cheat sheet（DSA）

吴迪 2400011611

一 语法

1. round和int的区别：

1. **功能定义**
   - `int()`：将数值转换为整数，直接**截断小数部分**（取整）。
     - 示例：`int(3.7) → 3`，`int(-2.9) → -2`。
   - `round()`：将数值**四舍五入**到最接近的整数。
     - 示例：`round(3.7) → 4`，`round(-2.9) → -3`。

2. `.strip()`:

A method in Python (and similar functions in other languages) that **removes leading/trailing whitespace or specified characters** from a string.
Example:

```
text = "   Hello, World!   "
print(text.strip())  # Output: "Hello, World!"
```

3. 修改递归深度

```
import sys
sys.setrecursionlimit(1>>30)
```

4. 一次性读取输入

```
import sys
data=sys.stdin.read().split()
#以字符串方式读入
```

建一个以每行输入的字符串为元素的列表:sys.stdin.readlines()

5. 缓存

```
from functools import lru_cache
@lru_cache(maxsize = None)
```

6. 判断数据类型

```
x = 5
print(isinstance(x, int))  # True
```

```
char = '5'
print(char.isdigit())  # 输出: True

char = 'Ⅴ'  # 罗马数字 V
print(char.isdigit())  # 输出: False (注意：罗马数字不是被识别为数字)
char = '5'
print(char.isnumeric())  # 输出: True

char = 'Ⅴ'  # 罗马数字 V
print(char.isnumeric())  # 输出: True
```

7.itertools

```
import itertools

def generate_binary_lists(length):
    # 使用 itertools.product 生成长度为 'length' 的所有0,1组合
    return list(itertools.product([0, 1], repeat=length))

import itertools

def generate_permutations(elements):
    # 使用 itertools.permutations 生成所有排列
    return list(itertools.permutations(elements))

combinations(list,k)        #生成list的k元组合（无序）（每个以元组形式存在）
```

8. math

1. **数值表示相关**
   - math.ceil(x)：返回不小于 x 的最小整数。
   - **math.factorial(x)：返回 x 的阶乘（x 必须是非负整数）。**
   - math.floor(x)：返回不大于 x 的最大整数。
   - math.log(x[, base])：返回 x 的自然对数，base 参数可选。
   - math.pow(x, y)：返回 x 的 y 次幂。
   - math.sqrt(x)：返回 x 的平方根
2. **角度转换**
   - math.degrees(x)：将弧度 x 转换为角度。
   - math.radians(x)：将角度 x 转换为弧度。

9.切片

```
sequence[start:stop:step]
```

10. 保留小数

```
print(f'{num:.2f}')
```

11. 进制转换

```
decimal_num = 10

# 十进制转二进制
binary_str = bin(decimal_num)       # 结果是 '0b1010'

# 十进制转八进制
octal_str = oct(decimal_num)        # 结果是 '0o12'

# 十进制转十六进制
hex_str = hex(decimal_num)          # 结果是 '0xa'

# 二进制转十进制
binary_str = "1010"
decimal_num = int(binary_str, 2)  # 结果是 10
```

```
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)
```

12. set的运算

运算类型	运算符	方法	描述	示例	结果
创建集合	`{}` / `set()`	-	创建集合	`s = {1, 2, 3}`	`{1, 2, 3}`

运算类型	运算符	方法	描述	示例	结果
并集	`|`	`union()`	返回所有唯一元素	`{1,2} | {2,3}`	`{1,2,3}`
交集	`&`	`intersection()`	返回共同元素	`{1,2} & {2,3}`	`{2}`
差集	`-`	`difference()`	返回在第一个集合但不在第二个集合的元素	`{1,2} - {2,3}`	`{1}`

```
# 方法1: ^ 运算符
c = a ^ b  # {1, 2, 4, 5}

# 方法2: symmetric_difference() 方法
d = a.symmetric_difference(b)  # {1, 2, 4, 5}
```

