Dijkstra模板

题目：

某同学在一处山地里，地面起伏很大，他想从一个地方走到另一个地方，并且希望能尽量走平路。
现有一个m*n的地形图，图上是数字代表该位置的高度，"#"代表该位置不可以经过。
该同学每一次只能向上下左右移动，每次移动消耗的体力为移动前后该同学所处高度的差的绝对值。现在给出该同学出发的地点和目的地，需要你求出他最少要消耗多少体力。

```
import heapq
directions=[(1,0),(-1,0),(0,1),(0,-1)]
def dijkstra(start_x,start_y,end_x,end_y):
    pos=[]
    dis=[[float("inf")]*n for _ in range(m)]
    dis[start_x][start_y]=0
    heapq.heappush(pos,(0,start_x,start_y))
    while pos:
        d,x,y=heapq.heappop(pos)
        if x==end_x and y==end_y:
            return d
        h=int(info[x][y])
        for dx,dy in directions:
            nx,ny=x+dx,y+dy
            if 0<=nx<m and 0<=ny<n and info[nx][ny]!="#":
                if dis[nx][ny]>d+abs(int(info[nx][ny])-h):
                    dis[nx][ny]=d+abs(int(info[nx][ny])-h)
                    heapq.heappush(pos,(dis[nx][ny],nx,ny))
```

Dilworth's Theorem

最小完整覆盖全数组的上升子序列数等于最长非上升子序列的长度。

二分查找模板

bisect_left:

```
lo,hi=0,len(a)
while lo<hi:
    mid=(lO+hi)//2
    if a[mid]<x:
        lo=mid+1
    else:
        hi=mid
```

bisect_right:

```
lo,hi=0,len(a)
while lo<hi:
    mid=(lo+hi)//2
    if x<a[mid]:
        hi=mid
    else:
        lo=mid+1
```

保留小数位数：

```
number=3.1415926
result="{:.2f}".format(number) #保留2位
print(result)
```

哈希表

求一个数组最长的和为0的连续子序列的方法（和的平均值为a）

```
def solve(arr,a):
    diff=[x-a for x in arr] #转化为最长和为0
    prefix_sum=0
    prefix_map={0:-1}
    max_len=0
    start_index=-1
    for i in range(len(diff)):
        prefix_sum+=diff[i]
        #如果前缀和已经出现过，计算子数组长度
        if prefix_sum in prefix_map:
            length=i-prefix_map[prefix_sum]
            if length>max_len:
                max_len=length
                start_index=prefix_map[prefix_sum]+1
        else:
            prefix_map[prefix_sum]=i
    if max_len>0:
        return arr[start_index:start_index+max_len]
    else:
        return []
```

利用二分查找求最长下降子序列

```
import bisect
k=int(input())
a=list(map(int,input().split()))
sub=[]
for i in range(k):
    pos=bisect.bisect_right(sub,a[i]) #sub中第一个大于a[i]的元素的索引
    if pos<len(sub):
        sub[pos]=a[i]
    else:
        sub.append(a[i])
print(len(sub))
```

`map(function,iterables)`

```
squared = list(map(lambda x: x**2, [1, 2, 3, 4]))  # [1, 4, 9, 16]
```

debug:RE：数组越界/除0，TLE/MLE：程序错误（未设置递归边界）

随机数

```
import random
x=random.randint(a,b) #a到b之间的随机整数
x=random.random() #0~1的随机浮点数
x=random.uniform(a,b) #a~b之间的随机浮点数
x=random.choice(list) #在list列表中随机选择
```

将列表中a~b的部分反转：

```
list[a:b+1]=reversed(list[a:b+1])
```

dp问题

红蓝玫瑰

```
roses=list(input())
n=len(roses)
r=[0]*n #r[i]表示将前i个都变红的最小次数
b=[0]*n
if roses[0]=="R":
    r[0]=0
    b[0]=1
else:
    r[0]=1
    b[0]=0
for i in range(1,n):
    if roses[i]=="R":
        r[i]=r[i-1]
        b[i]=min(b[i-1],r[i-1])+1
    else:
```

```
        r[i]=min(r[i-1],b[i-1])+1
        b[i]=b[i-1]
print(r[-1])
```

