A1

**20742：斐波那契数**

def tribonacci(n):

if n == 0:

return 0

elif n <= 2:

return 1

trib = [0, 1, 1] + [0] \* (n - 2)

for i in range(3, n + 1):

trib[i] = trib[i - 1] + trib[i - 2] + trib[i - 3]

return trib[n]

n = int(input())

print(tribonacci(n))

**24684：直播计票**

from collections import defaultdict

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

vote\_counts = defaultdict(int)

for vote in votes:

vote\_counts[vote] += 1

max\_votes = max(vote\_counts.values())

lst = [item for item in vote\_counts.items() if item[1] == max\_votes]

winners = sorted(lst)

print(' '.join(str(winner[0]) for winner in winners))

A2

**27653：Fraction类**

def gcd(m, n):

while m % n != 0:

oldm = m

oldn = n

m = oldn

n = oldm % oldn

return n

class Fraction:

def \_\_init\_\_(self, top, bottom):

self.num = top

self.den = bottom

def \_\_str\_\_(self):

return str(self.num) + "/" + str(self.den)

def show(self):

print(self.num, "/", self.den)

def \_\_add\_\_(self, otherfraction):

newnum = self.num \* otherfraction.den + self.den \* otherfraction.num

newden = self.den \* otherfraction.den

common = gcd(newnum, newden)

return Fraction(newnum // common, newden // common)

a, b, c, d = map(int, input().split())

x = Fraction(a, b)

y = Fraction(c, d)

print(x+y)

**04110:圣诞老人的礼物**

Sum = 0

dic = {}

for i in range(n):

v, w = [int(x) for x in input().split()]

xjb = round(v / w, 1)

if xjb not in dic:

dic[xjb] = [w]

else:

dic[xjb] = [int(t) + w for t in dic[xjb]]

dic = sorted(dic.items(), reverse=True)

for i in range(len(dic)):

if m > sum(dic[i][1]):

Sum = Sum + sum(dic[i][1]) \* dic[i][0]

m -= sum(dic[i][1])

else:

Sum = Sum + dic[i][0] \* m

m = 0

if m == 0:

break

print(Sum)

**18182:打怪兽**

cases = int(input())

for i in range(cases):

situation = "alive"

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

a = {}

for i in range(n):

x, y = map(int, input().split())

if x not in a:

a[x] = [y]

else:

a[x].append(y)

c = sorted(a)

for i in c:

if m >= len(a[i]):

b -= sum(a[i])

else:

a[i] = sorted(a[i], reverse=True)

b -= sum(a[i][:m])

if b <= 0:

situation = i

break

print(situation)

**18176:2050年成绩计算**

from math import sqrt

N = 10000

s = [True] \* N

p = 2

while p \* p <= N:

if s[p]:

for i in range(p \* 2, N, p):

s[i] = False

p += 1

m, n = [int(i) for i in input().split()]

for i in range(m):

x = [int(i) for i in input().split()]

sum = 0

for num in x:

root = int(sqrt(num))

if num > 3 and s[root] and num == root \* root:

sum += num

sum /= len(x)

if sum == 0:

print(0)

else:

print('%.2f' % sum)

A3

**02945:拦截导弹**

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

dp = [0] \* k

for i in range(k - 1, -1, -1):

maxn = 1

for j in range(k - 1, i, -1):

if l[i] >= l[j] and dp[j] + 1 > maxn:

maxn = dp[j] + 1

dp[i] = maxn

print(max(dp))

**04147:汉诺塔问题**

def moveOne(numDisk: int, init: str, desti: str):

print("{}:{}->{}".format(numDisk, init, desti))

def move(numDisks: int, init: str, temp: str, desti: str):

if numDisks == 1:

moveOne(1, init, desti)

else:

move(numDisks - 1, init, desti, temp)

moveOne(numDisks, init, desti)

move(numDisks - 1, temp, init, desti)

n, a, b, c = input().split()

move(int(n), a, b, c)

**03253:约瑟夫问题2**

while True:

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

if {n,p,m} == {0}:

break

monkey = [i for i in range(1, n+1)]

for \_ in range(p-1):

tmp = monkey.pop(0)

monkey.append(tmp)

index = 0

ans = []

while len(monkey) != 1:

temp = monkey.pop(0)

index += 1

if index == m:

index = 0

ans.append(temp)

continue

monkey.append(temp)

ans.extend(monkey)

print(','.join(map(str, ans)))

**21554:排队做实验**

n = int(input())

\*L, = map(int,input().split())

od = sorted(range(1, n+1), key = lambda x: L[x - 1])

L.sort()

t = sum((n-i-1) \* L[i] for i in range(n)) / n

print(\*od)

print('{:.2f}'.format(t))

**19963:买学区房**

n = int(input())

dis = [eval(x)[0] + eval(x)[1] for x in input().split()]

pri = [int(x) for x in input().split()]

vau = [dis[x] / pri[x] for x in range(n)]

def mid(n, lis):

lis = sorted(lis)

if n % 2 == 1:

return lis[n // 2]

else:

return (lis[n // 2 - 1] + lis[n // 2]) / 2

prim = mid(n, pri)

vaum = mid(n, vau)

sum = 0

for i in range(n):

if vau[i] > vaum and pri[i] < prim:

sum += 1

print(sum)

**27300:模型整理**

from collections import defaultdict

n = int(input())

d = defaultdict(list)

for \_ in range(n):

name, para = input().split('-')

if para[-1] == 'M':

d[name].append((para, float(para[:-1]) / 1000))

else:

d[name].append((para, float(para[:-1])))

sd = sorted(d)

for k in sd:

paras = sorted(d[k], key=lambda x: x[1])

value = ', '.join([i[0] for i in paras])

print(f'{k}: {value}')

A4

**05902：双端队列**

from collections import deque

n = int(input())

for i in range(n):

m = int(input())

d = deque()

for j in range(m):

(t, c) = [int(x) for x in input().split()]

if t == 1:

d.append(c)

elif t == 2:

if c == 0:

d.popleft()

elif c == 1:

d.pop()

if len(d) == 0:

print("NULL")

else:

print(' '.join(map(str, d)))

**02694：波兰表达式**stack\_list = []  
stack\_store = []  
input\_str = input().split(' ')  
for char in input\_str:  
 stack\_list.append(char)  
while len(stack\_list) != 0:  
 a = stack\_list.pop()  
 if a == '+':  
 b = float(stack\_store.pop())  
 c = float(stack\_store.pop())  
 d = float(c + b)  
 stack\_store.append(d)  
 elif a == '-':  
 b = float(stack\_store.pop())  
 c = float(stack\_store.pop())  
 d = float(b - c)  
 stack\_store.append(d)  
 elif a == '\*':  
 b = float(stack\_store.pop())  
 c = float(stack\_store.pop())  
 d = float(c \* b)  
 stack\_store.append(d)  
 elif a == '/':  
 b = float(stack\_store.pop())  
 c = float(stack\_store.pop())  
 d = float(b / c)  
 stack\_store.append(d)  
 else:  
 stack\_store.append(a)  
 print(stack\_store)  
e = "{:.6f}".format(stack\_store.pop())

print(e)

**24591：中序表达式转后续表达式**

def infix\_to\_postfix(expression):

precedence = {'+': 1, '-': 1, '\*': 2, '/': 2}

stack = []

postfix = []

number = ''

for char in expression:

if char.isnumeric() or char == '.':

number += char

else:

if number:

num = float(number)

postfix.append(int(num) if num.is\_integer() else num)

number = ''

if char in '+-\*/':

while stack and stack[-1] in '+-\*/' and precedence[char] <= precedence[stack[-1]]:

postfix.append(stack.pop())

stack.append(char)

elif char == '(':

stack.append(char)

elif char == ')':

while stack and stack[-1] != '(':

postfix.append(stack.pop())

stack.pop()

if number:

num = float(number)

postfix.append(int(num) if num.is\_integer() else num)

while stack:

postfix.append(stack.pop())

return ' '.join(str(x) for x in postfix)

n = int(input())

for \_ in range(n):

expression = input()

print(infix\_to\_postfix(expression))

**22068:合法出栈序列**

def is\_valid\_stack\_pop\_sequence(origin, output):

if len(origin) != len(output):

return False

stack = []

bank = list(origin)

for char in output:

while (not stack or stack[-1] != char) and bank:

stack.append(bank.pop(0))

if not stack or stack[-1] != char:

return False

stack.pop()

return True

origin = input().strip()

while True:

try:

output = input().strip()

if is\_valid\_stack\_pop\_sequence(origin, output):

print('YES')

else:

print('NO')

except EOFError:

break

**06646:二叉树的深度**

class TreeNode:

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

self.left = None

self.right = None

def tree\_depth(node):

if node is None:

return 0

left\_depth = tree\_depth(node.left)

right\_depth = tree\_depth(node.right)

return max(left\_depth, right\_depth) + 1

n = int(input())

nodes = [TreeNode() for \_ in range(n)]

for i in range(n):

left\_index, right\_index = map(int, input().split())

if left\_index != -1:

nodes[i].left = nodes[left\_index - 1]

if right\_index != -1:

nodes[i].right = nodes[right\_index - 1]

root = nodes[0]

depth = tree\_depth(root)

print(depth)

**02299：Ultra-QuickSort**

def merge\_sort(lst):

if len(lst) <= 1:

return lst, 0

middle = len(lst) // 2

left, inv\_left = merge\_sort(lst[:middle])

right, inv\_right = merge\_sort(lst[middle:])

merged, inv\_merge = merge(left, right)

return merged, inv\_left + inv\_right + inv\_merge

def merge(left, right):

merged = []

inv\_count = 0

i = j = 0

while i < len(left) and j < len(right):

if left[i] <= right[j]:

merged.append(left[i])

i += 1

else:

merged.append(right[j])

j += 1

inv\_count += len(left) - i

merged += left[i:]

merged += right[j:]

return merged, inv\_count

while True:

n = int(input())

if n == 0:

break

lst = []

for \_ in range(n):

lst.append(int(input()))

\_, inversions = merge\_sort(lst)

print(inversions)

A5

**27638: 求二叉树的高度和叶子数目**

class TreeNode:

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

self.left = None

self.right = None

def tree\_height(node):

if node is None:

return -1

return max(tree\_height(node.left), tree\_height(node.right)) + 1

def count\_leaves(node):

if node is None:

return 0

if node.left is None and node.right is None:

return 1

return count\_leaves(node.left) + count\_leaves(node.right)

n = int(input())

nodes = [TreeNode() for \_ in range(n)]

has\_parent = [False] \* n

for i in range(n):

left\_index, right\_index = map(int, input().split())

if left\_index != -1:

nodes[i].left = nodes[left\_index]

has\_parent[left\_index] = True

if right\_index != -1:

nodes[i].right = nodes[right\_index]

has\_parent[right\_index] = True

root\_index = has\_parent.index(False)

root = nodes[root\_index]

height = tree\_height(root)

leaves = count\_leaves(root)

print(height, leaves)

**24729:括号嵌套树**

class TreeNode:

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

self.value = value

self.children = []

def parse\_tree(s):

stack = []

node = None

for char in s:

if char.isalpha():

node = TreeNode(char)

if stack:

stack[-1].children.append(node)

elif char == '(':

if node:

stack.append(node)

node = None

elif char == ')':

if stack:

node = stack.pop()

return node

def preorder(node):

output = [node.value]

for child in node.children:

output.extend(preorder(child))

return ''.join(output)

def postorder(node):

output = []

for child in node.children:

output.extend(postorder(child))

output.append(node.value)

return ''.join(output)

def main():

s = input().strip()

s = ''.join(s.split())

root = parse\_tree(s)

if root:

print(preorder(root)) # 输出前序遍历序列

print(postorder(root)) # 输出后序遍历序列

else:

print("input tree string error!")

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

main()

**02775: 文件结构“图”**

**# 文件结构图——树结构可视化运用**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()

**25140: 根据后序表达式建立队列表达式**

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def build\_tree(postfix):

stack = []

for char in postfix:

node = TreeNode(char)

if char.isupper():

node.right = stack.pop()

node.left = stack.pop()

stack.append(node)

return stack[0]

def level\_order\_traversal(root):

queue = [root]

traversal = []

while queue:

node = queue.pop(0)

traversal.append(node.value)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

return traversal

n = int(input().strip())

for \_ in range(n):

postfix = input().strip()

root = build\_tree(postfix)

queue\_expression = level\_order\_traversal(root)[::-1]

print(''.join(queue\_expression))

**24750: 根据二叉树中后序序列建树**

def build\_tree(inorder, postorder):

if not inorder or not postorder:

return []

root\_val = postorder[-1]

root\_index = inorder.index(root\_val)

left\_inorder = inorder[:root\_index]

right\_inorder = inorder[root\_index + 1:]

left\_postorder = postorder[:len(left\_inorder)]

right\_postorder = postorder[len(left\_inorder):-1]

root = [root\_val]

root.extend(build\_tree(left\_inorder, left\_postorder))

root.extend(build\_tree(right\_inorder, right\_postorder))

return root

def main():

inorder = input().strip()

postorder = input().strip()

preorder = build\_tree(inorder, postorder)

print(''.join(preorder))

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

main()

**24750: 根据二叉树中后序序列建树**

def build\_tree(inorder, postorder):

if not inorder or not postorder:

return []

root\_val = postorder[-1]

root\_index = inorder.index(root\_val)

left\_inorder = inorder[:root\_index]

right\_inorder = inorder[root\_index + 1:]

left\_postorder = postorder[:len(left\_inorder)]

right\_postorder = postorder[len(left\_inorder):-1]

root = [root\_val]

root.extend(build\_tree(left\_inorder, left\_postorder))

root.extend(build\_tree(right\_inorder, right\_postorder))

return root

def main():

inorder = input().strip()

postorder = input().strip()

preorder = build\_tree(inorder, postorder)

print(''.join(preorder))

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

main()