二 算法

1. Floyd-Warshall

```
#oj05443 Rabbits and sakura
p=int(input())
node=[input() for i in range(p)]
d={i:j for j,i in enumerate(node)}
dist=[[float('inf')]*p for _ in range(p)]
road=[['']*p for _ in range(p)]
for i in range(p):
    dist[i][i]=0
for _ in range(int(input())):
    a,b,w=input().split()
    dist[d[a]][d[b]]=int(w)
    dist[d[b]][d[a]]=int(w)
    road[d[a]][d[b]]=f'{a}->({w})->'
    road[d[b]][d[a]]=f'{b}->({w})->'
for i in range(p):
    for j in range(p):
        for k in range(p):
            if dist[i][k]+dist[k][j]<dist[i][j]:
                dist[i][j]=dist[i][k]+dist[k][j]
                road[i][j]=road[i][k]+road[k][j]
for _ in range(int(input())):
    x,y=input().split()
    print(road[d[x]][d[y]]+y)
```

floyd算法不仅可以算最短路，还可以按同样dp的思路记录路径

2. shunting yard

用于中缀表达式转后缀表达式

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = [] # Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            #opStack.push(token)
            opStack.append(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            #while (not opStack.is_empty()) and (prec[opStack.peek()] >= prec[token]):
            while opStack and (prec[opStack[-1]] >= prec[token]):
                postfixList.append(opStack.pop())
            #opStack.push(token)
            opStack.append(token)

    #while not opStack.is_empty():
    while opStack:
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

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

def inorder_to_postorder(s):
    st=[]
    postfix=[]
    for char in s:
        if char in ['True','False']:
            postfix.append(char)
        if char=='(':
            st.append(char)
        elif char in priority:
            while st and priority[st[-1]]>priority[char]:
                postfix.append(st.pop())
```

```
            st.append(char)
        elif char==')':
            while st and st[-1]!='(':
                postfix.append(st.pop())
            st.pop()
    while st:
        postfix.append(st.pop())
    return postfix
```

```
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))
```

后序表达式求值

```
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in '+-*/':
            # 弹出栈顶的两个元素
            right_operand = stack.pop()
            left_operand = stack.pop()
            # 执行运算
            if token == '+':
                stack.append(left_operand + right_operand)
            elif token == '-':
                stack.append(left_operand - right_operand)
            elif token == '*':
                stack.append(left_operand * right_operand)
            elif token == '/':
                stack.append(left_operand / right_operand)
        else:
            # 将操作数转换为浮点数后入栈
            stack.append(float(token))

    # 栈顶元素就是表达式的结果
    return stack[0]
```

3. Kruscal

以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
    - 选择排序后的边集中权重最小的边。
    - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

```
def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
```

```
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))

    return minimum_spanning_tree
```

4. sort

1. merge sort

```
def MergeSort(arr):
    if len(arr)<=1: return arr
    else:
        l,r=arr[:len(arr)//2],arr[len(arr)//2:]
        return Merge(MergeSort(l),MergeSort(r))
def Merge(l,r):
    res=[]
    i,j=0,0
    while i<len(l) and j<len(r):
        if l[i]<=r[j]:
            res.append(l[i])
            i+=1
        else:
            res.append(r[j])
            j+=1
    res+=l[i:]+r[j:]
    return res
```

```
def merge(a1,a2):
    arr=[]
    i,j,k=0,0,0
    while i<len(a1) and j<len(a2):
        if a1[i]<a2[j]:
            arr.append(a1[i])
            i+=1
        else:
            arr.append(a2[j])
```

```
            j+=1
            k+=len(a1)-i
    arr+=a1[i:]
    arr+=a2[j:]
    return arr,k



def mergesort(a,n):
    if n<=1:
        return a,0
    middle=n//2
    a1,ans1=mergesort(a[:middle],middle)
    a2,ans2=mergesort(a[middle:],n-middle)
    arr,ans=merge(a1,a2)
    return arr,ans+ans1+ans2

while True:
    n=int(input())
    if n==0:
        break
    a=[]
    for _ in range(n):
       a.append(int(input()))
    arr,ans=mergesort(a,n)
    print(ans)
```

2. quick sort

