双指针：

```python
class Solution:
    def twoSum(self, price: List[int], target: int) -> List[int]:
        l, r = 0, len(price) - 1
        while l < r:
            s = price[l] + price[r]
            if s == target:
                return [l, r]
            elif s < target:
                l += 1
            else:
                r -= 1
        return []
```

二分：

```python
def search(self, nums: List[int], target: int) -> int:
    l,r=0,len(nums)-1
    while l <= r:
        mid=(l+r)//2
        if target >= nums[mid]:
            l=mid+1
        if target <= nums[mid]:
            r=mid-1
        if target == nums[mid]:
            return mid
    return -1
```

Febonacci（dp）：

```python
def fibonacci(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[0] = 0  # 初始化
    dp[1] = 1  # 初始化
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]  # 状态转移方程
    return dp[n]
```

从后开始的 dp：

```python
def replaceElements(self, arr: List[int]) -> List[int]:
    e=-1
    for i in reversed(range(len(arr))):
        x=arr[i]
        arr[i]=e
        e=max(x,e)
    return arr
```

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots}}}$$

：

```python
class Solution:
    def fraction(self, cont: List[int]) -> List[int]:
        n,m = 0, 1
        for a in cont[::-1]:
            n,m = m, (m * a + n)
        return [m, n]
```

比较经典的 dp（?

```python
def maxProfit(self, prices: List[int]) -> int:
    n=len(prices)
    dp=[0]*n
    minp=prices[0]
    for i in range(1,n):
        minp=min(minp,prices[i])
        dp[i]=max(dp[i-1],prices[i]-minp)
    return dp[-1]
```

合并区间：

```python
def merge(self, intervals: List[List[int]]) -> List[List[int]]:
    ans=[]
    intervals.sort(key=lambda x:x[0])
    for i in intervals:
        if not ans or ans[-1][1] < i[0]:
            ans.append(i)
        else:
            ans[-1][1] = max(ans[-1][1],i[1])
    return ans
```

矩阵*+：

```python
def mat():
    r,c=map(int,input().split())
    matrix=[list(map(int,input().split())) for _ in range(r)]
    return matrix,r,c

A,rA,cA=mat()
B,rB,cB=mat()
C,rC,cC=mat()

if cA != rB:
    print("Error!")
    exit()

ans=[[0]*cB for _ in range(rA)]
if cA ==rB:
    for i in range(rA):
        for j in range(cB):
            for k in range(cA):
                ans[i][j]+=A[i][k]*B[k][j]

if len(ans) != rC or len(ans[0]) != cC:
    print("Error!")
    exit()

res=[[0]*cC for _ in range(rC)]
if len(ans) == rC and len(ans[0]) == cC:
    for i in range(rC):
        for j in range(cC):
            res[i][j]=ans[i][j]+C[i][j]

for _ in res:
    print(" ".join(map(str,_)))
```

Stack：

```python
while True:
    try:
        s=input()
        mark=[" "]*len(s)
        stack=[]

        for i,ch in enumerate(s):
            if ch == "(":
                stack.append(i)
            elif ch == ")":
                if stack:
                    stack.pop()
                else:
                    mark[i]="?"

        if stack:
            for j in stack:
                mark[j]="$"

        print(s)
        print("".join(mark))
    except EOFError:
        break
```

进制：

bin(x)	十进制转二进制，返回字符串	bin(10)	'0b1010'
oct(x)	十进制转八进制，返回字符串	oct(9)	'0o11'
hex(x)	十进制转十六进制，返回字符串	hex(255)	'0xff'
int(s, b)	任意进制转十进制	int('11', 8)	9

```python
a=int(input())
stack=[]
if a == "0":
    print(0)
else:
    while a >0:
        stack.append(a%8)
        a//=8
    while stack:
        print(stack.pop(),end="")
```

找最大值：（滑动窗口）

```python
def maximumUniqueSubarray(nums):
    seen = set()
    left = 0
    current_sum = 0
    max_sum = 0

    for right in range(len(nums)):
        while nums[right] in seen:
            seen.remove(nums[left])
            current_sum -= nums[left]
            left += 1
        seen.add(nums[right])
        current_sum += nums[right]
        max_sum = max(max_sum, current_sum)

    return max_sum
```

```python
class Solution:
    def maxSum(self, nums: List[int]) -> int:
        max_sum = 0
        seen = set(nums)
        for num in seen:
            max_sum = max(max_sum, num + max_sum)
        if max_sum == 0:
            return max(nums)
        return max_sum
```

