]

merged = []

skip = False

i = 0

while i < len(filtered):

if i + 1 < len(filtered) and filtered[i] == filtered[i + 1]:

# 合并，注意每行内只允许合并一次

merged.append(filtered[i] \* 2)

i += 2

else:

merged.append(filtered[i])

i += 1

# 补0到尾部

merged += [0] \* (n - len(merged))

new\_board.append(merged)

return new\_board

def reverse\_board(board):

# 将每一行反转（用于模拟向右移动）

return [row[::-1] for row in board]

def transpose(board):

return [list(x) for x in zip(\*board)]

def move\_right(board):

# 向右移动：先反转->左移->再反转

reversed\_board = reverse\_board(board)

moved = move\_left(reversed\_board)

return reverse\_board(moved)

def move\_up(board):

# 向上移动：转置->左移->再转置

trans = transpose(board)

moved = move\_left(trans)

return transpose(moved)

def move\_down(board):

# 向下移动：转置->右移->再转置

trans = transpose(board)

moved = move\_right(trans)

return transpose(moved)

def get\_max\_tile(board):

return max(max(row) for row in board)

def dfs(board, moves\_left):

global answer

current\_max = get\_max\_tile(board)

answer = max(answer, current\_max)

if moves\_left == 0:

return

# 对四个方向进行移动

for move\_func in [move\_left, move\_right, move\_up, move\_down]:

new\_board = move\_func(board)

# 若该操作没有产生变化，则无需再搜索

if new\_board == board:

continue

dfs(new\_board, moves\_left - 1)

if \_\_name\_\_ == '\_\_main\_\_':

m, n, p = map(int, input().split())

board = []

for i in range(m):

row = list(map(int, input().split()))

board.append(row)

answer = 0

dfs(board, p)

print(answer)

并查集

归并排序

**4.2 强连通单元（SCCs）**

**Tarjan 算法**

Kosaraju算法

**Bellman-Ford 算法**

**T01860: Currency Exchange**