```
def quickSort(arr):
    if len(arr)<=1:
        return arr
    else:
        mid=arr[len(arr)//2]
        l,m,r=[],[],[]
        for i in arr:
            if i < mid : l.append(i)
            elif i > mid : r.append(i)
            else : m.append(i)
        return quickSort(l) + m + quickSort(r)
```

5. kmp

```
"""
compute_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，
其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。
该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。
```

```
"""
def compute_lps(pattern):
    """
    计算pattern字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表
    :param pattern: 模式字符串
    :return: lps表
    """

    m = len(pattern)
    lps = [0] * m  # 初始化lps数组
    length = 0  # 当前最长前后缀长度
    for i in range(1, m):  # 注意i从1开始，lps[0]永远是0
        while length > 0 and pattern[i] != pattern[length]:
            length = lps[length - 1]  # 回退到上一个有效前后缀长度
        if pattern[i] == pattern[length]:
            length += 1
        lps[i] = length

    return lps

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    if m == 0:
        return 0
    lps = compute_lps(pattern)
    matches = []

    # 在 text 中查找 pattern
    j = 0  # 模式串指针
    for i in range(n):  # 主串指针
        while j > 0 and text[i] != pattern[j]:
            j = lps[j - 1]  # 模式串回退
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            matches.append(i - j + 1)  # 匹配成功
            j = lps[j - 1]  # 查找下一个匹配

    return matches


text = "ABABABABCABABABABCABABABABC"
pattern = "ABABCABAB"
index = kmp_search(text, pattern)
print("pos matched：", index)
# pos matched：  [4, 13]
```

判断周期：

```
for i,j in enumerate(lps):
    if j>0 and (i+1)%(i-j+1)==0:
            print(i+1,(i+1)//(i-j+1))
```

注意构造lps一定是从1开始的

6. parse tree

后序表达式建树

```
def build(s):
    st=[]
    for i in s:
        cur=Node(i)
        if i in priority:
            cur.right=st.pop()
            if i!='not':
                cur.left=st.pop()
        st.append(cur)
    return st[0]
```

由解析树还原最简化括号表达式

```
def t(root):
    if not root:
        return []
    if root.val=='not':
        if root.right.val in priority:
            return [root.val]+['(']+t(root.right)+[')']
        return [root.val]+t(root.right)
    r,l=[],[]
    if root.left:
        if root.left.val in priority and priority[root.left.val]<priority[root.val]:
            l=['(']+t(root.left)+[')']
        else:
            l=t(root.left)
    if root.right:
        if root.right.val in priority and priority[root.right.val]<=priority[root.val]:
            r=['(']+t(root.right)+[')']
        else:
            r=t(root.right)
    return l+[root.val]+r
```

还原完全括号表达式

```
def printexp(root):
    if not root:
        return ''
    return '('+printexp(root.left)+root.val+printexp(root.right)+')'
```

层次遍历

```python
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []

        result = []
        queue = deque([root])

        while queue:
            level_size = len(queue)
            level = []

            for _ in range(level_size):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(level)

        return result
```

```python
#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
```

7. Trie

```
# Trie implementation in Python

class TrieNode:
    def __init__(self):
        # pointer array for child nodes of each node
        self.childNode = [None] * 26
        self.wordCount = 0

def insert_key(root, key):
    # Initialize the currentNode pointer with the root node
    currentNode = root

    # Iterate across the length of the string
    for c in key:
        # Check if the node exist for the current character in the Trie.
        if not currentNode.childNode[ord(c) - ord('a')]:
            # If node for current character does not exist
            # then make a new node
            newNode = TrieNode()
            # Keep the reference for the newly created node.
            currentNode.childNode[ord(c) - ord('a')] = newNode
        # Now, move the current node pointer to the newly created node.
        currentNode = currentNode.childNode[ord(c) - ord('a')]
    # Increment the wordEndCount for the last currentNode
    # pointer this implies that there is a string ending at currentNode.
    currentNode.wordCount += 1

def search_key(root, key):
    # Initialize the currentNode pointer with the root node
    currentNode = root