1 前序遍历（Preorder）

```python
def preorder(root):
    if root:
        print(root.val, end=' ')
        preorder(root.left)
        preorder(root.right)
```

2 中序遍历（Inorder）

```python
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val, end=' ')
        inorder(root.right)
```

3 后序遍历（Postorder）

```python
def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val, end=' ')
```

反转链表：

```python
Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        vals = []
        current_node = head
        while current_node is not None:
            vals.append(current_node.val)
            current_node = current_node.next
        return vals == vals[::-1]
```

二叉树深度（后序&层序）：献给阿吉尔侬的鲜花（bfs）：

```python
def maxDepth(self, root: Optional[TreeNode]) -> int:
    if not root: return 0
    return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

```python
def maxDepth(self, root: TreeNode) -> int:
    if not root: return 0
    queue, res = [root], 0
    while queue:
        tmp = []
        for node in queue:
            if node.left: tmp.append(node.left)
            if node.right: tmp.append(node.right)
        queue = tmp
        res += 1
    return res
```

```python
from collections import deque

def bfs(maps,start,end,R,C):
    dir=[(-1,0),(1,0),(0,-1),(0,1)]
    visited=[[False]*C for _ in range(R)]
    queue=[]
    queue.append((start[0],start[1],0))
    deq=deque(queue)
    visited[start[0]][start[1]]=True

    while deq:
        x,y,step=deq.popleft()
        if (x,y)==end:
            return step
        for x_,y_ in dir:
            nx,ny=x+x_,y+y_
            if 0 <= nx < R and 0 <= ny < C:
                if not visited[nx][ny] and maps[nx][ny] != "#":
                    visited[nx][ny]=True
                    deq.append((nx,ny,step+1))
    return False

T=int(input())
for _ in range(T):
    R,C=map(int,input().split())
    maps=[]
    for _ in range(R):
        r=input()
        maps.append(r)

    for i in range(R):
        for j in range(C):
            if maps[i][j]=="S":
                start=(i,j)
            elif maps[i][j]=="E":
                end=(i,j)

    res=bfs(maps,start,end,R,C)
    if res:
        print(res)
    else:
        print("oop!")
```

```
    A
   / \
  B   C
 / \   \
D   E   F
```

前序遍历	根 → 左子树 → 右子树	A B D E C F
中序遍历	左子树 → 根 → 右子树	D B E A C F
后序遍历	左子树 → 右子树 → 根	D E B F C A

二叉树直径：

```python
def __init__(self):
    self.max = 0

def diameterOfBinaryTree(self, root: TreeNode) -> int:
    self.depth(root)

    return self.max

def depth(self, root):
    if not root:
        return 0
    l = self.depth(root.left)
    r = self.depth(root.right)
    '''每个结点都要去判断左子树+右子树的高度是否大于self.max，更新最大值'''
    self.max = max(self.max, l+r)

    # 返回的是高度
    return max(l, r) + 1
```

平衡二叉搜索树：将有序数组转换为二叉树

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def sortedArrayToBST(self, nums):
        if not nums:
            return None

        mid = len(nums) // 2
        root = TreeNode(nums[mid])  # 中间的值作为根

        # 递归构建左右子树
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid+1:])

        return root
```

FBI 树：

```python
def get_type(s):
    if all(c == '0' for c in s):
        return 'B'
    elif all(c == '1' for c in s):
        return 'I'
    else:
        return 'F'

def build(s):
    if len(s) == 1:
        return get_type(s)
    mid = len(s) // 2
    left = build(s[:mid])
    right = build(s[mid:])
    return left + right + get_type(s)

# 输入读取
N = int(input())
s = input().strip()

# 构造并输出
result = build(s)
print(result)
```