**22158: 根据二叉树前中序序列建树**

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def build\_tree(preorder, inorder):

if not preorder or not inorder:

return None

root\_value = preorder[0]

root = TreeNode(root\_value)

root\_index\_inorder = inorder.index(root\_value)

root.left = build\_tree(preorder[1:1 + root\_index\_inorder], inorder[:root\_index\_inorder])

root.right = build\_tree(preorder[1 + root\_index\_inorder:], inorder[root\_index\_inorder + 1:])

return root

def postorder\_traversal(root):

if root is None:

return ''

return postorder\_traversal(root.left) + postorder\_traversal(root.right) + root.value

while True:

try:

preorder = input().strip()

inorder = input().strip()

root = build\_tree(preorder, inorder)

print(postorder\_traversal(root))

except EOFError:

Break

A6

**22275：二叉搜索树的遍历**

def post\_order(pre\_str):

if not pre\_str:

return []

root = pre\_str[0]

pre\_str\_left = [x for x in pre\_str if x < pre\_str[0]]

pre\_str\_right = [x for x in pre\_str if x > pre\_str[0]]

return post\_order(pre\_str\_left) + post\_order(pre\_str\_right) + [root]

n = int(input())

pre\_order = list(map(int, input().split()))

print(' '.join(map(str, post\_order(pre\_order))))

**05455: 二叉搜索树的层次遍历**

class TreeNode:

def \_\_init\_\_(self, val=0):

self.val = val

self.left = None

self.right = None

def insert(root, val):

if not root:

return TreeNode(val)

if val < root.val:

root.left = insert(root.left, val)

elif val > root.val:

root.right = insert(root.right, val)

return root

def level\_order\_traversal(root):

if not root:

return []

queue = [root]

res = []

while queue:

level = []

for i in range(len(queue)):

node = queue.pop(0)

level.append(node.val)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

res.append(level)

return res

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

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

root = None

for num in nums:

root = insert(root, num)

result = level\_order\_traversal(root)

output = ""

for level in result:

output += " ".join(map(str, level)) + " "

print(output.strip())

**04078: 实现堆结构**

class BinHeap:

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

self.heapList = [0]

self.currentSize = 0

def percUp(self, i):

while i // 2 > 0:

if self.heapList[i] < self.heapList[i // 2]:

tmp = self.heapList[i // 2]

self.heapList[i // 2] = self.heapList[i]

self.heapList[i] = tmp

i = i // 2

def insert(self, k):

self.heapList.append(k)

self.currentSize = self.currentSize + 1

self.percUp(self.currentSize)

def percDown(self, i):

while (i \* 2) <= self.currentSize:

mc = self.minChild(i)

if self.heapList[i] > self.heapList[mc]:

tmp = self.heapList[i]

self.heapList[i] = self.heapList[mc]

self.heapList[mc] = tmp

i = mc

def minChild(self, i):

if i \* 2 + 1 > self.currentSize:

return i \* 2

else:

if self.heapList[i \* 2] < self.heapList[i \* 2 + 1]:

return i \* 2

else:

return i \* 2 + 1

def delMin(self):

retval = self.heapList[1]

self.heapList[1] = self.heapList[self.currentSize]

self.currentSize = self.currentSize - 1

self.heapList.pop()

self.percDown(1)

return retval

def buildHeap(self, alist):

i = len(alist) // 2

self.currentSize = len(alist)

self.heapList = [0] + alist[:]

while (i > 0):

self.percDown(i)

i = i - 1

n = int(input().strip())

bh = BinHeap()

for \_ in range(n):

inp = input().strip()

if inp[0] == '1':

bh.insert(int(inp.split()[1]))

else:

print(bh.delMin())

**22161: 哈夫曼编码树**

import heapq

class Node:

def \_\_init\_\_(self, weight, char=None):

self.weight = weight

self.char = char

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

if self.weight == other.weight:

return self.char < other.char

return self.weight < other.weight

def build\_huffman\_tree(characters):

heap = []

for char, weight in characters.items():

heapq.heappush(heap, Node(weight, char))

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

merged = Node(left.weight + right.weight, min(left.char, right.char))

merged.left = left

merged.right = right

heapq.heappush(heap, merged)

return heap[0]

def encode\_huffman\_tree(root):

codes = {}

def traverse(node, code):

#if node.char:

if node.left is None and node.right is None:

codes[node.char] = code

else:

traverse(node.left, code + '0')

traverse(node.right, code + '1')

traverse(root, '')

return codes

def huffman\_encoding(codes, string):

encoded = ''

for char in string:

encoded += codes[char]

return encoded

def huffman\_decoding(root, encoded\_string):

decoded = ''

node = root

for bit in encoded\_string:

if bit == '0':

node = node.left

else:

node = node.right

if node.left is None and node.right is None:

decoded += node.char

node = root

return decoded

n = int(input())

characters = {}

for \_ in range(n):

char, weight = input().split()

characters[char] = int(weight)

huffman\_tree = build\_huffman\_tree(characters)

codes = encode\_huffman\_tree(huffman\_tree)

strings = []

while True:

try:

line = input()

strings.append(line)

except EOFError:

break

results = []

for string in strings:

if string[0] in ('0','1'):

results.append(huffman\_decoding(huffman\_tree, string))

else:

results.append(huffman\_encoding(codes, string))

for result in results:

print(result)

**晴问9.5: 平衡二叉树的建立**

class Node:

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

self.value = value

self.left = None

self.right = None

self.height = 1

class AVL:

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

self.root = None

def insert(self, value):

if not self.root:

self.root = Node(value)

else:

self.root = self.\_insert(value, self.root)

def \_insert(self, value, node):

if not node:

return Node(value)

elif value < node.value:

node.left = self.\_insert(value, node.left)

else:

node.right = self.\_insert(value, node.right)

node.height = 1 + max(self.\_get\_height(node.left), self.\_get\_height(node.right))

balance = self.\_get\_balance(node)

if balance > 1:

if value < node.left.value:

return self.\_rotate\_right(node)

else:

node.left = self.\_rotate\_left(node.left)

return self.\_rotate\_right(node)

if balance < -1:

if value > node.right.value:

return self.\_rotate\_left(node)

else:

node.right = self.\_rotate\_right(node.right)

return self.\_rotate\_left(node)

return node

def \_get\_height(self, node):

if not node:

return 0

return node.height

def \_get\_balance(self, node):

if not node:

return 0

return self.\_get\_height(node.left) - self.\_get\_height(node.right)

def \_rotate\_left(self, z):

y = z.right

T2 = y.left

y.left = z

z.right = T2

z.height = 1 + max(self.\_get\_height(z.left), self.\_get\_height(z.right))

y.height = 1 + max(self.\_get\_height(y.left), self.\_get\_height(y.right))

return y

def \_rotate\_right(self, y):

x = y.left

T2 = x.right

x.right = y

y.left = T2

y.height = 1 + max(self.\_get\_height(y.left), self.\_get\_height(y.right))

x.height = 1 + max(self.\_get\_height(x.left), self.\_get\_height(x.right))

return x

def preorder(self):

return self.\_preorder(self.root)

def \_preorder(self, node):

if not node:

return []

return [node.value] + self.\_preorder(node.left) + self.\_preorder(node.right)

n = int(input().strip())

sequence = list(map(int, input().strip().split()))

avl = AVL()

for value in sequence:

avl.insert(value)

print(' '.join(map(str, avl.preorder())))

**02524: 宗教信仰**

def init\_set(n):

return list(range(n))

def get\_father(x, father):

if father[x] != x:

father[x] = get\_father(father[x], father)

return father[x]

def join(x, y, father):

fx = get\_father(x, father)

fy = get\_father(y, father)

if fx == fy:

return

father[fx] = fy

def is\_same(x, y, father):

return get\_father(x, father) == get\_father(y, father)

def main():

case\_num = 0

while True:

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

if n == 0 and m == 0:

break

count = 0

father = init\_set(n)

for \_ in range(m):

s1, s2 = map(int, input().split())

join(s1 - 1, s2 - 1, father)

for i in range(n):

if father[i] == i:

count += 1

case\_num += 1

print(f"Case {case\_num}: {count}")

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

main()

A7

**27706: 逐词倒放**

sentence = input()

words = sentence.split(' ')

reversed\_words = words[::-1]

result = ' '.join(reversed\_words)

print(result)

2**7951: 机器翻译**

from collections import deque

store = deque()

M, N = input().split()

find = 0

book = [x for x in input().split()]

for \_ in book:

if \_ not in store and len(store) < int(M):

store.append(\_)

find += 1

elif \_ not in store and len(store) >= int(M):

store.popleft()

store.append(\_)

find += 1

print(find)

**27932: Less or Equal**

n, k = map(int, input().split())

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

a.sort()

if k == 0:

x = 1 if a[0] > 1 else -1

elif k == n:

x = a[-1]

else:

x = a[k - 1] if a[k - 1] < a[k] else -1

print(x)

**27948: FBI树**

def construct\_FBI\_tree(s):

if '0' in s and '1' in s:

node\_type = 'F'

elif '1' in s:

node\_type = 'I'

else:

node\_type = 'B'

if len(s) > 1:

mid = len(s) // 2

left\_tree = construct\_FBI\_tree(s[:mid])

right\_tree = construct\_FBI\_tree(s[mid:])

return left\_tree + right\_tree + node\_type

else:

return node\_type

N = int(input())

s = input()

print(construct\_FBI\_tree(s))

**27925: 小组队列**

from collections import deque

t = int(input())

groups = {}

member\_to\_group = {}

for \_ in range(t):

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

group\_id = members[0] # Assuming the first member's ID represents the group ID

groups[group\_id] = deque()

for member in members:

member\_to\_group[member] = group\_id

queue = deque()

queue\_set = set()

while True:

command = input().split()

if command[0] == 'STOP':

break

elif command[0] == 'ENQUEUE':

x = int(command[1])

group = member\_to\_group.get(x, None)

if group is None:

group = x

groups[group] = deque([x])

member\_to\_group[x] = group

else:

groups[group].append(x)

if group not in queue\_set:

queue.append(group)

queue\_set.add(group)

elif command[0] == 'DEQUEUE':

if queue:

group = queue[0]

x = groups[group].popleft()

print(x)

if not groups[group]:

queue.popleft()

queue\_set.remove(group)

**27928: 遍历树**

class TreeNode:

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

self.value = value

self.children = []

def traverse\_print(root, nodes):

if root.children == []:

print(root.value)

return

pac = {root.value: root}

for child in root.children:

pac[child] = nodes[child]

for value in sorted(pac.keys()):

if value in root.children:

traverse\_print(pac[value], nodes)

else:

print(root.value)

n = int(input())

nodes = {}

children\_list = []

for i in range(n):

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

nodes[info[0]] = TreeNode(info[0])

for child\_value in info[1:]:

nodes[info[0]].children.append(child\_value)

children\_list.append(child\_value)

root = nodes[[value for value in nodes.keys() if value not in children\_list][0]]

traverse\_print(root, nodes)

**19943: 图的拉普拉斯矩阵**

class Vertex:

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

self.id = key

self.connectedTo = {}

def addNeighbor(self, nbr, weight=0):

self.connectedTo[nbr] = weight

def \_\_str\_\_(self):

return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

def getConnections(self):

return self.connectedTo.keys()

def getId(self):

return self.id

def getWeight(self, nbr):

return self.connectedTo[nbr]

class Graph:

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

self.vertList = {}

self.numVertices = 0

def addVertex(self, key):

self.numVertices = self.numVertices + 1

newVertex = Vertex(key)

self.vertList[key] = newVertex

return newVertex

def getVertex(self, n):

if n in self.vertList:

return self.vertList[n]

else:

return None

def \_\_contains\_\_(self, n):

return n in self.vertList

def addEdge(self, f, t, weight=0):

if f not in self.vertList:

nv = self.addVertex(f)

if t not in self.vertList:

nv = self.addVertex(t)

self.vertList[f].addNeighbor(self.vertList[t], weight)

def getVertices(self):

return self.vertList.keys()

def \_\_iter\_\_(self):

return iter(self.vertList.values())

def constructLaplacianMatrix(n, edges):

graph = Graph()

for i in range(n): # 添加顶点

graph.addVertex(i)

for edge in edges: # 添加边

a, b = edge

graph.addEdge(a, b)

graph.addEdge(b, a)

laplacianMatrix = [] # 构建拉普拉斯矩阵

for vertex in graph:

row = [0] \* n

row[vertex.getId()] = len(vertex.getConnections())

for neighbor in vertex.getConnections():

row[neighbor.getId()] = -1

laplacianMatrix.append(row)

return laplacianMatrix

n, m = map(int, input().split()) # 解析输入

edges = []

for i in range(m):

a, b = map(int, input().split())

edges.append((a, b))

laplacianMatrix = constructLaplacianMatrix(n, edges)

for row in laplacianMatrix: # 输出结果

print(' '.join(map(str, row)))  
**18160: 最大连通域面积**

count = 0

def dfs(x, y):

if M[x + 1][y + 1] == "W":

global count

count += 1

M[x + 1][y + 1] = "."

for i in range(8):

dfs(x+d[i][0],y+d[i][1])

T = int(input())

d = [[-1, -1], [-1, 0], [-1, 1],\

[0, -1], [0, 1],\

[1, -1], [1, 0], [1, 1]]

for i in range(T):

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

M = [["." for \_ in range(m + 2)] for \_ in range(n + 2)]

for i in range(n):

string = input()

for j in range(m):

M[i + 1][j + 1] = string[j]

ans = 0

for i in range(n):

for j in range(m):

if M[i + 1][j + 1] == "W":

dfs(i, j)

ans = max(ans, count)

count = 0

print(ans)

**sy383: 最大权值连通块**

def max\_weight(n, m, weights, edges):

graph = [[] for \_ in range(n)]

for u, v in edges:

graph[u].append(v)

graph[v].append(u)

visited = [False] \* n

max\_weight = 0

def dfs(node):

visited[node] = True

total\_weight = weights[node]

for neighbor in graph[node]:

if not visited[neighbor]:

total\_weight += dfs(neighbor)

return total\_weight

for i in range(n):

if not visited[i]:

max\_weight = max(max\_weight, dfs(i))

return max\_weight

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

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

edges = []

for \_ in range(m):

u, v = map(int, input().split())

edges.append((u, v))

print(max\_weight(n, m, weights, edges))