    # Iterate across the length of the string
    for c in key:
        # Check if the node exist for the current character in the Trie.
        if not currentNode.childNode[ord(c) - ord('a')]:
            # Given word does not exist in Trie
            return False
        # Move the currentNode pointer to the already existing node for current character.
        currentNode = currentNode.childNode[ord(c) - ord('a')]

    return currentNode.wordCount > 0

def delete_key(root, word):
    currentNode = root
    lastBranchNode = None
    lastBrachChar = 'a'

    for c in word:
        if not currentNode.childNode[ord(c) - ord('a')]:
            return False
        else:
```

```
            count = 0
            for i in range(26):
                if currentNode.childNode[i]:
                    count += 1
            if count > 1:
                lastBranchNode = currentNode
                lastBrachChar = c
            currentNode = currentNode.childNode[ord(c) - ord('a')]

    count = 0
    for i in range(26):
        if currentNode.childNode[i]:
            count += 1

    # Case 1: The deleted word is a prefix of other words in Trie.
    if count > 0:
        currentNode.wordCount -= 1
        return True

    # Case 2: The deleted word shares a common prefix with other words in Trie.
    if lastBranchNode:
        lastBranchNode.childNode[ord(lastBrachChar) - ord('a')] = None
        return True
    # Case 3: The deleted word does not share any common prefix with other words in Trie.
    else:
        root.childNode[ord(word[0]) - ord('a')] = None
        return True
```

8. huffman

```
n=int(input())
class Node:
    def __init__(self,freq,letter,left=None,right=None):
        self.freq=freq
        self.letter=letter
        self.left=left
        self.right=right
    def mc(self,node):
        if not node.left and not node.right:
            return node.letter
        return min(self.mc(node.left),self.mc(node.right))
    def __lt__(self,other):
        if self.freq!=other.freq:
            return self.freq<other.freq
        return self.mc(self)<self.mc(other)
import heapq
q=[]
for _ in range(n):
    a,b=input().split()
    b=int(b)
    heapq.heappush(q,Node(b,a))
```

```
while len(q)>1:
    l,r=heapq.heappop(q),heapq.heappop(q)
    c=Node(l.freq+r.freq,None,l,r)
    heapq.heappush(q,c)
root=heapq.heappop(q)
code={}
def traverse(node,s):
    if node.letter:
        code[node.letter]=s
        return
    traverse(node.left,s+'0')
    traverse(node.right,s+'1')
traverse(root,'')
def encode(s):
    ans=''
    for i in s:
        ans+=code[i]
    return ans
def decode(s):
    ans=''
    cur=root
    for i in s:
        if i=='1':
            cur=cur.right
        else:
            cur=cur.left
        if cur.letter:
            ans+=cur.letter
            cur=root
    return ans
while True:
    try:
        s=input()
        if s[:1] in '01':
            print(decode(s))
        else:
            print(encode(s))
    except EOFError:
        break
```

9. kosaraju

对于图G，强连通单元C为最大的顶点子集$C \subset V$，其中对于每一对顶点$v, w \in C$，都有一条从v到w的路径和一条从w到v的路径。

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
```

```
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
    print(scc)

"""
Strongly Connected Components:
[0, 3, 2, 1]
[6, 7]
[5, 4]

"""
```