节点个数：

```python
def countNodes(root):
    if not root:
        return 0
    return 1 + countNodes(root.left) + countNodes(root.right)
```

zip 用法：

```
n=int(input())
res=[]
ind=[]
for _ in range(n):
    c,m,e=map(int,input().split())
    res.append([c+m+e,c,m,e])
    ind.append(_)
    b=list(zip(res,ind))
    b.sort(key=lambda x:[-x[0][0],-x[0][1]])

for i,j in enumerate(b[:5],start=1):
    print(f"{j[1]+1} {j[0][0]}")
```

counter 用法：

```
from collections import Counter
n,m = map(int, input().split())
tags = list(map(int, input().split()))

fruit_list = [input() for _ in range(m)]
counter = Counter(fruit_list)
counts = sorted(counter.values(), reverse=True)
tags_sorted = sorted(tags)

# 最小总价（最便宜的价格分配给购买次数最多的水果）
min_total=sum(c*p for c,p in zip(counts,tags_sorted))

# 最大总价（最贵的价格分配给购买次数最多的水果）
max_total=sum(c*p for c,p in zip(counts,reversed(tags_sorted)))

print(min_total, max_total)
```

kadene

```
arr = list(map(int, input().split(",")))  # 输入一行字符串，用逗号分割，变成整数列表
n = len(arr)  # 商品数量

# Step 1: 找最大连续子数组和（Kadane算法）
max_ending_here = max_so_far = arr[0]  # 初始化当前最大和和全局最大和为第一个元素
start = end = s = 0  # 用于记录最大子数组的开始和结束位置

for i in range(1, n):
    # 如果当前元素比加上之前的和还大，说明从这里重新开始
    if max_ending_here + arr[i] < arr[i]:
        max_ending_here = arr[i]
        s = i  # 记录当前子数组起点
    else:
        max_ending_here += arr[i]

    # 更新最大和和对应区间
    if max_ending_here > max_so_far:
        max_so_far = max_ending_here
        start = s
        end = i

# Step 2: 在找到的最大连续子数组中尝试去掉一个商品，看能不能得到更大价值
total = sum(arr[start:end+1])  # 最大子数组的总和
remove_one_max = total  # 初始最大价值就是不放回任何商品

for i in range(start, end + 1):
    # 试着去掉第 i 个商品
    # 只有当子数组长度大于1时，才有可能去掉一个商品
    if end - start >= 1:
        remove_one_max = max(remove_one_max, total - arr[i])

# 输出结果，就是最大价值（考虑了放回一个商品的情况）
print(remove_one_max)
```

分割字符串：

```
s=input().split(";")
s=s[:3]
l=[]

for i in range(3):
    if i < 3 and ":=" in s[i]:
        l.append(s[i][-1])
    else:
        l.append("0")

print(l)
```

try-except 方法：

```
s = input().split(";")
s = s[:3]
l = []

for i in range(3):
    try:
        l.append(s[i].split(":=")[1])
    except:
        l.append("0")

print(l)
```

将整数 n 分为 k 份（dfs dp）

```
from functools import lru_cache

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

@lru_cache(maxsize=None)
def count_partitions(n, k, m):
    if n == 0 and k == 0:
        return 1
    if n <= 0 or k <= 0 or m <= 0:
        return 0
    # 选 m 和不选 m 两种情况
    return count_partitions(n - m, k - 1, m) + count_partitions(n, k, m - 1)

# 最大数字不能超过 n
print(count_partitions(n, k, n))
```

股票（inf）：

```
prices = list(map(int, input().split()))
min_price = float('inf')
max_profit = 0
for price in prices:
    if price < min_price:
        min_price = price
    else:
        max_profit=max(max_profit, price - min_price)
print(max_profit)
```

单调栈

```
arr = [2, 1, 2, 4, 3]
stack = []
res = [-1] * len(arr)

for i, v in enumerate(arr):
    while stack and arr[stack[-1]] < v:
        idx = stack.pop()
        res[idx] = v  # 找到右边第一个更大的元素
    stack.append(i)

print(res)  # 输出：[4, 2, 4, -1, -1]
```

合法出栈

```
x = input().strip()

try:
    while True:
        target = input().strip()
        stack = []
        i = 0  # 入栈指针
        j = 0  # 目标序列指针

        while i < len(x):
            stack.append(x[i])
            i += 1

            # 尝试弹栈匹配目标序列
            while stack and j < len(target) and stack[-1] == target[j]:
                stack.pop()
                j += 1

        # 入栈完毕后，如果栈不空，还要继续尝试弹出
        while stack and j < len(target) and stack[-1] == target[j]:
            stack.pop()
            j += 1

        # 判断是否匹配
        if j == len(target) and not stack:
            print("YES")
        else:
            print("NO")

except EOFError:
    pass
```