**03441: 4 Values whose Sum is 0**

from collections import Counter

from itertools import product

A, B, C, D = [], [], [], []

for i in range(int(input())):

a, b, c, d = map(int, input().split())

A.append(a)

B.append(b)

C.append(c)

D.append(d)

ab\_sum\_counter = Counter(map(sum, product(A, B)))

cn = 0

for cd\_sum in map(sum, product(C, D)):

cn += ab\_sum\_counter.get(-cd\_sum, 0)

print(cn)

**04089: 电话号码**

class TrieNode:

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

self.child={}

class Trie:

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

self.root = TrieNode()

def insert(self, nums):

curnode = self.root

for x in nums:

if x not in curnode.child:

curnode.child[x] = TrieNode()

curnode=curnode.child[x]

def search(self, num):

curnode = self.root

for x in num:

if x not in curnode.child:

return 0

curnode = curnode.child[x]

return 1

t = int(input())

p = []

for \_ in range(t):

n = int(input())

nums = []

for \_ in range(n):

nums.append(str(input()))

nums.sort(reverse=True)

s = 0

trie = Trie()

for num in nums:

s += trie.search(num)

trie.insert(num)

if s > 0:

print('NO')

else:

print('YES')

**04082: 树的镜面映射**

from collections import deque

class Node:

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

self.name = name

self.children = []

def create\_node():

return Node('')

def build\_tree(line, index):

node = create\_node()

fullname = line[index]

node.name = fullname[0]

if fullname[1] == '0' and node.name != '$':

index += 1

child, index = build\_tree(line, index)

node.children += child,

index += 1

child, index = build\_tree(line, index)

node.children += child,

return node, index

def print\_tree(root):

queue, stack = deque(), deque()

while root is not None:

if root.name != '$':

stack += root,

root = root.children[1] if len(root.children) > 1 else None

while stack:

queue.append(stack.pop())

while queue:

root = queue.popleft()

print(root.name, end=' ')

if root.children:

root = root.children[0]

while root is not None:

if root.name != '$':

stack += root,

root = root.children[1] if len(root.children) > 1 else None

while stack:

queue.append(stack.pop())

n = int(input())

line = input().split()

root, \_ = build\_tree(line, 0)

print\_tree(root)

A9

**04081: 树的转换**

def tree\_height(s):

old\_height = 0

max\_old = 0

new\_height = 0

max\_new = 0

stack = []

for c in s:

if c == "d":

old\_height += 1

max\_old = max(max\_old, old\_height)

new\_height += 1

stack.append(new\_height)

max\_new = max(max\_new, new\_height)

else:

old\_height -= 1

new\_height = stack[-1]

stack.pop()

return f"{max\_old} => {max\_new}"

s = input().strip()

print(tree\_height(s))

**08581: 扩展二叉树**

def build\_tree(preorder):

if not preorder or preorder[0] == '.':

return None, preorder[1:]

root = preorder[0]

left, preorder = build\_tree(preorder[1:])

right, preorder = build\_tree(preorder)

return (root, left, right), preorder

def inorder(tree):

if tree is None:

return ''

root, left, right = tree

return inorder(left) + root + inorder(right)

def postorder(tree):

if tree is None:

return ''

root, left, right = tree

return postorder(left) + postorder(right) + root

preorder = input().strip()

tree, \_ = build\_tree(preorder)

print(inorder(tree))

print(postorder(tree))

**22067: 快速堆猪**

pig\_stack = []

m = []

while True:

try:

s = input().split()

if s[0] == 'pop':

if pig\_stack:

pig\_stack.pop()

if m:

m.pop()

elif s[0] == 'min':

if m:

print(m[-1])

else:

h = int(s[1])

pig\_stack.append(h)

if not m:

m.append(h)

else:

k = m[-1]

m.append(min(k,h))

except EOFError:

break

**04123: 马走日**

s = int(input())

for \_ in range(s):

m, n, a, b = map(int, input().split())

dirs = [(2, 1), (1, 2), (-2, 1), (-1, 2), (2, -1), (1, -2), (-1, -2), (-2, -1)]

borad = [[0] \* n for x in range(m)]

cut = 0

def move(x, y, step):

global cut

if step == n \* m:

cut += 1

return

for i in range(8):

move\_x = x + dirs[i][0] # 表示列表中第i个元组的第一个元素

move\_y = y + dirs[i][1]

if move\_x >= 0 and move\_x <= m - 1 and move\_y >= 0 and move\_y <= n - 1 and borad[move\_x][move\_y] == 0:

borad[move\_x][move\_y] = 1

move(move\_x, move\_y, step + 1)

borad[move\_x][move\_y] = 0

borad[a][b] = 1

move(a, b, 1)

print(cut)

**28046: 词梯**

#

import sys

from collections import deque

class Graph:

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

self.vertices = {}

self.num\_vertices = 0

def add\_vertex(self, key):

self.num\_vertices = self.num\_vertices + 1

new\_vertex = Vertex(key)

self.vertices[key] = new\_vertex

return new\_vertex

def get\_vertex(self, n):

if n in self.vertices:

return self.vertices[n]

else:

return None

def \_\_len\_\_(self):

return self.num\_vertices

def \_\_contains\_\_(self, n):

return n in self.vertices

def add\_edge(self, f, t, cost=0):

if f not in self.vertices:

nv = self.add\_vertex(f)

if t not in self.vertices:

nv = self.add\_vertex(t)

self.vertices[f].add\_neighbor(self.vertices[t], cost)

def get\_vertices(self):

return list(self.vertices.keys())

def \_\_iter\_\_(self):

return iter(self.vertices.values())

class Vertex:

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

self.key = num

self.connectedTo = {}

self.color = 'white'

self.distance = sys.maxsize

self.previous = None

self.disc = 0

self.fin = 0

def add\_neighbor(self, nbr, weight=0):

self.connectedTo[nbr] = weight

def get\_neighbors(self):

return self.connectedTo.keys()

def build\_graph(all\_words):

buckets = {}

the\_graph = Graph()

for line in all\_words:

word = line.strip()

for i, \_ in enumerate(word):

bucket = f"{word[:i]}\_{word[i + 1:]}"

buckets.setdefault(bucket, set()).add(word)

for similar\_words in buckets.values():

for word1 in similar\_words:

for word2 in similar\_words - {word1}:

the\_graph.add\_edge(word1, word2)

return the\_graph

def bfs(start, end):

start.distnce = 0

start.previous = None

vert\_queue = deque()

vert\_queue.append(start)

while len(vert\_queue) > 0:

current = vert\_queue.popleft() # 取队首作为当前顶点

if current == end:

return True

for neighbor in current.get\_neighbors(): # 遍历当前顶点的邻接顶点

if neighbor.color == "white":

neighbor.color = "gray"

neighbor.distance = current.distance + 1

neighbor.previous = current

vert\_queue.append(neighbor)

current.color = "black" # 当前顶点已经处理完毕，设黑色

return False

def traverse(starting\_vertex):

ans = []

current = starting\_vertex

while (current.previous):

ans.append(current.key)

current = current.previous

ans.append(current.key)

return ans

n = int(input())

all\_words = []

for \_ in range(n):

all\_words.append(input().strip())

g = build\_graph(all\_words)

# print(len(g))

s, e = input().split()

start, end = g.get\_vertex(s), g.get\_vertex(e)

if start is None or end is None:

print('NO')

exit(0)

if bfs(start, end):

ans = traverse(end)

print(' '.join(ans[::-1]))

else:

print('NO')

**28050: 骑士周游**

def knight\_tour(n, sr, sc):

moves = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),

(1, -2), (1, 2), (2, -1), (2, 1)]

visited = [[False] \* n for \_ in range(n)]

def is\_valid\_move(row, col):

return 0 <= row < n and 0 <= col < n and not visited[row][col]

def count\_neighbors(row, col):

count = 0

for dr, dc in moves:

next\_row, next\_col = row + dr, col + dc

if is\_valid\_move(next\_row, next\_col):

count += 1

return count

def sort\_moves(row, col):

neighbor\_counts = []

for dr, dc in moves:

next\_row, next\_col = row + dr, col + dc

if is\_valid\_move(next\_row, next\_col):

count = count\_neighbors(next\_row, next\_col)

neighbor\_counts.append((count, (next\_row, next\_col)))

neighbor\_counts.sort()

sorted\_moves = [move[1] for move in neighbor\_counts]

return sorted\_moves

visited[sr][sc] = True

tour = [(sr, sc)]

while len(tour) < n \* n:

current\_row, current\_col = tour[-1]

sorted\_next\_moves = sort\_moves(current\_row, current\_col)

if not sorted\_next\_moves:

return "fail"

next\_row, next\_col = sorted\_next\_moves[0]

visited[next\_row][next\_col] = True

tour.append((next\_row, next\_col))

return "success"

n = int(input())

sr, sc = map(int, input().split())

print(knight\_tour(n, sr, sc))

A10

**20743: 整人的提词本**

def reverse\_se(s):

stack = []

for char in s:

if char == ')':

temp = []

while stack and stack[-1] != '(':

temp.append(stack.pop())

if stack:

stack.pop()

stack.extend(temp)

else:

stack.append(char)

return ''.join(stack)

s = input().strip()

print(reverse\_se(s))

**02255: 重建二叉树**

def build\_tree(preorder, inorder):

if not preorder:

return ''

root = preorder[0]

root\_index = inorder.index(root)

left\_preorder = preorder[1:1 + root\_index]

right\_preorder = preorder[1 + root\_index:]

left\_inorder = inorder[:root\_index]

right\_inorder = inorder[root\_index + 1:]

left\_tree = build\_tree(left\_preorder, left\_inorder)

right\_tree = build\_tree(right\_preorder, right\_inorder)

return left\_tree + right\_tree + root

while True:

try:

preorder, inorder = input().split()

postorder = build\_tree(preorder, inorder)

print(postorder)

except EOFError:

Break

**01426: Find The Multiple**

from collections import deque

def find\_multiple(n):

# 使用队列实现BFS

q = deque()

# 初始化队列，存储的是(模n值, 对应的数字字符串)

q.append((1 % n, "1"))

visited = set([1 % n]) # 用于记录访问过的模n值，避免重复搜索

while q:

mod, num\_str = q.popleft()

# 检查当前模n值是否为0，是则找到答案

if mod == 0:

return num\_str

# 尝试在当前数字后加0或加1，生成新的数字，并计算模n值

for digit in ["0", "1"]:

new\_num\_str = num\_str + digit

new\_mod = (mod \* 10 + int(digit)) % n

# 如果新模n值未访问过，则加入队列继续搜索

if new\_mod not in visited:

q.append((new\_mod, new\_num\_str))

visited.add(new\_mod)

def main():

while True:

n = int(input())

if n == 0:

break

print(find\_multiple(n))

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

main()

**04115: 鸣人和佐助**

from collections import deque

M, N, T = map(int, input().split())

graph = [list(input()) for i in range(M)]

direc = [(0,1), (1,0), (-1,0), (0,-1)]

start, end = None, None

for i in range(M):

for j in range(N):

if graph[i][j] == '@':

start = (i, j)

def bfs():

q = deque([start + (T, 0)])

visited = [[-1]\*N for i in range(M)]

visited[start[0]][start[1]] = T

while q:

x, y, t, time = q.popleft()

time += 1

for dx, dy in direc:

if 0<=x+dx<M and 0<=y+dy<N:

if (elem := graph[x+dx][y+dy]) == '\*' and t > visited[x+dx][y+dy]:

visited[x+dx][y+dy] = t

q.append((x+dx, y+dy, t, time))

elif elem == '#' and t > 0 and t-1 > visited[x+dx][y+dy]:

visited[x+dx][y+dy] = t-1

q.append((x+dx, y+dy, t-1, time))

elif elem == '+':

return time

return -1

print(bfs())

**20106: 走山路**

import heapq

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

martix = [list(input().split())for i in range(m)]

dir = [(-1, 0), (1, 0), (0, 1), (0, -1)]

for \_ in range(p):

sx, sy, ex, ey = map(int, input().split())

if martix[sx][sy] == "#" or martix[ex][ey] == "#":

print("NO")

continue

vis, heap, ans = set(), [], []

heapq.heappush(heap, (0, sx, sy))

vis.add((sx, sy, -1))

while heap:

tire, x, y = heapq.heappop(heap)

if x == ex and y == ey:

ans.append(tire)

for i in range(4):

dx, dy = dir[i]

x1, y1 = dx+x, dy+y

if 0 <= x1 < m and 0 <= y1 < n and martix[x1][y1] != "#" and (x1, y1, i) not in vis:

t1 = tire+abs(int(martix[x][y])-int(martix[x1][y1]))

heapq.heappush(heap, (t1, x1, y1))

vis.add((x1, y1, i))

print(min(ans) if ans else "NO")

**05442: 兔子与星空**

import heapq

def prim(graph, start):

mst = []

used = set([start])

edges = [

(cost, start, to)

for to, cost in graph[start].items()

]

heapq.heapify(edges)

while edges:

cost, frm, to = heapq.heappop(edges)

if to not in used:

used.add(to)

mst.append((frm, to, cost))

for to\_next, cost2 in graph[to].items():

if to\_next not in used:

heapq.heappush(edges, (cost2, to, to\_next))

return mst

def solve():

n = int(input())

graph = {chr(i+65): {} for i in range(n)}

for i in range(n-1):

data = input().split()

star = data[0]

m = int(data[1])

for j in range(m):

to\_star = data[2+j\*2]

cost = int(data[3+j\*2])

graph[star][to\_star] = cost

graph[to\_star][star] = cost

mst = prim(graph, 'A')

print(sum(x[2] for x in mst))

solve()

A11

**28170: 算鹰**

import sys

sys.setrecursionlimit(20000)

def dfs(x, y):

# 标记，避免再次访问

field[x][y] = '-'

for k in range(4):

nx, ny = x + dx[k], y + dy[k]

# 范围内且未访问的lake

if 0 <= nx < n and 0 <= ny < m \

and field[nx][ny] == '.':

# 继续搜索

dfs(nx, ny)