0-1背包

```
n=len(weights)
dp=[[0]*(w+1) for _ in range(n+1)]
for i in range(1,n+1):
    for j in range(1,w+1):
        if j>=weights[i-1]:
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-weights[i-1]]+weights[i-1])
        else:
            dp[i][j]=dp[i-1][j]
print(dp[n][w])
```

一维优化：

```
def knapsack_1d(weights, values, W):  # 一维视为二维的滚动数组实现
    n = len(weights)
    dp = [0] * (W + 1)  # 初始化 dp 数组，容量从 0 到 W
    for i in range(n):  # 遍历每件物品
        for j in range(W, weights[i] - 1, -1):  # 倒序遍历背包容量(保证每件物品只能选一次)
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[W]
```

完全背包

```
t,m=map(int,input().split())
time=[]
value=[]
for _ in range(m):
    a,b=map(int,input().split())
    time.append(a)
    value.append(b)
dp=[[0]*(t+1) for _ in range(m+1)]
for i in range(1,m+1):
    for j in range(1,t+1):
        dp[i][j]=dp[i-1][j]
        if j>=time[i-1]:
            dp[i][j]=max(dp[i][j],dp[i][j-time[i-1]]+value[i-1])
print(dp[m][t])
```

每种物品可以无限次选取

一维优化：

```python
def knapsack_complete(weights, values, capacity):
    dp = [0] * (capacity + 1)  # dp[j]为当背包容量为j时,背包所能容纳的最大价值
    dp[0] = 0
    for i in range(len(weights)):  # 遍历所有物品
        for j in range(weights[i], capacity + 1):  # 从当前物品的重量开始，计算每个容量的最大价值
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[capacity]
```

必须装满

```python
def knapsack_complete_fill(weights, values, capacity):
    dp = [-float('inf')] * (capacity + 1)  # 初始值为负无穷，表示不能达到该容量
    dp[0] = 0  # 容量为 0 时，价值为 0
    for i in range(len(weights)):  # 遍历所有物品
        for w in range(weights[i], capacity + 1):  # 遍历所有容量，从 weights[i] 开始
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    # 如果 dp[capacity] 仍为 -inf，说明无法填满背包
    return dp[capacity] if dp[capacity] != -float('inf') else 0
```

多重背包

```python
def binary_optimized_multi_knapsack(weights, values, quantities, capacity):
    # 使用二进制优化解决多重背包问题
    n = len(weights)
    items = []
    # 将每种物品根据数量拆分成若干子物品（使用二进制优化）
    for i in range(n):
        w, v, q = weights[i], values[i], quantities[i]
        k = 1
        while k < q:
            items.append((k * w, k * v))  # 添加子物品(weight, value)
            q -= k
            k << 1  # 位运算,相当于k *= 2,按二进制拆分,物品时间复杂度由q变为log(q)
        if q > 0:
            items.append((q * w, q * v))  # 添加剩余部分,如果有的话
    # 动态规划求解0-1背包问题
    dp = [0] * (capacity + 1)
    for w, v in items:  # 遍历所有子物品
        for j in range(capacity, w - 1, -1):  # 01背包的倒序遍历
            dp[j] = max(dp[j], dp[j - w] + v)
    return dp[capacity]
```

双重dp

```python
value = list(map(int, input().split(",")))
dp_keep = value[0]      # 不放回
dp_remove = value[0]    # 放回一件商品
ans = value[0]  # 答案记录
for i in range(1, len(value)):  # 对结尾为第i件商品的选法
    previous_dp_keep = dp_keep  # 保存结尾尾i-1时的选法
    dp_keep = max(dp_keep + value[i], value[i])  # 维护dp_keep
    dp_remove = max(previous_dp_keep, dp_remove + value[i])  # 判断是否放回第i个商品更划算
    ans = max(ans, dp_keep, dp_remove)
print(ans)
```