实现堆结构

```
n = int(input())
heap = []

for _ in range(n):
    ops = input().split()
    t = int(ops[0])
    if t == 1:
        # 插入操作
        u = int(ops[1])
        heapq.heappush(heap, u)
    else:
        # 弹出最小元素
        if heap:
            print(heapq.heappop(heap))
```

河中跳房子 二分（最大化“最短跳跃距离”） 中序转后序表达式 stack 兔子&星空 MST

```python
L, N, M = map(int, input().split())
rocks = [0]  # 起点
for _ in range(N):
    rocks.append(int(input()))
rocks.append(L)  # 终点

# 二分范围
left, right = 1, L
ans = 0

while left <= right:
    mid = (left + right) // 2  # 猜测最短跳跃距离
    removed = 0
    prev = 0  # 记录上一个保留的岩石索引，初始为起点0

    for i in range(1, N+2):  # 遍历所有岩石（含终点）
        if rocks[i] - rocks[prev] < mid:
            # 这块岩石距离上一个保留岩石小于mid，考虑移除它
            removed += 1
            if removed > M:
                break
        else:
            # 保留这块岩石，更新prev
            prev = i

    if removed <= M:
        ans = mid  # mid可行，尝试更大的距离
        left = mid + 1
    else:
        right = mid - 1  # mid不可行，尝试更小的距离

print(ans)
```

```python
def precedence(op):
    if op in ('*', '/'):
        return 2
    elif op in ('+', '-'):
        return 1
    else:
        return 0  # 括号等

def to_postfix(expression):
    import re
    # 正则表达式分词，匹配数字（含小数）和运算符、括号
    tokens = re.findall(r'\d+\.?\d*|[()+\-*/]', expression)
    stack = []
    output = []

    for token in tokens:
        if re.match(r'\d+\.?\d*', token):  # 数字，直接输出
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            # 弹出直到遇见 '('
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # 弹出 '(' 不输出
        else:  # 运算符
            while stack and precedence(stack[-1]) >= precedence(token):
                output.append(stack.pop())
            stack.append(token)

    # 弹出剩余运算符
    while stack:
        output.append(stack.pop())

    return ' '.join(output)

# 主程序
n = int(input())
for _ in range(n):
    expr = input().strip()
    print(to_postfix(expr))
```

输入

第一行只包含一个表示星星个数的数n，n不大于26，并且这n个星星是由大写字母表里的前n个字母表示。接下来的n-1行是由字母表的前n-1个字母开头。最后一个星星表示的字母不用输入。对于每一行，以每个星星表示的字母开头，然后后面跟着一个数字，表示有多少条边可以从这个星星到后面字母表中的星星。如果k是大于0，表示该行后面会表示k条边的k个数据。每条边的数据是由表示连接到另一端星星的字母和该边的权值组成。权值是正整数的并且小于100。该行的所有数据字段分隔单一空白。该星星网络将始终连接所有的星星。该星星网络将永远不会超过75条边。没有任何一个星星会有超过15条的边连接到其他星星（之前或之后的字母）。在下面的示例输入，数据是与上面的图相一致的。