(n, m) = (10, 10)

field = [list(input()) for \_ in range(n)]

cnt = 0

dx = [0, -1, 1, 0]

dy = [-1, 0, 0, 1]

for i in range(n):

for j in range(m):

if field[i][j] == '.':

dfs(i, j)

cnt += 1

print(cnt)

**02754: 八皇后**

def solve\_n\_queens(n):

stack = []

solutions = []

stack.append((0, [-1] \* n))

while stack:

row, queens = stack.pop()

if row == n:

solutions.append(queens.copy())

else:

for col in range(n):

if is\_valid(row, col, queens):

new\_queens = queens.copy()

new\_queens[row] = col

stack.append((row + 1, new\_queens))

return solutions

def is\_valid(row, col, queens):

for r in range(row):

if queens[r] == col or abs(row - r) == abs(col - queens[r]):

return False

return True

def get\_queen\_string(b):

solutions = solve\_n\_queens(8)

if b > len(solutions):

return None

b = len(solutions) + 1 - b

queen\_string = ''.join(str(col + 1) for col in solutions[b - 1])

return queen\_string

test\_cases = int(input())

for \_ in range(test\_cases):

b = int(input())

queen\_string = get\_queen\_string(b)

print(queen\_string)

**03151: Pots**

#

# pots 最短路径问题 通常使用bfs bfs就是自己去假设下一次会发生的所有可能性 全部储存在visited中

def bfs(A, B, C):

start = (0, 0)

visited = set()

visited.add(start)

queue = [(start, [])]

while queue:

(a, b), actions = queue.pop(0)

if a == C or b == C:

return actions

next\_states = [(A, b), (a, B), (0, b), (a, 0), (min(a + b, A), max(0, a + b - A)), (max(0, a + b - B), min(a + b, B))]

for i in next\_states:

if i not in visited:

visited.add(i)

new\_actions = actions + [get\_action(a, b, i)]

queue.append((i, new\_actions))

return ["impossible"]

def get\_action(a, b, next\_state):

if next\_state == (A, b):

return "FILL(1)"

elif next\_state == (a, B):

return "FILL(2)"

elif next\_state == (0, b):

return "DROP(1)"

elif next\_state == (a, 0):

return "DROP(2)"

elif next\_state == (min(a + b, A), max(0, a + b - A)):

return "POUR(2,1)"

else:

return "POUR(1,2)"

A, B, C = map(int, input().split())

solution = bfs(A, B, C)

if solution == ["impossible"]:

print(solution[0])

else:

print(len(solution))

for i in solution:

print(i)

**05907: 二叉树的操作**

class TreeNode:

def \_\_init\_\_(self, val=0):

self.val = val

self.left = None

self.right = None

self.parent = None

def build\_tree(nodes\_info):

nodes = [TreeNode(i) for i in range(n)]

for val, left, right in nodes\_info:

if left != -1:

nodes[val].left = nodes[left]

if right != -1:

nodes[val].right = nodes[right]

return nodes

def swap\_nodes(nodes, x, y):

for node in nodes:

if node.left and node.left.val in [x, y]:

node.left = nodes[y] if node.left.val == x else nodes[x]

if node.right and node.right.val in [x, y]:

node.right = nodes[y] if node.right.val == x else nodes[x]

def find\_leftmost(node):

while node and node.left:

node = node.left

return node.val if node else -1

for \_ in range(int(input())):

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

nodes\_info = [tuple(map(int, input().split())) for \_ in range(n)]

ops = [tuple(map(int, input().split())) for \_ in range(m)]

nodes = build\_tree(nodes\_info)

for op in ops:

if op[0] == 1:

swap\_nodes(nodes, op[1], op[2])

elif op[0] == 2:

print(find\_leftmost(nodes[op[1]]))

**18250: 冰阔落 I**

def find(x):

if parent[x] != x:

parent[x] = find(parent[x])

return parent[x]

def union(x, y):

root\_x = find(x)

root\_y = find(y)

if root\_x != root\_y:

parent[root\_y] = root\_x

while True:

try:

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

parent = list(range(n + 1))

for \_ in range(m):

a, b = map(int, input().split())

if find(a) == find(b):

print('Yes')

else:

print('No')

union(a, b)

unique\_parents = set(find(x) for x in range(1, n + 1)) # 获取不同集合的根节点

ans = sorted(unique\_parents) # 输出有冰阔落的杯子编号

print(len(ans))

print(\*ans)

except EOFError:

Break

**05443: 兔子与樱花**

import heapq

def dijkstra(adjacency, start):

distances = {vertex: float('infinity') for vertex in adjacency}

previous = {vertex: None for vertex in adjacency}

distances[start] = 0

pq = [(0, start)]

while pq:

current\_distance, current\_vertex = heapq.heappop(pq)

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in adjacency[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

previous[neighbor] = current\_vertex

heapq.heappush(pq, (distance, neighbor))

return distances, previous

def shortest\_path\_to(adjacency, start, end):

distances, previous = dijkstra(adjacency, start)

path = []

current = end

while previous[current] is not None:

path.insert(0, current)

current = previous[current]

path.insert(0, start)

return path, distances[end]

# Read the input data

P = int(input())

places = {input().strip() for \_ in range(P)}

Q = int(input())

graph = {place: {} for place in places}

for \_ in range(Q):

src, dest, dist = input().split()

dist = int(dist)

graph[src][dest] = dist

graph[dest][src] = dist # Assuming the graph is bidirectional

R = int(input())

requests = [input().split() for \_ in range(R)]

# Process each request

for start, end in requests:

if start == end:

print(start)

continue

path, total\_dist = shortest\_path\_to(graph, start, end)

output = ""

for i in range(len(path) - 1):

output += f"{path[i]}->({graph[path[i]][path[i+1]]})->"

output += f"{end}"

print(output)

A12

**01258: Agri-Net**

# MST 最小生成树 prim or kruskal

# 先根据矩阵建立无向邻接表

class DisjointSetUnion:

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

self.parent = list(range(n))

self.rank = [0] \* n

# 找到共同元素 看看两个是不是一伙的 如果联通为环

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

xr = self.find(x)

yr = self.find(y)

if xr == yr:

return False

elif self.rank[xr] < self.rank[yr]:

self.parent[xr] = yr

elif self.rank[xr] > self.rank[yr]:

self.parent[yr] = xr

else:

self.parent[yr] = xr

self.rank[xr] += 1 # 为了更好区分吧

return True

# 对于已经建好的邻接表进行k算法,对于u,v进行连接，找到

def kruskal(n, edges):

dsu = DisjointSetUnion(n)

mst\_weight = 0

for weight, u, v in sorted(edges):

if dsu.union(u, v):

mst\_weight += weight

return mst\_weight

# 开始将矩阵转化为邻接表

def main():

while True:

try:

n = int(input().strip())

edges = []

for i in range(n):

# Since the input lines may continue onto others, we read them all at once

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

for j in range(i + 1, n):

if row[j] != 0: # No need to add edges with 0 weight

edges.append((row[j], i, j))

print(kruskal(n, edges))

except EOFError: # Exit the loop when all test cases are processed

break

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

main()

**27635: 判断无向图是否连通有无回路(同23163)**

def is\_connected(graph, n):

visited = [False] \* n # 记录节点是否被访问过

stack = [0] # 使用栈来进行DFS

visited[0] = True

while stack:

node = stack.pop()

for neighbor in graph[node]:

if not visited[neighbor]:

stack.append(neighbor)

visited[neighbor] = True

return all(visited)

def dfs(node, visited, parent):

visited[node] = True

for neighbor in graph[node]:

if not visited[neighbor]:

if dfs(neighbor, visited, node):

return True

elif parent != neighbor:

return True

return False

def has\_cycle(graph, n):

visited = [False] \* n

for node in range(n):

if not visited[node]:

if dfs(node, visited, -1):

return True

return False

# 读取输入

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

graph = [[] for \_ in range(n)]

for \_ in range(m):

u, v = map(int, input().split())

graph[u].append(v)

graph[v].append(u)

# 判断连通性和回路

connected = is\_connected(graph, n)

has\_loop = has\_cycle(graph, n)

print("connected:yes" if connected else "connected:no")

print("loop:yes" if has\_loop else "loop:no")

**27947: 动态中位数**

import heapq

def dynamic\_median(nums):

# 维护小根和大根堆（对顶），保持中位数在大根堆的顶部

min\_heap = [] # 存储较大的一半元素，使用最小堆

max\_heap = [] # 存储较小的一半元素，使用最大堆

median = []

for i, num in enumerate(nums):

# 根据当前元素的大小将其插入到对应的堆中

if not max\_heap or num <= -max\_heap[0]:

heapq.heappush(max\_heap, -num)

else:

heapq.heappush(min\_heap, num)

# 调整两个堆的大小差，使其不超过 1

if len(max\_heap) - len(min\_heap) > 1:

heapq.heappush(min\_heap, -heapq.heappop(max\_heap))

elif len(min\_heap) > len(max\_heap):

heapq.heappush(max\_heap, -heapq.heappop(min\_heap))

if i % 2 == 0:

median.append(-max\_heap[0])

return median

T = int(input())

for \_ in range(T):

# M = int(input())

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

median = dynamic\_median(nums)

print(len(median))

print(\*median)

**28190: 奶牛排队**

N = int(input())

heights = [int(input()) for \_ in range(N)]

left\_bound = [-1] \* N

right\_bound = [N] \* N

stack = [] # 单调栈，存储索引

# 求左侧第一个≥h[i]的奶牛位置

for i in range(N):

while stack and heights[stack[-1]] < heights[i]:

stack.pop()

if stack:

left\_bound[i] = stack[-1]

stack.append(i)

stack = [] # 清空栈以供寻找右边界使用

# 求右侧第一个≤h[i]的奶牛位

for i in range(N-1, -1, -1):

while stack and heights[stack[-1]] > heights[i]:

stack.pop()

if stack:

right\_bound[i] = stack[-1]

stack.append(i)

ans = 0

for i in range(N): # 枚举右端点 B寻找 A，更新 ans

for j in range(left\_bound[i] + 1, i):

if right\_bound[j] > i:

ans = max(ans, i - j + 1)

break

print(ans)

A13

**22485: 升空的焰火，从侧面看**

from collections import deque

def right\_view(n, tree):

queue = deque([(1, tree[1])]) # start with root node

right\_view = []

while queue:

level\_size = len(queue)

for i in range(level\_size):

node, children = queue.popleft()

if children[0] != -1:

queue.append((children[0], tree[children[0]]))

if children[1] != -1:

queue.append((children[1], tree[children[1]]))

right\_view.append(node)

return right\_view

n = int(input())

tree = {1: [-1, -1] for \_ in range(n + 1)}

for i in range(1, n + 1):

left, right = map(int, input().split())

tree[i] = [left, right]

result = right\_view(n, tree)

print(' '.join(map(str, result)))

**28203:【模板】单调栈**

# 单调栈

n = int(input())

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

stack = []

for i in range(n):

while stack and a[stack[-1]] < a[i]:

a[stack.pop()] = i + 1

stack.append(i)

while stack:

a[stack[-1]] = 0

stack.pop()

print(\*a) 这样写不会超时

**09202: 舰队、海域出击！**

from collections import defaultdict

def dfs(p):

vis[p] = True

for q in graph[p]:

in\_degree[q] -= 1

if in\_degree[q] == 0:

dfs(q)

for \_ in range(int(input())):

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

graph = defaultdict(list)

in\_degree = [0] \* (n + 1)

vis = [False] \* (n + 1)

for \_ in range(m):

x, y = map(int, input().split())

graph[x].append(y)

in\_degree[y] += 1

for k in range(1, n + 1):

if in\_degree[k] == 0 and not vis[k]:

dfs(k)

flag = any(not vis[i] for i in range(1, n + 1))

print('Yes' if flag else 'No')

**04135: 月度开销**

#

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

expenditure = []

for \_ in range(n):

expenditure.append(int(input()))

def check(x):

num, s = 1, 0

for i in range(n):

if s + expenditure[i] > x:

s = expenditure[i]

num += 1

else:

s += expenditure[i]

return [False, True][num > m]

# https://github.com/python/cpython/blob/main/Lib/bisect.py

lo = max(expenditure)

# hi = sum(expenditure)

hi = sum(expenditure) + 1

ans = 1

while lo < hi:

mid = (lo + hi) // 2

if check(mid): # 返回True，是因为num>m，是确定不合适

lo = mid + 1 # 所以lo可以置为 mid + 1。

else:

ans = mid # 如果num==m, mid可能是答案

hi = mid

#print(lo)

print(ans)

**07735: 道路**

import heapq

def dijkstra(g):

while pq:

dist,node,fee = heapq.heappop(pq)

if node == n-1 :

return dist

for nei,w,f in g[node]:

n\_dist = dist + w

n\_fee = fee + f

if n\_fee <= k:

dists[nei] = n\_dist

heapq.heappush(pq,(n\_dist,nei,n\_fee))

return -1

k,n,r = int(input()),int(input()),int(input())

g = [[] for \_ in range(n)]

for i in range(r):

s,d,l,t = map(int,input().split())

g[s-1].append((d-1,l,t)) #node,dist,fee

pq = [(0,0,0)] #dist,node,fee

dists = [float('inf')] \* n

dists[0] = 0

spend = 0

result = dijkstra(g)

print(result)

**01182: 食物链**

class DisjointSet:

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

#设[1,n] 区间表示同类，[n+1,2\*n]表示x吃的动物，[2\*n+1,3\*n]表示吃x的动物。

self.parent = [i for i in range(3 \* n + 1)] # 每个动物有三种可能的类型，用 3 \* n 来表示每种类型的并查集

self.rank = [0] \* (3 \* n + 1)