区间dp

```python
n = int(input())  # 石子的堆数
stones = list(map(int, input().split()))
sum_ = [0] * (n + 1)  # 前缀和，用于快速计算区间和
for i in range(1, n + 1):
    sum_[i] = sum_[i - 1] + stones[i - 1]
dp = [[float('inf')] * n for _ in range(n)]  # dp 数组，初始化为正无穷
for i in range(n):
    dp[i][i] = 0  # 单堆的代价为 0
for L in range(2, n + 1):  # 枚举区间长度 L，从 2 到 n
    for i in range(n - L + 1):  # 起始位置从0到n-L
        j = i + L - 1  # 长度为L后的终点
        for k in range(i, j):  # 枚举分割点 k
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + sum_[j + 1] - sum_[i])
print(dp[0][n - 1])
```

defultdict:会为缺失的键提供一个默认值(`int->0,list->[],set->set(),str->""`),而不是抛出`KeyError`

```python
from collections import defaultdict
dictionary = defaultdict(list)  # 自定义默认值defaultdict(lambda: XXX)
```

(index, item)生成器enumerate `enumerate(list,tuple,string)`,输出一些tuple.欲对`item`排列,需转化为新列表:

```python
indexed_list1 = list(enumerate(list1))
indexed_list1.sort(key=lambda x: x[1], reverse = True)  # False升序,True降序;也可直接-x[1]
```

二分插入内置函数bisect `import bisect`,重要的两个方法如下: `bisect_left(a, x, lo=0, hi=len(a))` 返回 `x` 在升序列表 `a` 中的插入位置(左起寻找第一个满足条件的位置,right相反)

```python
print(bisect.bisect_left([1, 3, 4, 7, 9], 5))  # >>>3
```

Math

```python
import math
pow(a,b) #a^b
bin(12) #0b1100 从第三位开始为二进制表示
math.ceil(2.3) #3，向上取整
math.floort(2.3) #2，向下取整
gcd(a,b) #a，b的最大公约数
math.log(a,b) #log_a^b
```

排序

归并排序

稳定排序

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

快速排序

```python
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)
```

deque

```
from collections import deque
deque.popleft(item)#左弹出
deque.pop(item)#右弹出
deque.appendleft(item)#左加入
deque.append(item)#右加入
```

欧拉筛

```
def primes(n):
    is_prime=[True]*(n+1)
    primes=[]
    for i in range(2,n+1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if p*i>n:
                break
            is_prime[p*i]=False
            if i%p==0:
                break
    return primes
```

如果想判断一个数是不是质数就把primes改为set进行查找。

下一个排列

```
def NP(nums):
    for i in range(len(nums)-2,-1,-1):
        if nums[i]<nums[i+1]:
            for j in range(len(nums)-1,i,-1):
                if nums[j]>nums[i]:
                    nums[j],nums[i] = nums[i],nums[j]
                    tmp=nums[len(nums)-1:i:-1]
                    nums[i+1:]=tmp
                    return nums
    else:
        nums.reverse()
        return nums
print(NP([4,2,6,3]))
```

得到所有排列

```python
def permute(nums):
    if len(nums) == 1: return [nums]  # 递归终止条件：只有一个元素时返回自身
    permutations = []  # 存储答案
    for i in range(len(nums)):
        current = nums[i]  # 当前元素
        remaining = nums[:i] + nums[i+1:]  # 剩余元素
        for p in permute(remaining):  # 递归生成剩余元素的排列，并加上当前元素
            permutations.append([current] + p)
    return permutations
```

全排列

```python
import itertools
n=int(input())
lst=list(range(1,n+1))
permutations=itertools.permutations(lst)
for i in permutations:
    print(" ".join(map(str,i)))
```