输出

输出是一个整数，表示最小的权值和

样例输入

```
9
A 2 B 12 I 25
B 3 C 10 H 40 I 8
C 2 D 18 G 55
D 1 E 44
E 2 F 60 G 38
F 0
G 1 H 35
H 1 I 35
```

样例输出

```
216
```

根据二叉树前中序序列建树 tree disjointset 宗教信仰

```python
def build_postorder(preorder, inorder):
    if not preorder:
        return ''
    root = preorder[0]
    root_index = inorder.index(root)
    # 左子树的中序和前序
    left_inorder = inorder[:root_index]
    left_preorder = preorder[1:1+len(left_inorder)]
    # 右子树的中序和前序
    right_inorder = inorder[root_index+1:]
    right_preorder = preorder[1+len(left_inorder):]

    left_post = build_postorder(left_preorder, left_inorder)
    right_post = build_postorder(right_preorder, right_inorder)
    return left_post + right_post + root

import sys

lines = sys.stdin.read().strip().split('\n')
for i in range(0, len(lines), 2):
    preorder = lines[i].strip()
    inorder = lines[i+1].strip()
    print(build_postorder(preorder, inorder))
```

```python
def find(parent, x):
    # 查找x的根节点，同时做路径压缩
    if parent[x] != x:
        parent[x] = find(parent, parent[x])  # 路径压缩优化
    return parent[x]

def union(parent, a, b):
    # 合并a和b所属的集合
    rootA = find(parent, a)
    rootB = find(parent, b)
    if rootA != rootB:
        parent[rootB] = rootA  # 合并两个集合

case_num = 1  # 每组输出 Case 编号
while True:
    n, m = map(int, input().split())
    if n == 0 and m == 0:
        break  # 结束条件

    parent = [i for i in range(n + 1)]  # 初始化并查集：每人自成一个集合

    for _ in range(m):
        a, b = map(int, input().split())
        union(parent, a, b)  # 合并a和b属于同一宗教

    # 统计不同的宗教数 = 不同的根节点数
    roots = set()
    for i in range(1, n + 1):
        roots.add(find(parent, i))

    print(f"Case {case_num}: {len(roots)}")
    case_num += 1
```

```python
# 并查集：用于判断是否成环
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))

    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):
        fx, fy = self.find(x), self.find(y)
        if fx == fy:
            return False  # 已在同一个集合中
        self.parent[fx] = fy
        return True

# 主逻辑
n = int(input())
edges = []

for _ in range(n - 1):  # 最后一个点不用输入
    parts = input().split()
    from_star = ord(parts[0]) - ord('A')
    k = int(parts[1])
    for i in range(k):
        to_star = ord(parts[2 + i * 2]) - ord('A')
        weight = int(parts[3 + i * 2])
        edges.append((weight, from_star, to_star))  # 存边：权值，起点，终点

# Kruskal 算法
edges.sort()  # 按权重升序
uf = UnionFind(n)
mst_weight = 0
edge_count = 0

for weight, u, v in edges:
    if uf.union(u, v):
        mst_weight += weight
        edge_count += 1
        if edge_count == n - 1:
            break

print(mst_weight)
```

骑士游历（马走日）回溯 词梯 bfs

```python
import sys
sys.setrecursionlimit(10000)  # 防止递归栈溢出

# 马走日的8个方向
dx = [-2, -2, -1, -1, 1, 1, 2, 2]
dy = [-1, 1, -2, 2, -2, 2, -1, 1]

def knight_tour(n, m, x, y):
    board = [[False for _ in range(m)] for _ in range(n)]  # 标记访问
    total_cells = n * m
    count = [0]  # 使用列表是为了在递归中可以修改它的值

    def dfs(cx, cy, visited):
        if visited == total_cells:
            count[0] += 1  # 找到一种完整路径
            return

        for i in range(8):
            nx, ny = cx + dx[i], cy + dy[i]
            if 0 <= nx < n and 0 <= ny < m and not board[nx][ny]:
                board[nx][ny] = True  # 标记访问
                dfs(nx, ny, visited + 1)
                board[nx][ny] = False  # 回溯

    board[x][y] = True  # 从起点出发
    dfs(x, y, 1)
    return count[0]

# 读取输入
T = int(input())
for _ in range(T):
    n, m, x, y = map(int, input().split())
    result = knight_tour(n, m, x, y)
    print(result)
```

```python
from collections import deque

def can_link(w1, w2):
    """判断两个单词是否仅差一个字符"""
    diff = 0
    for a, b in zip(w1, w2):
        if a != b:
            diff += 1
        if diff > 1:
            return False
    return diff == 1

n = int(input())
words = [input().strip() for _ in range(n)]

word_set = set(words)  # 为快速查找
word_index = {word: i for i, word in enumerate(words)}  # word -> index

# 构建图
graph = [[] for _ in range(n)]
for i in range(n):
    for j in range(i + 1, n):
        if can_link(words[i], words[j]):
            graph[i].append(j)
            graph[j].append(i)

# 读取起点和终点
start_word, end_word = input().strip().split()

if start_word not in word_index or end_word not in word_index:
    print("NO")
else:
    start = word_index[start_word]
    end = word_index[end_word]
```

```python
# BFS
queue = deque()
visited = [False] * n
prev = [-1] * n  # 用于还原路径

queue.append(start)
visited[start] = True

found = False
while queue:
    curr = queue.popleft()
    if curr == end:
        found = True
        break
    for neighbor in graph[curr]:
        if not visited[neighbor]:
            visited[neighbor] = True
            prev[neighbor] = curr
            queue.append(neighbor)

if not found:
    print("NO")
else:
    # 还原路径
    path = []
    curr = end
    while curr != -1:
        path.append(words[curr])
        curr = prev[curr]
    path.reverse()
    print(" ".join(path))
```