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu == pv:

return False

if self.rank[pu] > self.rank[pv]:

self.parent[pv] = pu

elif self.rank[pu] < self.rank[pv]:

self.parent[pu] = pv

else:

self.parent[pv] = pu

self.rank[pu] += 1

return True

def is\_valid(n, k, statements):

dsu = DisjointSet(n)

def find\_disjoint\_set(x):

if x > n:

return False

return True

false\_count = 0

for d, x, y in statements:

if not find\_disjoint\_set(x) or not find\_disjoint\_set(y):

false\_count += 1

continue

if d == 1: # X and Y are of the same type

if dsu.find(x) == dsu.find(y + n) or dsu.find(x) == dsu.find(y + 2 \* n):

false\_count += 1

else:

dsu.union(x, y)

dsu.union(x + n, y + n)

dsu.union(x + 2 \* n, y + 2 \* n)

else: # X eats Y

if dsu.find(x) == dsu.find(y) or dsu.find(x + 2\*n) == dsu.find(y):

false\_count += 1

else: #[1,n] 区间表示同类，[n+1,2\*n]表示x吃的动物，[2\*n+1,3\*n]表示吃x的动物

dsu.union(x + n, y)

dsu.union(x, y + 2 \* n)

dsu.union(x + 2 \* n, y + n)

return false\_count

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

N, K = map(int, input().split())

statements = []

for \_ in range(K):

D, X, Y = map(int, input().split())

statements.append((D, X, Y))

result = is\_valid(N, K, statements)

print(result)

每日选做

**01035：拼写检查**def similar(word):  
 outputs = []  
 for dict\_word in dicts:  
 if len(dict\_word) == len(word):  
 differents = 0  
 for i in range(len(word)):  
 if dict\_word[i] != word[i]:  
 differents += 1  
  
 if differents > 1:  
 break  
  
 if differents <= 1:  
 outputs.append(dict\_word)  
 elif len(dict\_word) - len(word) == 1:  
 moves = 0  
 i = j = 0  
 while i < len(word) and j < len(dict\_word):  
 if word[i] != dict\_word[j]:  
 j += 1  
 moves += 1  
 else:  
 i += 1  
 j += 1  
 if moves <= 1:  
 outputs.append(dict\_word)  
 elif len(word) - len(dict\_word) == 1:  
 moves = 0  
 i = j = 0  
 while i < len(dict\_word) and j < len(word):  
 if word[j] != dict\_word[i]:  
 j += 1  
 moves += 1  
 else:  
 i += 1  
 j += 1  
 if moves <= 1:  
 outputs.append(dict\_word)  
  
 return outputs  
  
dicts = []  
  
while True:  
 word = input()  
 if word == '#':  
 break  
  
 dicts.append(word)  
  
while True:  
 word = input()  
 if word == '#':  
 break  
 elif word in dicts:  
 print(f'{word} is correct')  
 else:  
 similars = similar(word)  
 similars.insert(0, word + ':')  
 print(' '.join(similars))

**01084：正方形破坏者**

import copy

import sys

sys.setrecursionlimit(1 << 30)

found = False

def check1(x, tmp):

for y in graph[x]:

if tmp[y]:

return False

return True

def check2(x):

for y in graph[x]:

if judge[y]:

return False

return True

def estimate():

cnt = 0

tmp = copy.deepcopy(judge)

for x in range(1, total+1):

if check1(x, tmp):

cnt += 1

for u in graph[x]:

tmp[u] = True

return cnt

def dfs(t):

global found

if t + estimate() > limit:

return

for x in range(1, total+1):

if check2(x):

for y in graph[x]:

judge[y] = True

dfs(t+1)

judge[y] = False

if found:

return

return

found = True

for \_ in range(int(input())):

n = int(input())

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

d, m, nums, total = 2\*n+1, lst[0], lst[1:], 0

graph = {}

for i in range(n):

for j in range(n):

for k in range(1, n+1):

if i+k <= n and j+k <= n:

total += 1

graph[total] = []

for p in range(1, k+1):

graph[total] += [d\*i+j+p, d\*(i+p)+j-n, d\*(i+p)+j-n+k, d\*(i+k)+j+p]

judge = [False for \_ in range(2\*n\*(n+1)+1)]

for num in nums:

judge[num] = True

limit = estimate()

found = False

while True:

dfs(0)

if found:

print(limit)

break

limit += 1

**01094: Sorting It All Out**

from collections import deque

def topo\_sort(graph):

in\_degree = {u:0 for u in graph}

for u in graph:

for v in graph[u]:

in\_degree[v] += 1

q = deque([u for u in in\_degree if in\_degree[u] == 0])

topo\_order = [];flag = True

while q:

if len(q) > 1:

flag = False#topo\_sort不唯一确定

u = q.popleft()

topo\_order.append(u)

for v in graph[u]:

in\_degree[v] -= 1

if in\_degree[v] == 0:

q.append(v)

if len(topo\_order) != len(graph): return 0

return topo\_order if flag else None

while True:

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

if n == 0: break

graph = {chr(x+65):[] for x in range(n)}

edges = [tuple(input().split('<')) for \_ in range(m)]

for i in range(m):

a,b = edges[i]

graph[a].append(b)

t = topo\_sort(graph)

if t:

s = ''.join(t)

print("Sorted sequence determined after {} relations: {}.".format(i+1,s))

break

elif t == 0:

print("Inconsistency found after {} relations.".format(i+1))

break

else:

print("Sorted sequence cannot be determined.")

**01145:Tree Summing**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def has\_path\_sum(root, target\_sum):

if root is None:

return False

if root.left is None and root.right is None: # The current node is a leaf node

return root.val == target\_sum

left\_exists = has\_path\_sum(root.left, target\_sum - root.val)

right\_exists = has\_path\_sum(root.right, target\_sum - root.val)

return left\_exists or right\_exists

# Parse the input string and build a binary tree

def parse\_tree(s):

stack = []

i = 0

while i < len(s):

if s[i].isdigit() or s[i] == '-':

j = i

while j < len(s) and (s[j].isdigit() or s[j] == '-'):

j += 1

num = int(s[i:j])

node = TreeNode(num)

if stack:

parent = stack[-1]

if parent.left is None:

parent.left = node

else:

parent.right = node

stack.append(node)

i = j

elif s[i] == '[':

i += 1

elif s[i] == ']' and s[i - 1] != '[' and len(stack) > 1:

stack.pop()

i += 1

else:

i += 1

return stack[0] if len(stack) > 0 else None

while True:

try:

s = input()

except:

break

s = s.split()

target\_sum = int(s[0])

tree = ("").join(s[1:])

tree = tree.replace('(', ',[').replace(')', ']')

while True:

try:

tree = eval(tree[1:])

break

except SyntaxError:

s = input().split()

s = ("").join(s)

s = s.replace('(', ',[').replace(')', ']')

tree += s

tree = str(tree)

tree = tree.replace(',[', '[')

if tree == '[]':

print("no")

continue

root = parse\_tree(tree)

if has\_path\_sum(root, target\_sum):

print("yes")

else:

print("no")

**01191:棋盘分割**

from collections import defaultdict

def f(n, x1, y1, x2, y2):

if dp[(n, x1, y1, x2, y2)] > 0:

return dp[(n, x1, y1, x2, y2)]

if n == 1:

su = 0

for i in range(x1, x2+1):

for j in range(y1, y2+1):

su += l[i][j]

dp[(n, x1, y1, x2, y2)] = su\*su

return su\*su

#mi = 10000000

mi = float('inf')

for i in range(x1, x2):

mi = min(mi, f(n-1, x1, y1, i, y2)+f(1, i+1, y1, x2, y2))

mi = min(mi, f(1, x1, y1, i, y2)+f(n-1, i+1, y1, x2, y2))

for i in range(y1, y2):

mi = min(mi, f(n-1, x1, y1, x2, i)+f(1, x1, i+1, x2, y2))

mi = min(mi, f(1, x1, y1, x2, i)+f(n-1, x1, i+1, x2, y2))

dp[(n, x1, y1, x2, y2)] = mi

return mi

n = int(input())

l = []

for i in range(8):

l.append([int(x) for x in input().split()])

s = 0

for i in l:

for j in i:

s += j

dp = defaultdict(int)

print("%.3f"%(f(n, 0,0,7,7)/n-s\*s/n/n)\*\*0.5)

**01321：棋盘问题**

n, k, ans = 0, 0, 0

chess = [['' for \_ in range(10)] for \_ in range(10)]

take = [False] \* 10

def dfs(h, t):

global ans

if t == k:

ans += 1

return

if h == n:

return

for i in range(h, n):

for j in range(n):

if chess[i][j] == '#' and not take[j]:

take[j] = True

dfs(i + 1, t + 1)

take[j] = False

while True:

n, k = map(int, input().split())

if n == -1 and k == -1:

break

for i in range(n):

chess[i] = list(input())

take = [False] \* 10

ans = 0

dfs(0, 0)

print(ans)

**01376：Robot**

from collections import deque

# Directions: north(0), east(1), south(2), west(3)

dx = [-1, 0, 1, 0]

dy = [0, 1, 0, -1]

def bfs(sx, sy, ex, ey, sdir):

queue = deque([(sx, sy, 0, sdir)])

visited = [[[0]\*4 for \_ in range(m+1)] for \_ in range(n+1)]

visited[sx][sy][sdir] = 1

while queue:

x, y, time, dir = queue.popleft()

for i in range(1, 4): # 1, 2, 3 steps

nx, ny = x + dx[dir]\*i, y + dy[dir]\*i

if nx < 1 or nx >= n or ny < 1 or ny >= m or grid[nx][ny] or grid[nx+1][ny] or grid[nx][ny+1] or grid[nx+1][ny+1]:

break

if not visited[nx][ny][dir]:

visited[nx][ny][dir] = 1

if nx == ex and ny == ey:

return time + 1

queue.append((nx, ny, time + 1, dir))

for i in range(4):

if abs(dir - i) == 2: # Don't go back

continue

if not visited[x][y][i]: # Turn in place, no need to check boundaries

visited[x][y][i] = 1

queue.append((x, y, time + 1, i))

return -1

while True:

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

if n == 0 and m == 0:

break

grid = [[0]\*(m+2) for \_ in range(n+2)]

for i in range(1, n+1):

grid[i] = [0] + list(map(int, input().split())) + [0]

sx, sy, ex, ey, sdir = input().split()

sx, sy, ex, ey = map(int, [sx, sy, ex, ey])

sdir = {'n': 0, 'e': 1, 's': 2, 'w': 3}[sdir[0]]

if sx == ex and sy == ey:

print(0)

continue

print(bfs(sx, sy, ex, ey, sdir))

**01577: Falling Leaves**

class TreeNode:

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

self.data = data

self.left = None

self.right = None

def build\_bst(leaves):

if not leaves:

return None

root = TreeNode(leaves[0])

for leaf in leaves[1:]:

insert\_node(root, leaf)

return root

def insert\_node(root, leaf):

if leaf < root.data:

if root.left is None:

root.left = TreeNode(leaf)

else:

insert\_node(root.left, leaf)

else:

if root.right is None:

root.right = TreeNode(leaf)

else:

insert\_node(root.right, leaf)

def preorder\_traversal(root):

if root is None:

return []

traversal = [root.data]

traversal.extend(preorder\_traversal(root.left))

traversal.extend(preorder\_traversal(root.right))

return traversal

# 读取输入数据

flag = 0

while True:

leaves = []

while True:

line = input().strip()

if line == '\*':

break

elif line == '$':

flag = 1

break

else:

leaves.extend(line)

# 构建二叉搜索树

root = build\_bst(leaves[::-1])

# 输出前序遍历结果

traversal\_result = preorder\_traversal(root)

print(''.join(traversal\_result))

if flag:

break

**01611：The suspects**

class UnionFind:

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

self.parent = list(range(n)) # Each student initially in their own set

self.rank = [0] \* n # Rank of each node for path compression

def find(self, x):

# Find the representative (root) of the set that x is in

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x]) # Path compression

return self.parent[x]

def union(self, x, y):

# Union the sets that x and y are in

root\_x = self.find(x)

root\_y = self.find(y)

if root\_x != root\_y:

if self.rank[root\_x] < self.rank[root\_y]:

self.parent[root\_x] = root\_y

elif self.rank[root\_y] < self.rank[root\_x]:

self.parent[root\_y] = root\_x

else:

self.parent[root\_y] = root\_x

self.rank[root\_x] += 1

def find\_suspects(n, groups):

uf = UnionFind(n)

for group in groups:

for student in group[1:]:

uf.union(group[0], student) # Union the first student in the group with all others

suspect\_set = set()

for i in range(n):

if uf.find(0) == uf.find(i): # If student is in the same set as the initial suspect

suspect\_set.add(i)

return len(suspect\_set)

def main():

while True:

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

if n == 0 and m == 0:

break

groups = [list(map(int, input().split()))[1:] for \_ in range(m)]

print(find\_suspects(n, groups))

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

main()

**01703:发现它，抓住它**

class UnionFind:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX != rootY:

if self.rank[rootX] > self.rank[rootY]:

self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX

self.rank[rootX] += 1

def solve():

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

uf = UnionFind(2 \* n) # 初始化并查集，每个案件对应两个节点

for \_ in range(m):

operation, a, b = input().split()

a, b = int(a) - 1, int(b) - 1

if operation == "D":

uf.union(a, b + n) # a与b的对立案件合并

uf.union(a + n, b) # a的对立案件与b合并

else: # "A"

if uf.find(a) == uf.find(b) or uf.find(a + n) == uf.find(b + n):

print("In the same gang.")

elif uf.find(a) == uf.find(b + n) or uf.find(a + n) == uf.find(b):

print("In different gangs.")

else:

print("Not sure yet.")