最长上升子序列

```python
import bisect
def lis(a):
    dp=[float('inf')]*(len(a)+2)
    for i in range(len(a)):
        dp[bisect.bisect_left(dp,a[i])]=a[i]
    print(dp)
    return bisect.bisect_left(dp,float('inf'))
```

连续子序列和最大

```python
def kadane(v):#卡丹算法求最大子序列
    max_cur=0
    max_all=0
    for i in range(len(v)):
        max_cur=max(v[i],max_cur+v[i])
        max_all=max(max_all,max_cur)
    return max_all
```

设置递归深度：

```python
import sys
sys.setrecursionlimit(1<<30)
```

接雨水（双指针）

```
```

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = left = pre_max = suf_max = 0  # 初始化结果、左指针和两个最大高度为0
        right = len(height) - 1  # 初始化右指针为数组末尾
        while left < right:  # 当左指针小于右指针时循环
            pre_max = max(pre_max, height[left])  # 更新左指针位置的最大高度
            suf_max = max(suf_max, height[right])  # 更新右指针位置的最大高度
            if pre_max < suf_max:  # 如果左指针位置的最大高度小于右指针位置的最大高度
                ans += pre_max - height[left]  # 计算并累加左指针位置能够接住的雨水量
                left += 1  # 移动左指针
            else:  # 否则
                ans += suf_max - height[right]  # 计算并累加右指针位置能够接住的雨水量
                right -= 1  # 移动右指针
        return ans  # 返回最终结果
```

搜索

求最长回文子串

```python
def manacher(s):
    # 1.预处理字符串
    t = '^#' + '#'.join(s) + '#$'  # 字符间插入#,从而对于偶数子串也可以中心扩展
    n = len(t)  # 得到新字符串长度
    P = [0] * n  # P[i]表示以t[i]为中心的回文半径
    C, R = 0, 0  # C为当前回文中心,R为当前回文的右边界
    # 2.计算回文半径
    for i in range(1, n - 1):  # i位置为中心
        # 如果 i 在 R 范围内，用对称位置的回文半径初始化 P[i]
        P[i] = min(R - i, P[2 * C - i]) if i < R else 0
        # 中心扩展，尝试扩展回文半径
        while t[i + P[i] + 1] == t[i - P[i] - 1]:
            P[i] += 1
        # 更新回文的中心和右边界
        if i + P[i] > R:
            C, R = i, i + P[i]
    # 3.找到最长回文
    max_len = max(P)  # 最长回文半径
    center_index = P.index(max_len)  # 最长回文对应的中心索引
    # 原始字符串中的起始索引
    start = (center_index - max_len) // 2
    return s[start:start + max_len]
```

滑雪

```python
from functools import lru_cache
def solve(r,c,heights):
    directions=[(1,0),(-1,0),(0,1),(0,-1)]
    dp=[[-1 for _ in range(c)] for _ in range(r)]
    @lru_cache
    def dfs(x,y):
```

```
        if dp[x][y]!=-1:
            return dp[x][y]
        max_length=1
        for dx,dy in directions:
            nx,ny=x+dx,y+dy
            if 0<=nx<r and 0<=ny<c and heights[nx][ny]<heights[x][y]:
                max_length=max(max_length,dfs(nx,ny)+1)
        dp[x][y]=max_length
        return max_length
    ans=0
    for i in range(r):
        for j in range(c):
            ans=max(ans,dfs(i,j))
    return ans
r,c=map(int,input().split())
heights=[list(map(int,input().split())) for _ in range(r)]
print(solve(r,c,heights))
```

bfs