10. disjoint set

短码版本

```
p=[i for i in range(26)]
def find(i):
    if p[i]!=i:
        p[i]=find(p[i])
    return p[i]
#join
p[find(x)]=find(y)
```

```
class DisjSet:
    def __init__(self, n):
        # Constructor to create and initialize sets of n items
        self.rank = [1] * n
        self.parent = [i for i in range(n)]


    # Finds set of given item x
    def find(self, x):

        # Finds the representative of the set that x is an element of
        if (self.parent[x] != x):

            # if x is not the parent of itself
            # Then x is not the representative of its set
            self.parent[x] = self.find(self.parent[x])

            # so we recursively call Find on its parent
            # and move i's node directly under the
            # representative of this set

        return self.parent[x]


    # Do union of two sets represented by x and y.
    def Union(self, x, y):

        # Find current sets of x and y
        xset = self.find(x)
        yset = self.find(y)

        # If they are already in same set
        if xset == yset:
            return

        # Put smaller ranked item under
        # bigger ranked item if ranks are different
        if self.rank[xset] < self.rank[yset]:
            self.parent[xset] = yset

        elif self.rank[xset] > self.rank[yset]:
            self.parent[yset] = xset

        # If ranks are same, then move y under x (doesn't matter
```

```
    # which one goes where) and increment rank of x's tree
        else:
            self.parent[yset] = xset
            self.rank[xset] = self.rank[xset] + 1
```

```
#food chain

n,k=map(int,input().split())
p=[i for i in range(n*3)]
def f(i):
    if p[i]!=i:
        p[i]=f(p[i])
    return p[i]
ans=0
for _ in range(k):
    d,x,y=map(int,input().split())
    if x>n or y>n:
        ans+=1
        continue
    x-=1
    y-=1
    if d==1:
        if f(x)==f(y+n) or f(x)==f(y+2*n):
            ans+=1
            continue
        p[f(x)]=f(y)
        p[f(y+n)]=f(x+n)
        p[f(y+2*n)]=f(x+2*n)
    else:
        if f(x)==f(y) or f(x)==f(y+n*2):
            ans+=1
            continue
        p[f(y+n)]=f(x)
        p[f(x+2*n)]=f(y)
        p[f(x+n)]=f(y+2*n)

print(ans)
```

11. hash table

12. 差分数组

差分数组（**Difference Array**）是一种用于**高效处理区间更新**的数据结构。给定原数组 `A = [a₀, a₁, ..., aₙ₋₁]`，其差分数组 `D` 定义为：

$$\begin{cases} d_0 = a_0 \\ d_i = a_i - a_{i-1} \quad \text{for } 1 \le i < n \end{cases}$$

1. 对原数组区间 `[l, r]` 增加 `k`：

```
def update(l, r, k):
    diff[l] += k
    if r + 1 < n:  # 防止越界
        diff[r + 1] -= k
```

时间复杂度：$O(1)$

2. 通过前缀和还原操作后的数组：

```
def restore():
    res = [0] * n
    res[0] = diff[0]
    for i in range(1, n):
        res[i] = res[i-1] + diff[i]
    return res
```

时间复杂度：$O(n)$

用于处理矩阵的子矩阵增减操作：

```
class Diff2D:
    def __init__(self, m, n):
        """初始化二维差分数组"""
        self.m = m
        self.n = n
        # 创建(m+2) x (n+2)的差分数组（多两行两列处理边界）
        self.diff = [[0] * (n + 2) for _ in range(m + 2)]

    def update(self, x1, y1, x2, y2, c):
        """
        更新子矩阵区域 [x1, y1] 到 [x2, y2]
        所有元素增加c
        """
        self.diff[x1+1][y1+1] += c
        self.diff[x1+1][y2+2] -= c
        self.diff[x2+2][y1+1] -= c
        self.diff[x2+2][y2+2] += c

    def build_matrix(self):
        """计算最终矩阵"""
        matrix = [[0] * self.n for _ in range(self.m)]

        # 计算二维前缀和
        for i in range(1, self.m+1):
            for j in range(1, self.n+1):
                # 累加四个方向的差分值
                self.diff[i][j] += self.diff[i][j-1] + self.diff[i-1][j] - self.diff[i-1]
[j-1]
                matrix[i-1][j-1] = self.diff[i][j]
```

```
        return matrix


# 使用示例
if __name__ == "__main__":
    # 创建3x3矩阵的差分数组
    diff2d = Diff2D(3, 3)

    # 更新整个矩阵+5
    diff2d.update(0, 0, 2, 2, 5)

    # 更新左上角2x2区域+3
    diff2d.update(0, 0, 1, 1, 3)

    # 更新右下角1x1区域-2
    diff2d.update(2, 2, 2, 2, -2)

    # 获取最终矩阵
    result = diff2d.build_matrix()

    # 打印结果
    for row in result:
        print(row)
    """
    输出:
    [8, 8, 5]
    [8, 8, 5]
    [5, 5, 3]
    """
```