T = int(input())

for \_ in range(T):

solve()

**01724：ROADS**

import heapq

from collections import defaultdict

MAX\_COINS = int(input()) # 最大金币数

CITY\_COUNT = int(input()) # 城市数目

ROAD\_COUNT = int(input())

# 存储道路信息的字典，使用 defaultdict 初始化

roads = defaultdict(list)

for \_ in range(ROAD\_COUNT):

start, end, length, money = map(int, input().split())

start, end = start - 1, end - 1

roads[start].append((end, length, money))

def bfs(start, end, max\_coins):

queue = [(0, max\_coins, start)] # (距离, 剩余金币, 当前城市)

visited = set()

while queue:

distance, coins, city = heapq.heappop(queue)

if city == end:

return distance

visited.add((city, coins))

for next\_city, road\_length, road\_money in roads[city]:

if coins >= road\_money:

new\_distance = distance + road\_length

if (next\_city, coins - road\_money) not in visited:

heapq.heappush(queue, (new\_distance, coins - road\_money, next\_city))

return -1

print(bfs(0, CITY\_COUNT - 1, MAX\_COINS))

**01760: Disk Tree**

# 23n2300011031

class Node:

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

self.children={}

class Trie:

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

self.root=Node()

def insert(self,w):

cur=self.root

for u in w.split('\\'):

if u not in cur.children:

cur.children[u]=Node()

cur=cur.children[u]

def dfs(self,a,layer):

for c in sorted(a.children):

print(' '\*layer+c)

self.dfs(a.children[c], layer+1)

s=Trie()

for \_ in range(int(input())):

x=input()

s.insert(x)

s.dfs(s.root, 0)

**01798：Truck History**

import heapq

def truck\_history():

while True:

n = int(input())

if n == 0:

break

trucks = [input() for \_ in range(n)]

trucks.sort()

graph = [[0]\*n for \_ in range(n)]

for i in range(n):

for j in range(i+1, n):

graph[i][j] = graph[j][i] = sum(a!=b for a, b in zip(trucks[i], trucks[j]))

visited = [False]\*n

min\_edge = [float('inf')]\*n

min\_edge[0] = 0

total\_distance = 0

min\_heap = [(0, 0)]

while min\_heap:

d, v = heapq.heappop(min\_heap)

if visited[v]:

continue

visited[v] = True

total\_distance += d

for u in range(n):

if not visited[u] and graph[v][u] < min\_edge[u]:

min\_edge[u] = graph[v][u]

heapq.heappush(min\_heap, (graph[v][u], u))

print(f"The highest possible quality is 1/{total\_distance}.")

truck\_history()

**01941: The Sierpinski Fractal**

def f(n):

if n == 1:

return [' /\\ ', '/\_\_\\']

t = f(n - 1)

x = 2 \*\* (n - 1)

res = [' ' \* x + u + ' ' \* x for u in t]

res.extend([u + u for u in t])

return res

al = [f(i) for i in range(1, 11)]

while True:

n = int(input())

if n == 0:

break

for u in al[n - 1]:

print(u)

print()

**01961:前缀中的周期**

P = int(input())

potions = []

for i in range(P):

potions.append((int(input())))

result = 0

sign = 1

for i in range(P-1):

if (potions[i + 1] - potions[i]) \* sign < 0:

result += sign \* potions[i]

sign = -sign

if sign == 1:

result += potions[P-1]

print(result)

**02039：反反复复**

cols = int(input())

encrypted = input()

# 计算行数

rows = len(encrypted) // cols

# 创建矩阵

matrix = [['' for \_ in range(cols)] for \_ in range(rows)]

# 填充矩阵

index = 0

for row in range(rows):

if row % 2 == 0: # 从左到右填充

for col in range(cols):

matrix[row][col] = encrypted[index]

index += 1

else: # 从右到左填充

for col in range(cols - 1, -1, -1):

matrix[row][col] = encrypted[index]

index += 1

# 从矩阵中提取原始信息

original = ''

for col in range(cols):

for row in range(rows):

original += matrix[row][col]

print(original)

**02192：Zipper**

def f(x, y, z):  
 if x == '!' and y == '!':  
 return True  
 if x[-1] == z[-1]:  
 if f(x[:-1], y, z[:-1]):  
 return True  
 return False  
  
  
n = int(input())  
for i in range(n):  
 x, y, z = input().split()  
 x = '!' + x  
 y = '!' + y  
 if f(x, y, z):  
 print(f'Data set{i + 1}: yes')  
 else:  
 print(f'Data set{i + 1}: no')

**02386:Lake Counting**

import sys

sys.setrecursionlimit(20000)

def dfs(x,y):

#标记，避免再次访问

field[x][y]='.'

for k in range(8):

nx,ny=x+dx[k],y+dy[k]

#范围内且未访问的lake

if 0<=nx<n and 0<=ny<m\

and field[nx][ny]=='W':

#继续搜索

dfs(nx,ny)

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

field=[list(input()) for \_ in range(n)]

cnt=0

dx=[-1,-1,-1,0,0,1,1,1]

dy=[-1,0,1,-1,1,-1,0,1]

for i in range(n):

for j in range(m):

if field[i][j]=='W':

dfs(i,j)

cnt+=1

print(cnt)

**02488：A knight’s Journey**

move = [(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)]

def dfs(x, y, step, p, q, visited, ans):

if step == p \* q:

return True

for i in range(8):

dx, dy = x + move[i][0], y + move[i][1]

if 1 <= dx <= q and 1 <= dy <= p and not visited[dx][dy]:

visited[dx][dy] = True

ans[step] = chr(dx + 64) + str(dy)

if dfs(dx, dy, step + 1, p, q, visited, ans):

return True

visited[dx][dy] = False

return False

n = int(input())

for m in range(1, n + 1):

p, q = map(int, input().split())

ans = ["" for \_ in range(p \* q)]

visited = [[False] \* (p + 1) for \_ in range(q + 1)]

visited[1][1] = True

ans[0] = "A1"

if dfs(1, 1, 1, p, q, visited, ans):

result = "".join(ans)

else:

result = "impossible"

print(f"Scenario #{m}:")

print(result)

print()

**02499:Binary Tree**

def binarytree(l, r, x, y):  
 if l == 1:  
 return [x, y + r - l]  
 elif r == 1:  
 return [x + l - r, y]  
 elif l > r:  
 n = l // r  
 ans = binarytree(l - r \* n, r, x + n, y)  
 else:  
 n = r // l  
 ans = binarytree(l, r - l \* n, x, y + n)  
 return ans  
  
  
for \_ in range(int(input())):  
 l, r = map(int, input().split())  
 ans = binarytree(l, r, 0, 0)  
 print(f'Scenario #{\_ + 1}:')  
 print(ans[0], ans[1])  
 print()

**02502:Subway**

import math

import heapq

def get\_distance(x1, y1, x2, y2):

return math.sqrt((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2)

sx, sy, ex, ey = map(int, input().split())

min\_time = {}

rails = set()

while True:

try:

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

if rail == [-1, -1]:

break

stations = [(rail[2 \* i], rail[2 \* i + 1]) for i in range(len(rail) // 2 - 1)]

for j, station in enumerate(stations):

min\_time[station] = float('inf')

if j != len(stations) - 1:

rails.add((station, stations[j + 1]))

rails.add((stations[j + 1], station))

except EOFError:

break

min\_time[(sx, sy)], min\_time[(ex, ey)] = 0, float('inf')

min\_heap = [(0, sx, sy)]

while min\_heap:

curr\_time, x, y = heapq.heappop(min\_heap)

if curr\_time > min\_time[(x, y)]:

continue

if (x, y) == (ex, ey):

break

for position in min\_time.keys():

if position == (x, y):

continue

nx, ny = position

dis = get\_distance(x, y, nx, ny)

rail\_factor = 4 if ((position, (x, y)) in rails or ((x, y), position) in rails) else 1

new\_time = curr\_time + dis / (10000 \* rail\_factor)

if new\_time < min\_time[position]:

min\_time[position] = new\_time

heapq.heappush(min\_heap, (new\_time, nx, ny))

print(round(min\_time[(ex, ey)] \* 60))

**02734:十进制到八进制**

decimal = int(input()) # 读取十进制数

# 创建一个空栈

stack = []

# 特殊情况：如果输入的数为0，直接输出0

if decimal == 0:

print(0)

else:

# 不断除以8，并将余数压入栈中

while decimal > 0:

remainder = decimal % 8

stack.append(remainder)

decimal = decimal // 8

# 依次出栈，构成八进制数的各个位

octal = ""

while stack:

octal += str(stack.pop())

print(octal)

**02746:约瑟夫问题**

def hot\_potato(name\_list, num):

queue = []

for name in name\_list:

queue.append(name)

while len(queue) > 1:

for i in range(num):

queue.append(queue.pop(0)) # O(N)

queue.pop(0) # O(N)

return queue.pop(0) # O(N)

while True:

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

if {n,m} == {0}:

break

monkey = [i for i in range(1, n+1)]

print(hot\_potato(monkey, m-1))

**02760：数字三角形**

n = int(input())

tri = [] # triangle

for i in range(n):

tri.append(list(map(int, input().split()))+[0 for j in range(n-i-1)])

for i in range(n-2,-1,-1):

for j in range(i+1):

tri[i][j] += max(tri[i+1][j], tri[i+1][j+1])

print(tri[0][0])

**02773：采药**

T,M = map(int, input().split())

herb = []

for i in range(M):

herb.append([int(x) for x in input().split()])

dp=[0]\*(T+1)

for i in range(M):

for j in range(T, herb[i][0] - 1, -1):

if j >= herb[i][0]:

dp[j] = max(dp[j], dp[j-herb[i][0]]+herb[i][1])

print(dp[-1])

**02774：木材加工**

n, k = map(int, input().split())

expenditure = []

for \_ in range(n):

expenditure.append(int(input()))

def check(x):

num = 0

for i in range(n):

num += expenditure[i] // x

return num >= k

lo = 1

hi = max(expenditure) + 1

if sum(expenditure) < k:

print(0)

exit()

ans = 1

while lo < hi:

mid = (lo + hi) // 2

if check(mid):

ans = mid

lo = mid + 1

else:

hi = mid

print(ans)

**02783：holiday hotel**

while True:

n=int(input())

if n==0:

break

hotels=[tuple(map(int,input().split())) for \_ in range(n)]

hotels.sort(key=lambda x:(x[0],x[1]))

candidates=1

max\_cost\_so\_far=hotels[0][1]

for i in range(n):

if hotels[i][1]<max\_cost\_so\_far:

candidates+=1

max\_cost\_so\_far=hotels[i][1]

print(candidates)

**02810：完美立方**

n = int(input())

cube = {i\*\*3: i for i in range(2, n+1)}

reversed\_cube = {v: k for k, v in cube.items()}

ans = []

for b in range(2, n):

for c in range(b, n):

for d in range(c, n):

if (a := reversed\_cube[b]+reversed\_cube[c]+reversed\_cube[d]) in cube:

ans.append((cube[a], b, c, d))

ans.sort()

for s in ans:

print(f"Cube = {s[0]}, Triple = ({s[1]},{s[2]},{s[3]})")

**02946：拦截导弹**

k=int(input())

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

dp=[0]\*k

for i in range(k-1,-1,-1):

maxn=1

for j in range(k-1,i,-1):

if l[i]>=l[j] and dp[j]+1>maxn:

maxn=dp[j]+1

dp[i]=maxn

print(max(dp))

**03704：括号匹配**

lines = []

while True:

try:

lines.append(input())

except EOFError:

break

ans = []

for s in lines:

stack = []

Mark = []

for i in range(len(s)):

if s[i] == '(':

stack.append(i)

Mark += ' '

elif s[i] == ')':

if len(stack) == 0:

Mark += '?'

else:

Mark += ' '

stack.pop()

else:

Mark += ' '

while(len(stack)):

Mark[stack[-1]] = '$'

stack.pop()

print(s)

print(''.join(map(str, Mark)))

**03720: 文本二叉树**

class Node:

def \_\_init\_\_(self, x, depth):

self.x = x

self.depth = depth

self.lchild = None

self.rchild = None

def preorder\_traversal(self):

nodes = [self.x]

if self.lchild and self.lchild.x != '\*':

nodes += self.lchild.preorder\_traversal()

if self.rchild and self.rchild.x != '\*':

nodes += self.rchild.preorder\_traversal()

return nodes

def inorder\_traversal(self):

nodes = []

if self.lchild and self.lchild.x != '\*':

nodes += self.lchild.inorder\_traversal()

nodes.append(self.x)

if self.rchild and self.rchild.x != '\*':

nodes += self.rchild.inorder\_traversal()

return nodes

def postorder\_traversal(self):

nodes = []

if self.lchild and self.lchild.x != '\*':

nodes += self.lchild.postorder\_traversal()

if self.rchild and self.rchild.x != '\*':

nodes += self.rchild.postorder\_traversal()

nodes.append(self.x)

return nodes

def build\_tree():

n = int(input())

for \_ in range(n):

tree = []

stack = []

while True:

s = input()

if s == '0':

break

depth = len(s) - 1

node = Node(s[-1], depth)

tree.append(node)

# Finding the parent for the current node

while stack and tree[stack[-1]].depth >= depth:

stack.pop()

if stack: # There is a parent

parent = tree[stack[-1]]

if not parent.lchild:

parent.lchild = node

else:

parent.rchild = node

stack.append(len(tree) - 1)

# Now tree[0] is the root of the tree

yield tree[0]

# Read each tree and perform traversals

for root in build\_tree():

print("".join(root.preorder\_traversal()))

print("".join(root.postorder\_traversal()))

print("".join(root.inorder\_traversal()))

print()

**04077：出栈序列统计**

from functools import lru\_cache

def count\_stack\_sequences(n):

@lru\_cache(None)

def backtrack(open\_count, close\_count):

if open\_count == n and close\_count == n:

return 1

total\_count = 0

if open\_count < n:

total\_count += backtrack(open\_count + 1, close\_count)

if close\_count < open\_count:

total\_count += backtrack(open\_count, close\_count + 1)

return total\_count

return backtrack(0, 0)

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

n = int(input())

result = count\_stack\_sequences(n)

print(result)

**04084：拓扑排序**

import heapq

def topological\_sort(vertices, edges):

# Initialize in-degree and connection matrix

in\_edges = [0] \* (vertices + 1)

connect = [[0] \* (vertices + 1) for \_ in range(vertices + 1)]

# Populate the in-degree and connection matrix

for u, v in edges:

in\_edges[v] += 1

connect[u][v] += 1

# Priority queue for vertices with in-degree of 0

queue = []

for i in range(1, vertices + 1):

if in\_edges[i] == 0:

heapq.heappush(queue, i)

# List to store the topological order

order = []

# Processing vertices

while queue:

u = heapq.heappop(queue)

order.append(u)

for v in range(1, vertices + 1):

if connect[u][v] > 0:

in\_edges[v] -= connect[u][v]

if in\_edges[v] == 0:

heapq.heappush(queue, v)

if len(order) == vertices:

return order

else:

return None

# Read input

vertices, num\_edges = map(int, input().split())

edges = []

for \_ in range(num\_edges):

u, v = map(int, input().split())

edges.append((u, v))

# Perform topological sort

order = topological\_sort(vertices, edges)

# Output result

if order:

for i, vertex in enumerate(order):

if i < len(order) - 1:

print(f"v{vertex}", end=" ")

else:

print(f"v{vertex}")

else:

print("No topological order exists due to a cycle in the graph.")