```
from collections import deque
n=int(input())
matrix=[list(map(int,input().split())) for _ in range(n)]
directions=[(1,0),(-1,0),(0,1),(0,-1)]
for i in range(n):
    if 5 in matrix[i]:
        x1,y1=i,matrix[i].index(5)
        break
for dx,dy in directions:
    x=x1+dx
    y=y1+dy
    if 0<=x<n and 0<=y<n and matrix[x][y]==5:
        x2,y2=x,y
        break
def bfs(x1,y1,x2,y2):
    queue=deque()
    queue.append((x1,y1,x2,y2))
    visited=set()
    visited.add((x1,y1,x2,y2))
    while queue:
        x11,y11,x22,y22=queue.popleft()
        if (matrix[x11][y11]==9 or matrix[x22][y22]==9) and 0<=x11<n and
0<=x22<n and 0<=y11<n and 0<=y22<n and matrix[x1][y1]!=1 and matrix[x2]
[y2]!=1:
            return "yes"
        for dx,dy in directions:
            nx1,ny1=x11+dx,y11+dy
            nx2,ny2=x22+dx,y22+dy
            if 0<=nx1<n and 0<=ny1<n and 0<=nx2<n and 0<=ny2<n and matrix[nx1]
[ny1]!=1 and matrix[nx2][ny2]!=1:
                state=(nx1,ny1,nx2,ny2)
                if state not in visited:
                    visited.add(state)
                    queue.append(state)
```

```
    return "no"
print(bfs(x1,y1,x2,y2))
```

体育游戏跳房子

```
from collections import deque
def bfs(n,m):
    queue=deque()
    queue.append((0,n,""))
    visited=set()
    visited.add(n)
    while queue:
        step,position,path=queue.popleft()
        if position==m:
            return step,path
        if position*3 not in visited:
            queue.append((step+1,position*3,path+"H"))
            visited.add(position*3)
        if position//2 not in visited:
            queue.append((step+1,position//2,path+"O"))
            visited.add(position//2)
while True:
    n,m=map(int,input().split())
    if n==0 and m==0:
        break
    step,path=bfs(n,m)
    print(step)
    print(path)
```

变换的迷宫

```
from collections import deque
def bfs(matrix,r,c,k):
    directions=[(1,0),(-1,0),(0,1),(0,-1)]
    for i in range(r):
        if "S" in matrix[i]:
            start=(i,matrix[i].index("S"))
            break
    queue=deque([(0,start[0],start[1])])
    visited=set()
    visited.add((0,start[0],start[1]))
    while queue:
        time,x,y=queue.popleft()
        for dx,dy in directions:
            nx,ny=x+dx,y+dy
            tmp=(time+1)%k
            if 0<=nx<r and 0<=ny<c:
                cell=matrix[nx][ny]
                if cell=="E":
                    return time+1
                elif cell!="#" or tmp==0:
                    if (tmp,nx,ny) not in visited:
                        queue.append((time+1,nx,ny))
                        visited.add((tmp,nx,ny))
```

```
    return "Oop!"
t = int(input())
for _ in range(t):
    r, c, k = map(int, input().split())
    matrix = [input().strip() for _ in range(r)]
    print(bfs(matrix, r, c, k))
```

dfs

马走日

```
def solve(n,m,x,y):
    directions=[(1,2),(2,1),(2,-1),(1,-2),(-1,-2),(-2,-1),(-2,1),(-1,2)]
    visited=[[False]*m for _ in range(n)]
    cnt=0

    def dfs(cx,cy,count):
        nonlocal cnt
        if count==n*m:
            cnt+=1
            return

        for dx,dy in directions:
            nx,ny=cx+dx,cy+dy
            if 0<=nx<n and 0<=ny<m and not visited[nx][ny]:
                visited[nx][ny]=True
                dfs(nx,ny,count+1)
                visited[nx][ny]=False

    visited[x][y]=True
    dfs(x,y,1)
    return cnt

t=int(input())
for _ in range(t):
    n,m,x,y=map(int,input().split())
    print(solve(n,m,x,y))
```

积木

```
def judge(word,blocks,used):
    if word=="":
        return True
    for i in range(4):
        if word[0] in blocks[i] and not used[i]:
            used[i]=True
            if judge(word[1:],blocks,used):
                return True
            used[i]=False
    return False
n=int(input())
blocks=[]
```