13. heap

```
class BinaryHeap:
    def __init__(self):
        self._heap = []

    def _perc_up(self, i):
        while (i - 1) // 2 >= 0:
            parent_idx = (i - 1) // 2
            if self._heap[i] < self._heap[parent_idx]:
                self._heap[i], self._heap[parent_idx] = (
                    self._heap[parent_idx],
                    self._heap[i],
                )
            i = parent_idx

    def insert(self, item):
        self._heap.append(item)
        self._perc_up(len(self._heap) - 1)

    def _perc_down(self, i):
        while 2 * i + 1 < len(self._heap):
            sm_child = self._get_min_child(i)
```

```
            if self._heap[i] > self._heap[sm_child]:
                self._heap[i], self._heap[sm_child] = (
                    self._heap[sm_child],
                    self._heap[i],
                )
            else:
                break
            i = sm_child

    def _get_min_child(self, i):
        if 2 * i + 2 > len(self._heap) - 1:
            return 2 * i + 1
        if self._heap[2 * i + 1] < self._heap[2 * i + 2]:
            return 2 * i + 1
        return 2 * i + 2

    def delete(self):
        self._heap[0], self._heap[-1] = self._heap[-1], self._heap[0]
        result = self._heap.pop()
        self._perc_down(0)
        return result

    def heapify(self, not_a_heap):
        self._heap = not_a_heap[:]
        i = len(self._heap) // 2 - 1    # 超过中点的节点都是叶子节点
        while i >= 0:
            print(f'i = {i}, {self._heap}')
            self._perc_down(i)
            i = i - 1
```

14. kadane

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        max_end_here=-float('inf')
        max_so_far=-float('inf')
        for num in nums:
            max_end_here=max(num,max_end_here+num)
            max_so_far=max(max_end_here,max_so_far)
        return max_so_far
```

15. 子集

```
class Solution(object):
    def subsets(self, nums):
        res=[[]]
        for i in nums:
            for j in range(len(res)):
                res.append(res[j]+[i])
        return res
```

16. binary search

Function	Returns	When x exists	When x missing
`bisect_left(a, x)`	First position ≥ x	First occurrence	Insert position
`bisect_right(a, x)`	First position > x	Index after last occurrence	Same as `bisect_left`

17.binary search

```
n,k=map(int,input().split())
a=[int(float(input())*100) for _ in range(n)]
def can(m):
    cnt=0
    for i in a:
        cnt+=i//m
        if cnt>=k:
            return True
    return False
i,j=0,max(a)+1
while i<j-1:
    if can((i+j)//2):
        i=(i+j)//2
    else:
        j=(i+j)//2
print(f'{i/100:.2f}')
```

18. avl

```
from functools import lru_cache

@lru_cache(maxsize=None)
def min_nodes(h):
    if h == 0: return 0
    if h == 1: return 1
    return min_nodes(h-1) + min_nodes(h-2) + 1

def max_height(n):
```

```
    h = 0
    while min_nodes(h) <= n:
        h += 1
    return h - 1

n = int(input())
print(max_height(n))
```

19. bellman-ford

```
def bellman_ford(graph, V, source):
    # 初始化距离
    dist = [float('inf')] * V
    dist[source] = 0