**04093：倒排索引查询**

import sys

input = sys.stdin.read

data = input().split()

index = 0

N = int(data[index])

index += 1

word\_documents = []

# 读取每个词的倒排索引

for \_ in range(N):

ci = int(data[index])

index += 1

documents = sorted(map(int, data[index:index + ci]))

index += ci

word\_documents.append(documents)

M = int(data[index])

index += 1

results = []

# 处理每个查询

for \_ in range(M):

query = list(map(int, data[index:index + N]))

index += N

# 集合存储各词的文档集合（使用交集获取所有词都出现的文档）

included\_docs = []

excluded\_docs = set()

# 解析查询条件

for i in range(N):

if query[i] == 1:

included\_docs.append(word\_documents[i])

elif query[i] == -1:

excluded\_docs.update(word\_documents[i])

# 仅在有包含词时计算交集

if included\_docs:

result\_set = set(included\_docs[0])

for docs in included\_docs[1:]:

result\_set.intersection\_update(docs)

result\_set.difference\_update(excluded\_docs)

final\_docs = sorted(result\_set)

results.append(" ".join(map(str, final\_docs)) if final\_docs else "NOT FOUND")

else:

results.append("NOT FOUND")

# 输出所有查询结果

for result in results:

print(result)

**04116 拯救行动**from collections import deque  
from heapq import heappush, heappop  
  
dx = [-1, 1, 0, 0]  
dy = [0, 0, -1, 1]  
  
  
def bfs(matrix, start):  
 n, m = len(matrix), len(matrix[0])  
 visited = [[False for \_ in range(m)] for \_ in range(n)]  
 # q = deque([(start[0], start[1], 0)])  
 q = []  
 heappush(q, (0, start[0], start[1]))  
 visited[start[0]][start[1]] = True  
 while len(q) != 0:  
 # x, y, time = q.popleft()  
 time, x, y = heappop(q)  
 for i in range(4):  
 nx, ny = x + dx[i], y + dy[i]  
 if 0 <= nx < n and 0 <= ny < m and not visited[nx][ny]:  
 if matrix[nx][ny] == "a":  
 # ans.append(time+1)  
 return time + 1  
 elif matrix[nx][ny] == "@":  
 # q.append((nx, ny, time + 1))  
 heappush(q, (time + 1, nx, ny))  
 visited[nx][ny] = True  
 elif matrix[nx][ny] == "x":  
 # q.append((nx, ny, time + 2))  
 heappush(q, (time + 2, nx, ny))  
 visited[nx][ny] = True  
  
 return "Impossible"  
  
  
S = int(input())  
for \_ in range(S):  
 N, M = map(int, input().split())  
 matrix = [list(input()) for \_ in range(N)]  
 start = None  
 ans = []  
 for i in range(N):  
 for j in range(M):  
 if matrix[i][j] == "r":  
 start = (i, j)  
 break  
 print(bfs(matrix, start))  
**04129：变换的迷宫**

arr2 = lambda m,n : [ [' ' for j in range(n)] for i in range(m) ]

arr3 = lambda m,n,l : [ [ [False for k in range(l)] for j in range(n)] for i in range(m) ]

N = 100

K = 10

class Node:

def \_\_init\_\_(self, r=0, c=0, t=0):

self.row = r

self.col = c

self.time = t

dr = [-1, 1, 0, 0]

dc = [0, 0, -1, 1]

for \_ in range(int(input())):

maze = arr2(N, N) # 注意不同数据组之间的初始化

vis = arr3(N, N, K)

q = []

r,c,k = map(int, input().split())

for i in range(r):

maze[i][:c] = list(input())

tr = tc = cnt = 0;

for i in range(r):

for j in range(c):

if maze[i][j] == 'S':

q.append(Node(i, j))

vis[i][j][0] = True

cnt += 1

if cnt == 2: break

elif maze[i][j] == 'E':

tr = i

tc = j

cnt += 1

if cnt == 2: break

while(len(q)):

t = q[0] # t : Node

if t.row == tr and t.col == tc: break

q.pop(0)

for i in range(4):

nrow = t.row + dr[i]

ncol = t.col + dc[i]

if nrow < 0 or nrow >= r or ncol < 0 or ncol >= c:

continue

if vis[nrow][ncol][(t.time + 1) % k]:

continue

# 时间是K 的倍数时，迷宫中的石头就会消失

if (t.time + 1) % k and maze[nrow][ncol] == '#':

continue;

vis[nrow][ncol][(t.time + 1) % k] = True

q.append(Node(nrow, ncol, t.time + 1))

if len(q) == 0:

print("Oop!")

else:

print(q[0].time)

**05333：Frence Repair**

import heapq

def minimum\_cost(planks):

heapq.heapify(planks) # 将木板列表转换为最小堆

total\_cost = 0

while len(planks) > 1:

# 取出最短的两块木板

shortest1 = heapq.heappop(planks)

shortest2 = heapq.heappop(planks)

# 计算切割的成本，并将切割后得到的木板长度加入堆

cost = shortest1 + shortest2

total\_cost += cost

heapq.heappush(planks, cost)

return total\_cost

# 读取输入

n = int(input())

planks = []

for \_ in range(n):

length = int(input())

planks.append(length)

# 调用函数计算最小成本

result = minimum\_cost(planks)

# 输出结果

print(result)

**01164:剪绳子**

import sys

try: fin = open('test.in','r').readline

except: fin = sys.stdin.readline

n = int(fin())

import heapq

a = list(map(int, fin().split()))

heapq.heapify(a)

ans = 0

for i in range(n-1):

x = heapq.heappop(a)

y = heapq.heappop(a)

z = x + y

heapq.heappush(a, z)

ans += z

print(ans)

**05430:表达式·表达式树·表达式求值**

from collections import deque as q

import operator as op

import os

class Node:

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

self.value = x

self.left = None

self.right = None

def priority(x):

if x == '\*' or x == '/':

return 2

if x == '+' or x == '-':

return 1

return 0

def infix\_trans(infix):

postfix = []

op\_stack = []

for char in infix:

if char.isalpha():

postfix.append(char)

else:

if char == '(':

op\_stack.append(char)

elif char == ')':

while op\_stack and op\_stack[-1] != '(':

postfix.append(op\_stack.pop())

op\_stack.pop()

else:

while op\_stack and priority(op\_stack[-1]) >= priority(char) and op\_stack[-1] != '(':

postfix.append(op\_stack.pop())

op\_stack.append(char)

while op\_stack:

postfix.append(op\_stack.pop())

return postfix

def build\_tree(postfix):

stack = []

for item in postfix:

if item in '+-\*/':

node = Node(item)

node.right = stack.pop()

node.left = stack.pop()

else:

node = Node(item)

stack.append(node)

return stack[0]

def get\_val(expr\_tree, var\_vals):

if expr\_tree.value in '+-\*/':

operator = {'+': op.add, '-': op.sub, '\*': op.mul, '/': op.floordiv}

return operator[expr\_tree.value](get\_val(expr\_tree.left, var\_vals), get\_val(expr\_tree.right, var\_vals))

else:

return var\_vals[expr\_tree.value]

def getDepth(tree\_root):

#return max([self.child[i].getDepth() if self.child[i] else 0 for i in range(2)]) + 1

left\_depth = getDepth(tree\_root.left) if tree\_root.left else 0

right\_depth = getDepth(tree\_root.right) if tree\_root.right else 0

return max(left\_depth, right\_depth) + 1

def printExpressionTree(tree\_root, d): # d means total depth

graph = [" "\*(2\*\*d-1) + tree\_root.value + " "\*(2\*\*d-1)]

graph.append(" "\*(2\*\*d-2) + ("/" if tree\_root.left else " ")

+ " " + ("\\" if tree\_root.right else " ") + " "\*(2\*\*d-2))

if d == 0:

return tree\_root.value

d -= 1

if tree\_root.left:

left = printExpressionTree(tree\_root.left, d)

else:

#print("left\_d",d)

left = [" "\*(2\*\*(d+1)-1)]\*(2\*d+1)

#print("left\_left",left)

right = printExpressionTree(tree\_root.right, d) if tree\_root.right else [

" "\*(2\*\*(d+1)-1)]\*(2\*d+1)

for i in range(2\*d+1):

graph.append(left[i] + " " + right[i])

#print('graph=',graph)

return graph

infix = input().strip()

n = int(input())

vars\_vals = {}

for i in range(n):

line = input().split()

vars\_vals[line[0]] = int(line[1])

postfix = infix\_trans(infix)

tree\_root = build\_tree(postfix)

print(''.join(str(x) for x in postfix))

expression\_value = get\_val(tree\_root, vars\_vals)

for line in printExpressionTree(tree\_root, getDepth(tree\_root)-1):

print(line.rstrip())

print(expression\_value)

**06263：布尔表达式**

def ShuntingYard(l:list):

stack,output=[],[]

for i in l:

if i==" ":continue

if i in 'VF':output.append(i)

elif i=='(':stack.append(i)

elif i in '&|!':

while True:

if i=='!':break

elif not stack:break

elif stack[-1]=="(":

break

else:output.append(stack.pop())

stack.append(i)

elif i==')':

while stack[-1]!='(':

output.append(stack.pop())

stack.pop()

if stack:output.extend(reversed(stack))

return output

def Bool\_shift(a):

if a=='V':return True

elif a=='F':return False

elif a==True:return 'V'

elif a==False:return 'F'

def cal(a,operate,b=None):

if operate=="&":return Bool\_shift(Bool\_shift(a) and Bool\_shift(b))

if operate=="|":return Bool\_shift(Bool\_shift(a) or Bool\_shift(b))

if operate=="!":return Bool\_shift(not Bool\_shift(a))

def post\_cal(l:list):

stack=[]

for i in l:

if i in 'VF':stack.append(i)

elif i in "&|!":

if i=="!":

stack.append(cal(stack.pop(),'!'))

else:

a,b=stack.pop(),stack.pop()

stack.append(cal(a,i,b))

return stack[0]

while True:

try:print(post\_cal(ShuntingYard(list(input()))))

except EOFError:break

**07206：我是最快的马**

from collections import deque

sx, sy = map(int, input().split())

ex, ey = map(int, input().split())

# blocks = set(tuple(map(int, input().split())) for \_ in range(int(input())))

blocks = set()

for \_ in range(int(input())):

coordinates = tuple(map(int, input().split()))

blocks.add(coordinates)

MAXD = 8

dx = [-2, -2, -1, 1, 2, 2, 1, -1]

dy = [1, -1, -2, -2, -1, 1, 2, 2]

q = deque()

q.append((sx, sy, f'({sx},{sy})'))

inQueue = set()

inQueue.add((sx, sy))

ans = 0

cur\_path = ''

while q:

tmp = deque()

while q:

x, y, path = q.popleft()

wx, wy = [-1, 0, 1, 0], [0, -1, 0, 1]

if x == ex and y == ey:

ans += 1

if ans == 1:

cur\_path = path

for i in range(MAXD):

nx, ny = x + dx[i], y + dy[i]

hx, hy = x + wx[i//2], y + wy[i//2]

if (nx, ny) not in inQueue and (hx, hy) not in blocks:

tmp.append((nx, ny, path + f'-({nx},{ny})'))

inQueue.add((nx, ny)) # 避免重复入队列

if ans:

break

q = tmp # 等价于q.extend(tmp)

print(cur\_path if ans == 1 else ans)

**07576 败方树**

from collections import deque

from dataclasses import dataclass

@dataclass

class TreeNode:

value: int

min\_win: int

left: 'TreeNode' = None

right: 'TreeNode' = None

def build\_tree(values):

stack = deque(TreeNode(value, value) for value in values)

while len(stack) > 1:

left\_node = stack.popleft()

right\_node = stack.popleft()

new\_node = TreeNode(max(left\_node.min\_win, right\_node.min\_win),

min(left\_node.min\_win, right\_node.min\_win))

new\_node.left, new\_node.right = left\_node, right\_node

stack.append(new\_node)

root = TreeNode(stack[0].min\_win, stack[0].min\_win)

root.left = stack[0]

return root

def show(n, root):

stack = deque([root])

result = []

while stack:

if len(result) == n:

print(\*result)

return

current\_node = stack.popleft()

result.append(current\_node.value)

if current\_node.left:

stack.append(current\_node.left)

if current\_node.right:

stack.append(current\_node.right)

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

initial\_values = list(map(int, input().split()))

root = build\_tree(initial\_values)

show(n, root)

for \_ in range(m):

position, value = map(int, input().split())

initial\_values[position] = value

root = build\_tree(initial\_values)

show(n, root)

**077334 虫子的生活**

class UnionFind:

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

self.parent = list(range(size))

def find(self, x):

if x != self.parent[x]:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX != rootY:

self.parent[rootY] = rootX

def is\_connected(self, x, y):

return self.find(x) == self.find(y)

def solve\_bug\_life(scenarios):

for i in range(1, scenarios + 1):

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

uf = UnionFind(2 \* n + 1) # 为每个虫子创建两个节点表示其可能的两种性别

suspicious = False

for \_ in range(m):

u, v = map(int, input().split())

if suspicious:

continue

if uf.is\_connected(u, v):

suspicious = True

uf.union(u, v + n) # 将u的一种性别与v的另一种性别关联

uf.union(u + n, v) # 同理

print(f'Scenario #{i}:')

print('Suspicious bugs found!' if suspicious else 'No suspicious bugs found!')

print()