归并排序:（冒泡（？逆序对）　　海军（拓扑）

```python
import sys
sys.setrecursionlimit(1000000)

def merge_sort(arr):
    def sort_and_count(left, right):
        if right - left <= 1:
            return 0
        mid = (left + right) // 2
        count = sort_and_count(left, mid) + sort_and_count(mid, right)
        i, j = left, mid
        tmp = []
        while i < mid and j < right:
            if arr[i] <= arr[j]:
                tmp.append(arr[i])
                i += 1
            else:
                tmp.append(arr[j])
                count += mid - i  # 左边剩下的都是比它大的
                j += 1
        tmp.extend(arr[i:mid])
        tmp.extend(arr[j:right])
        arr[left:right] = tmp
        return count

    return sort_and_count(0, len(arr))

# 主循环处理输入
while True:
    line = sys.stdin.readline()
    if not line:
        break
    n = int(line.strip())
    if n == 0:
        break
    arr = []
    for _ in range(n):
        arr.append(int(sys.stdin.readline()))
    print(merge_sort(arr))
```

```python
from collections import deque
import sys
input = sys.stdin.readline

T = int(input())
for _ in range(T):
    N, M = map(int, input().split())
    graph = [[] for _ in range(N + 1)]
    indegree = [0] * (N + 1)

    for _ in range(M):
        u, v = map(int, input().split())
        graph[u].append(v)
        indegree[v] += 1

    queue = deque([i for i in range(1, N + 1) if indegree[i] == 0])
    count = 0

    while queue:
        node = queue.popleft()
        count += 1
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    if count == N:
        print("No")  # 无环
    else:
        print("Yes")  # 有环
```

走山路：最小体力 Dijkstra

```python
import heapq
import sys
input = sys.stdin.readline

# 读取输入
m, n, p = map(int, input().split())
grid = []

for _ in range(m):
    row = input().split()
    grid.append(row)

queries = []
for _ in range(p):
    sx, sy, ex, ey = map(int, input().split())
    queries.append((sx, sy, ex, ey))

# 四个方向
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def in_bounds(x, y):
    return 0 <= x < m and 0 <= y < n
```

```python
def dijkstra(sx, sy, ex, ey):
    if grid[sx][sy] == "#" or grid[ex][ey] == "#":
        return "NO"

    dist = [[float('inf')] * n for _ in range(m)]
    dist[sx][sy] = 0
    heap = [(0, sx, sy)]

    while heap:
        cost, x, y = heapq.heappop(heap)
        if (x, y) == (ex, ey):
            return cost
        for dx, dy in dirs:
            nx, ny = x + dx, y + dy
            if in_bounds(nx, ny) and grid[nx][ny] != "#":
                height_diff = abs(int(grid[x][y]) - int(grid[nx][ny]))
                new_cost = cost + height_diff
                if new_cost < dist[nx][ny]:
                    dist[nx][ny] = new_cost
                    heapq.heappush(heap, (new_cost, nx, ny))

    return "NO"

# 执行每个查询
for sx, sy, ex, ey in queries:
    print(dijkstra(sx, sy, ex, ey))
```