    # 松弛 V-1 次
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # 检测负权环
    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("图中存在负权环")
            return None

    return dist

# 图是边列表，每条边是 (起点, 终点, 权重)
edges = [
    (0, 1, 5),
    (0, 2, 4),
    (1, 3, 3),
    (2, 1, 6),
    (3, 2, -2)
]
V = 4
source = 0

print(bellman_ford(edges, V, source))
```

20.dijkstra

```
#Saving Tang Monk(the most tough problem in pre-problem list)
import heapq
dire=[(1,0),(-1,0),(0,1),(0,-1)]

while True:
    n,m=map(int,input().split())
    if n==m==0:
        break
```

```
    ma=[list(input()) for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if ma[i][j]=='K':
                x1,y1=i,j
            elif ma[i][j]=='T':
                ma[i][j]=m+1
            elif ma[i][j].isnumeric():
                ma[i][j]=int(ma[i][j])
    ans=0
    kills=[(0,x1,y1,[])]
    for tar in range(1,m+2):
        new=[]
        min_now=float('inf')
        for kill1 in kills:
            q = [kill1]
            visited = set()
            while q:
                step, x, y, kill = heapq.heappop(q)
                kill.sort()
                if step>min_now+5:
                    break
                if (x, y, tuple(kill)) not in visited:
                    visited.add((x, y, tuple(kill)))
                    if ma[x][y] == tar:
                        min_now=min(min_now,step)
                        new.append((step,x,y,kill[:]))
                    for dx,dy in dire:
                        if 0<=x+dx<n and 0<=y+dy<n and ma[x+dx][y+dy]!='#':
                            if ma[x+dx][y+dy]=='S':
                                if (x+dx,y+dy) not in kill and
(x+dx,y+dy,tuple(list(sorted(kill+[(x+dx,y+dy)])))) not in visited:
                                    heapq.heappush(q,(step+2,x+dx,y+dy,kill[:]+
[(x+dx,y+dy)]))
                                if (x+dx,y+dy) in kill and (x+dx,y+dy,tuple(kill)) not in
visited:
                                    heapq.heappush(q,(step+1,x+dx,y+dy,kill[:]))
                            elif (x+dx,y+dy,tuple(kill)) not in visited:
                                heapq.heappush(q,(step+1,x+dx,y+dy,kill[:]))
        ans=min_now
        kills=new
    if ans<1e9:
        print(ans)
    else:
        print('impossible')
```

三 各种概念

1. 二叉搜索树（Binary Search Tree，BST），

它是映射的另一种实现。我们感兴趣的不是元素在树中的确切位置，而是如何利用二叉树结构提供高效的搜索。

二叉搜索树依赖于这样一个性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。

---

2. 位运算

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

与运算的用途：

清零

如果想将一个单元清零，即使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。

取一个数的指定位

比如取数 X=1010 1110 的低4位，只需要另找一个数Y，令Y的低4位为1，其余位为0，即Y=0000 1111，然后将X与Y进行按位与运算X&Y=0000 1110即可得到X的指定位。

判断奇偶

只要根据最末位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用if ((a & 1) == 0)来判断a是不是偶数。

出现过的bug

1. 5.31 把行号m和列号n看反了
2. 5.31 almost犯了和之前一样的错误：在结束接受一组测试数据之前提前break掉；循环内部变量i，j，k混用
3. oj04093
4. T02448:用时34分钟，由于要求字典序，所以对dire的顺序有一定的要求，才开始记不得'A'的ord了，就print看了一下，结果 后面忘删了，提交的时候卡了半天才发现，看来之后还是要新开一个test.py
5. T06648: Sequence
6. lc136:位运算

7. lc3478
8. lc146:lru缓存，极易出bug
9. lc3510:完全没做出来
10. lc78:超短代码
11. T28046: 词梯：做过的题，还是 debug 了半天，要注意浅拷贝
12. debug 了半天，然后发现是因为新定义了一个 c 覆盖了原来的值。 python 的变量命名真的不能只图省事
13. 列表和集合都是unhashable
14. visited中加入tuple套tuple时要记得排序（saving tang monk）
15. 浅拷贝
16. 当前队列的中位数：当一道题既有懒删除又要不断添加元素时，一定要注意改变的元素是不是已经被'删除'了，如果被删除了就不必更新指标，否则wa。所以最好在每次改变指标前都尝试懒删除一次，保证对顶元素不是已经被删除的元素。
17. 尝试删除，查询列表或字典元素时，要先确认是否为空，否则RE