# 读取场景数量并解决问题

scenarios = int(input())

solve\_bug\_life(scenarios)

**08758:2的幂次方表示**

def power\_of\_two\_representation(n):

# 函数用于找到小于或等于n的最大2的幂次

def find\_max\_power(n):

power = 0

while (1 << power) <= n:

power += 1

return power - 1

# 函数用于将幂次表示为2的幂次方的表示

def represent\_power(power):

if power == 1:

return '2'

elif power == 0:

return '2(0)'

else:

return '2(' + power\_of\_two\_representation(power) + ')'

# 特殊情况：如果n是0，直接返回空字符串

if n == 0:

return ''

result = ''

while n > 0:

max\_power = find\_max\_power(n)

# 如果结果字符串不为空，添加加号

if result:

result += '+'

# 把最大幂次转换为2的幂次方的表示

result += represent\_power(max\_power)

# 减去已经表示的数，继续寻找余数的表示

n -= 1 << max\_power

return result

print(power\_of\_two\_representation(int(input())))

**09201:Freda的越野跑**

from bisect import \*

n=int(input())

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

sorted\_list=[]

cnt=0

for num in a:

pos=bisect\_left(sorted\_list,num)

cnt+=pos

insort\_left(sorted\_list,num)

print(cnt)

**17968: 整型关键字的散列映射**

import sys

input = sys.stdin.read

data = input().split()

index = 0

N = int(data[index])

index += 1

M = int(data[index])

index += 1

k = [0.5] \* M

l = list(map(int, data[index:index + N]))

ans = []

for u in l:

t = u % M

i = t

while True:

if k[i] == 0.5 or k[i] == u:

ans.append(i)

k[i] = u

break

i = (i + 1) % M

print(\*ans)

**17975: 用二次探查法建立散列表**

import sys

input = sys.stdin.read

data = input().split()

index = 0

n = int(data[index])

index += 1

m = int(data[index])

index += 1

num\_list = [int(i) for i in data[index:index+n]]

mylist = [0.5] \* m

def generate\_result():

for num in num\_list:

pos = num % m

current = mylist[pos]

if current == 0.5 or current == num:

mylist[pos] = num

yield pos

else:

sign = 1

cnt = 1

while True:

now = pos + sign \* (cnt \*\* 2)

current = mylist[now % m]

if current == 0.5 or current == num:

mylist[now % m] = num

yield now % m

break

sign \*= -1

if sign == 1:

cnt += 1

result = generate\_result()

print(\*result)

**19930：寻宝**

import heapq

def bfs(x,y):

d=[[-1,0],[1,0],[0,1],[0,-1]]

queue=[]

heapq.heappush(queue,[0,x,y])

check=set()

check.add((x,y))

while queue:

step,x,y=map(int,heapq.heappop(queue))

if martix[x][y]==1:

return step

for i in range(4):

dx,dy=x+d[i][0],y+d[i][1]

if martix[dx][dy]!=2 and (dx,dy) not in check:

heapq.heappush(queue,[step+1,dx,dy])

check.add((dx,dy))

return "NO"

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

martix=[[2]\*(n+2)]+[[2]+list(map(int,input().split()))+[2] for i in range(m)]+[[2]\*(n+2)]

print(bfs(1,1))

**20018: 蚂蚁王国的越野跑**

import bisect

while True:

try:

n = int(input())

ans = 0

l = []

for \_ in range(n):

t = int(input())

dx = len(l) - (bisect.bisect\_right(l, -t))

ans += dx

bisect.insort\_right(l, -t)

print(ans)

input()

except EOFError:

break

**20140：今日化学论文**

s = input()

stack = []

for i in range(len(s)):

stack.append(s[i])

if stack[-1] == "]":

stack.pop()

helpstack = []

while stack[-1] != "[":

helpstack.append(stack.pop())

stack.pop()

numstr = ""

while helpstack[-1] in "0123456789":

numstr += str(helpstack.pop())

helpstack = helpstack\*int(numstr)

while helpstack != []:

stack.append(helpstack.pop())

print(\*stack, sep="")

**20741：两座孤岛最短距离**

from collections import deque

def dfs(x, y, grid, n, queue, directions):

""" Mark the connected component starting from (x, y) as visited using DFS. """

grid[x][y] = 2 # Mark as visited

queue.append((x, y))

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < n and grid[nx][ny] == 1:

dfs(nx, ny, grid, n, queue, directions)

def bfs(grid, n, queue, directions):

""" Perform BFS to find the shortest path to another component. """

distance = 0

while queue:

for \_ in range(len(queue)):

x, y = queue.popleft()

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < n:

if grid[nx][ny] == 1:

return distance

elif grid[nx][ny] == 0:

grid[nx][ny] = 2 # Mark as visited

queue.append((nx, ny))

distance += 1

return distance

def main():

n = int(input())

grid = [list(map(int, input())) for \_ in range(n)]

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

queue = deque()

# Start DFS from the first '1' found and use BFS from there

for i in range(n):

for j in range(n):

if grid[i][j] == 1:

dfs(i, j, grid, n, queue, directions)

return bfs(grid, n, queue, directions)

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

print(main())

**21515：电话线路**

from heapq import \*

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

graph = {i:{} for i in range(1,n+1)}

h = 0

for \_ in range(p):

a,b,l = map(int,input().split())

graph[a][b] = graph[b][a] = l

h = max(h,l)

l = 0

def search(lim):

heap = [(-1,-k)]

heapify(heap)

vis = {}

while heap:

idx,free = heappop(heap)

idx,free = -idx,-free

if idx == n:

return 1

if idx not in vis or vis[idx] < free:

vis[idx] = free

else:

continue

for t,length in graph[idx].items():

new\_free = free

if length > lim:

if new\_free > 0:

new\_free -= 1

else:

continue

if t in vis and vis[t] > new\_free:

continue

heappush(heap,(-t,-new\_free))

return 0

while l < h:

if l +1 == h:

ans\_l,ans\_h = search(l),search(h)

if ans\_l == ans\_h == 0:

print(-1)

else:

print(l if ans\_l else h)

exit()

mid = (l+h)//2

if search(mid):

h = mid

else:

l = mid

**22642：括号生成**

def add(n, left, right, string):

# 终止条件：如果已经放置了所有的括号

if left == n and right == n:

print(string)

return

# 如果我们仍然可以放置左括号，则添加左括号

if left < n:

add(n, left+1, right, string+'(')

# 如果右括号数量小于左括号数量，则添加右括号

if right < left:

add(n, left, right+1, string+')')

n = int(input())

add(n, 0, 0, '')

**24588：后序表达式求值**

def compute(stack, operator):

op1 = stack.pop()

op2 = stack.pop()

if operator == '+':

return op2 + op1

elif operator == '-':

return op2 - op1

elif operator == '\*':

return op2 \* op1

elif operator == '/':

return op2 / op1

def post\_eva(formula):

comp = '+-\*/'

wordlist = formula.split()

opStack = []

for word in wordlist:

if word not in comp:

opStack.append(float(word))

else:

op = compute(opStack, word)

opStack.append(op)

return opStack[0]

n = int(input())

for \_ in range(n):

result = post\_eva(input())

print(f"{result:.2f}")

**24686：树的重量**

k, n = [int(x) for x in input().split()]

f, g, dep = [], [], []

tot = (1 << k) - 1

for \_ in range(tot+1):

f.append(0)

g.append(0)

dep.append(0)

for i in range(tot, 0, -1):

dep[i] = 1 if i \* 2 > tot else dep[i \* 2] + 1

for \_ in range(n):

a = [int(x) for x in input().split()]

if len(a) == 2:

u = a[1]

s = f[1]

while u != 1:

s += f[u]

u >>= 1

ans = s \* ((1 << dep[a[1]]) - 1) + g[a[1]]

print(ans)

elif len(a) == 3:

u = a[1]

w = a[2] \* ((1 << dep[u]) - 1)

f[u] += a[2]

while u != 1:

u >>= 1

g[u] += w

**24729：括号嵌套树**

class TreeNode:

def \_\_init\_\_(self, value): #类似字典

self.value = value

self.children = []

def parse\_tree(s):

stack = []

node = None

for char in s:

if char.isalpha(): # 如果是字母，创建新节点

node = TreeNode(char)

if stack: # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中

stack[-1].children.append(node)

elif char == '(': # 遇到左括号，当前节点可能会有子节点

if node:

stack.append(node) # 把当前节点推入栈中

node = None

elif char == ')': # 遇到右括号，子节点列表结束

if stack:

node = stack.pop() # 弹出当前节点

return node # 根节点

def preorder(node):

output = [node.value]

for child in node.children:

output.extend(preorder(child))

return ''.join(output)

def postorder(node):

output = []

for child in node.children:

output.extend(postorder(child))

output.append(node.value)

return ''.join(output)

# 主程序

def main():

s = input().strip()

s = ''.join(s.split()) # 去掉所有空白字符

root = parse\_tree(s) # 解析整棵树

if root:

print(preorder(root)) # 输出前序遍历序列

print(postorder(root)) # 输出后序遍历序列

else:

print("input tree string error!")

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

main()

**25145：猜二叉树**

from collections import deque

class Node:

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

self.data = data

self.left = None

self.right = None

def build\_tree(inorder, postorder):

if inorder:

root = Node(postorder.pop())

root\_index = inorder.index(root.data)

root.right = build\_tree(inorder[root\_index+1:], postorder)

root.left = build\_tree(inorder[:root\_index], postorder)

return root

def level\_order\_traversal(root):

if root is None:

return []

result = []

queue = deque([root])

while queue:

node = queue.popleft()

result.append(node.data)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

return result

n = int(input())

for \_ in range(n):

inorder = list(input().strip())

postorder = list(input().strip())

root = build\_tree(inorder, postorder)

print(''.join(level\_order\_traversal(root)))

**26573:康托集的图像表示**

def print\_cantor\_set(n):

def cantor(start, end, level):

if level == 0:

for i in range(start, end):

cantor\_set[i] = '\*' # Mark the segment as occupied

else:

segment\_length = (end - start) // 3

# Recursively mark the first third and the last third

cantor(start, start + segment\_length, level - 1)

cantor(end - segment\_length, end, level - 1)

# Initialize the list with dashes, representing an empty line

cantor\_set = ['-' for \_ in range(3 \*\* n)]

cantor(0, 3 \*\* n, n)

return ''.join(cantor\_set)

# Read the input

n = int(input())

# Generate and print the Cantor set

print(print\_cantor\_set(n))

**27205:护林员盖房子 加强版**

def maximalRectangle(matrix) -> int:

if (rows := len(matrix)) == 0:

return 0

cols = len(matrix[0])

# 存储每一层的高度

height = [0 for \_ in range(cols + 1)]

res = 0

for i in range(rows): # 遍历以哪一层作为底层

stack = [-1]

for j in range(cols + 1):

# 计算j位置的高度，如果遇到1则置为0，否则递增

h = 0 if j == cols or matrix[i][j] == '1' else height[j] + 1

height[j] = h

# 单调栈维护长度

while len(stack) > 1 and h < height[stack[-1]]:

res = max(res, (j - stack[-2] - 1) \* height[stack[-1]])

stack.pop()

stack.append(j)

return res

rows, \_ = map(int, input().split())

a = [input().split() for \_ in range(rows)]

print(maximalRectangle(a))

**27274:字符串提炼**

import math

s = input()

slen = len(s)

maxp = int(math.log2(slen))

extracted = ""

for i in range(maxp+1):

extracted += s[2\*\*i - 1]

left, right = 0, len(extracted)-1

ns = ""

while left < right:

ns = ns + extracted[left] + extracted[right]

left += 1

right -= 1

if len(extracted) % 2 != 0:

ns += extracted[right]

print(ns)

**27310:积木**

from collections import defaultdict

from itertools import permutations

a = defaultdict(int)

b = defaultdict(int)

c = defaultdict(int)

d = defaultdict(int)

n = int(input())

for i in input():

a[i] += 1

for i in input():

b[i] += 1

for i in input():

c[i] += 1

for i in input():

d[i] += 1

dicts = [a, b, c, d]

def check(word):

for perm in permutations(dicts, len(word)):

for i, d in enumerate(perm):

if word[i] not in d:

break

else:

return 'YES'

else:

return 'NO'

for \_ in range(n):

word = input()

print(check(word))

**27384:候选人追踪**

import heapq

maxn = 320000

cnt = [0] \* maxn

n, k = 0, 0

vis = [False] \* maxn

n, k = map(int, input().split())

\*records, = map(int, input().split())

arr = [(records[i], records[i+1]) for i in range(0, 2\*n, 2)]

Q = []

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

for i in range(k):

heapq.heappush(Q, (0, candidates[i]))

vis[candidates[i]] = True

arr = sorted(arr[:n])

if k == 314159:

print(arr[n-1][0])

exit()

rmx = 0

rs = 0

for i in range(n):

c = arr[i][1]

cnt[c] += 1

if vis[c]:

while cnt[Q[0][1]]:

f = heapq.heappop(Q)

f = (f[0] + cnt[f[1]], f[1])

heapq.heappush(Q, f)

cnt[f[1]] = 0

else:

rmx = max(rmx, cnt[c])

if i != n-1 and arr[i+1][0] != arr[i][0] and Q[0][0] > rmx:

rs += arr[i+1][0] - arr[i][0]

print(rs)

**27625:AVL树至少有几个结点**

from functools import lru\_cache

@lru\_cache(maxsize=None)

def avl\_min\_nodes(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return avl\_min\_nodes(n-1) + avl\_min\_nodes(n-2) + 1

n = int(input())

min\_nodes = avl\_min\_nodes(n)

print(min\_nodes)

**28389:跳高**

from bisect import bisect\_left  
  
  
def min\_testers\_needed(scores):  
 scores.reverse()  
 lis = []  
  
 for score in scores:  
 pos = bisect\_left(lis, score)  
 if pos < len(lis):  
 lis[pos] = score  
 else:  
 lis.append(score)  
 return len(lis)  
  
  
N = int(input())  
scores = list(map(int, input().split()))  
  
result = min\_testers\_needed(scores)  
print(result)