Trie 电话号码

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

def is_consistent(phone_numbers):
    root = TrieNode()
    for number in sorted(phone_numbers):
        node = root
        for digit in number:
            if node.is_end:
                return False  # 之前已有一个完整号码是当前号码的前缀
            if digit not in node.children:
                node.children[digit] = TrieNode()
            node = node.children[digit]
        if node.children:
            return False  # 当前号码是其他号码的前缀
        node.is_end = True
    return True

t = int(input())
for _ in range(t):
    n = int(input())
    numbers = [input().strip().replace(" ", "") for _ in range(n)]
    print("YES" if is_consistent(numbers) else "NO")
```

Compile Error：

1.这个多半是变量名字打错了，或者多打;：之类的，这个好查，一般本地都运行不了。

2.OJ的pylint是静态检查，有时候报的compile error不对。解决方法有两种，如下：

1）第一行加# pylint: skip-file

2）方法二：如果函数内使用全局变量（变量类型是immutable，如int），则需要在程序最开始声明一下。如果是全局变量是list类型，则不受影响。

Runtime Error：

1.指针越界，比如长度为5（index为0，1，2，3，4）的数组你去获取list[5]，但是注意list[-5]是合法的（index可以为-1，-2，-3，-4，-5）。

2.数组开太大了，比如开到1000000000（9个0）就会

3.递归爆栈：这个用以下代码解决

```
from sys import setrecursionlimit
setrecursionlimit(10000)#python 默认 200
```

4.输入读取错误，一般是输入没读完就exit（）　（所以要谨慎使用这个函数）

5.除以0，检查一下变量是不是可以是0

Wrong Answer

这个需要好好审一遍代码

如果没有逻辑性的错误，就查一下边界值，比如0啊1啊什么的，很可能在这些地方出错

如果还不行，就仔细审题，看看哪个地方理解错了

print(f'{ans},{res:.1f}')print是可以带sep和end参数的

可以用round进行四舍六入五成双的操作

枚举：　for i,x in enumerate(list),遍历list中的（下标，值）对

集合：
并:'|'　交:'&'　差：'-'

Presentation Error

只遇到过一次，就是该输出空行的时候没输出空行

Time Limit Exceeded

1.死循环（这种在BFS类里容易碰到，要注意visited的逻辑是否正确）

2.算法问题（注意每次读题时注意数据量，以便快速确定算法）

3.卡常数，这种情况用pypy3就可以了，或者再优化一下代码，去除一些鸡肋的O（n）操作

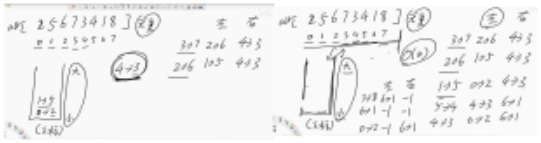
Memory Limit Exceeded

数组开太大了，一般DP难题会遇到，这时候就要考虑压缩空间了。

还有就是BFS的时候queue可能会超，要考虑进堆的数据的优化

单调栈：（来自柱状图最大矩形，具体题目需要变形）

求左右最近的小于自身的数：



有重复数字也是一样的操作（等于也弹出），但是最后要进行一遍右答案的修正（因为有可能记录的是相等的值）（从右往左修正）

```
#求左右两边严格小于自身的最近的数 并且有重复值 的模板
#遍历
for i in range(n):
    while st and arr[st[-1]]>=arr[i]:#不严格不用清算
        cur=st.pop()
        ans[cur][0]=st[-1] if st else -1
        ans[cur][1]=i
    st.append(i)
#清算
while st:
    cur=st.pop()
    ans[cur][0]=st[-1] if st else -1
    ans[cur][1]=-1
#修正
#n-1一定是-1，所以不需要修正
for i in range(n-2,-1,-1):
    if ans[i][1]!=-1 and arr[ans[i][1]]==arr[i]:
        ans[i][1]=ans[ans[i][1]][1]
```

重复一定要特判，子数组一题重复的就要作为ans才可以不重不漏。有些时候中间的相等值答案可能不对，只要后续的相等值进来能把答案修正对就可以了（回忆最大矩形一题，相等也弹出）

妙题：01矩阵中面积最大的长方形：枚举每一行，以每一行作为底去进行单调栈即可（不连续就变成0，还要记得复用上一行的数据）

其他用法：维持答案的一种可能性，比如求数组中的坡，维持栈中是递减的，遇到大的弹出，然后再从右往左更新答案。

比如字典序最小的规定字符的字符串，先用counter记录能不能删某个字符，再用单调栈去维护字典序最小