```
for _ in range(4):
    s=input()
    blocks.append(s)
for _ in range(n):
    word=input()
    used=[False]*4
    print("YES") if judge(word,blocks,used) else print("NO")
```

八皇后

```
def solve(n):
    if n==13:
        return [[1,3,5,2,9,12,10,13,4,6,8,11,7],
                [1,3,5,7,9,11,13,2,4,6,8,10,12],
                [1,3,5,7,12,10,13,6,4,2,8,11,9]],73712
    solutions=[]
    cnt=0
    solution=[0]*n
    col1=[False]*n
    diag1=[False]*(2*n-1)
    diag2=[False]*(2*n-1)
    def backtrack(row):
        nonlocal cnt
        if row==n:
            cnt+=1
            if len(solutions)<3:
                solutions.append(solution[:])
        for col in range(n):
            if col1[col] or diag1[row-col+n-1] or diag2[row+col]:
                continue
            solution[row]=col+1
            col1[col]=True
            diag1[row-col+n-1]=True
            diag2[row+col]=True
            backtrack(row+1)
            col1[col]=False
            diag1[row-col+n-1]=False
            diag2[row+col]=False
    backtrack(0)
    return solutions,cnt
n=int(input())
solutions,cnt=solve(n)
for solution in solutions:
    print(" ".join(map(str,solution)))
print(cnt)
```

连通域染色

```
directions=((0,1),(0,-1),(-1,0),(1,0),(1,-1),(1,1),(-1,-1),(-1,1))
def dfs(x,y):
    global n,m,color,board
    board[x][y]=color
    area=1
    for d in directions:
        nx=x+d[0]
        ny=y+d[1]
        if and((nx<0,ny<0,nx>=n,ny>=m)):
            continue
        if board[nx][ny]=="W":
            area+=dfs(nx,ny)
    return area
```

迷宫问题

```
directions = [...]
...
visited = [[False]*n for _ in range(m)]
def dfs(x,y):
    global area
    if vis[x][y]:
        return
    visited[x][y] = True
    ...
    for dx,dy in directions:
        nx,ny=x+dx,y+dy
        if 0<=nx<m and 0<=ny<n and vis[nx][ny] and ...:
            dfs(nx,ny)
#此处还可以在dfs前不用标记vis，在for循环里：
#vis[nx][ny]=1
#dfs(nx,ny)
#vis[nx][ny]=0
#这样自身形成回溯
for i in range(m):
    for j in range(n):
        if not visited[i][j]:
            dfs(i,j)
```

两座孤岛最短距离

```
from collections import deque
def solve(grid):
    m=len(grid)
    n=len(grid[0])
    queue=deque()
    def dfs(i,j):
        if i<0 or i>=m or j<0 or j>=n or grid[i][j]!=1:
            return
        grid[i][j]=2
        queue.append((i,j))
        dfs(i-1,j)
        dfs(i+1,j)
        dfs(i,j-1)
```

```
        dfs(i,j+1)
    found=False
    for i in range(m):
        if found:
            break
        for j in range(n):
            if grid[i][j]==1:
                dfs(i,j)
                found=True
                break
    distance=0
    directions=[(1,0),(-1,0),(0,1),(0,-1)]
    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<m and 0<=ny<n:
                    if grid[nx][ny]==2:
                        continue
                    if grid[nx][ny]==1:
                        return distance
                    grid[nx][ny]=2
                    queue.append((nx,ny))
        distance+=1
n=int(input())
grid=[]
for _ in range(n):
    row=list(map(int,input()))
    grid.append(row)
print(solve(grid))
```

质因数分解

```
def solve(x):
    nums=[]
    while x%2==0:
        nums.append(2)
        x//=2
    for i in range(3,isqrt(x)+1,2):
        if is_prime(i):
            while x%i==0:
                nums.append(i)
                x//=i
    if x>1:
        nums.append(x)
```