25fall CS101 CP by 2500012264 edition 20251224 final

ASCII 码：大写字母 A 65 到 Z90，小写字母 a 97 到 z 122，数字 0 48 到 9 57。ord() chr()

列表：list=[]

del list[2] 删掉某一项

list.append(obj) 在列表末尾添加新的对象 list.count(obj) 统计某个元素在列表中出现的次数

list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

list.index(obj) 从列表中找出某个值第一个匹配项的索引位置 O(1)

list.insert(index, obj) 将对象插入列表

list.pop([index=-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值 O(1)

list.remove(obj) 移除列表中某个值的第一个匹配项 list.reverse() 反向列表中元素

list.sort(key=None, reverse=False) 对原列表进行排序

list.clear() 清空列表 list.copy() 复制列表

生成排序过的序列 listed_elderly=sorted(elderly,key=lambda x:(x[0],x[1]),reverse=True)默认小到大

字符串：

从后面索引:	-6	-5	-4	-3	-2	-1	切片[a:b]是左闭右开区间，列表也有切片
从前面索引:	0	1	2	3	4	5	
	R	u	n	o	o	b	
从前面截取:	:	1	2	3	4	5	:
从后面截取:	:	-5	-4	-3	-2	-1	:

isalnum() 检查字符串是否由字母和数字组成，即字符串中的所有字符都是字母或数字。如果字符串至少有一个字符，并且所有字符都是字母或数字，则返回 True；否则返回 False。

isalpha() 如果字符串至少有一个字符并且所有字符都是字母或中文字则返回 True, 否则返回 False

isdigit() 如果字符串只包含数字则返回 True 否则返回 False.

islower() 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False

replace(old, new [, max]) 将字符串中的 old 替换成 new,如果 max 指定，则替换不超过 max 次

reversed_s = s[::-1] 字符串倒序

count() 计数

string.lower() ; string.upper() ; string.swapcase() # 转大写；转小写；大转小小转大

字符串是不可变对象，可以索引、切片、连接（'+'），不能修改，修改要创一个列表

string.find() # 查找指定字符，有就返回第一个找到的 index，没有会返回-1

string.zfill() # 自动在前面补 0 补到所需位数

str.strip() ; string.rstrip() ; string.lstrip() #去掉空格；去掉头部/尾部的空格

''.join(list) # 注意 list 里面要是 str 类型。或者 str(x) for x in list

print(*ans) ; print(a, b, end="\n") # 换行

字典：dict={}

键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行

dict.clear() 删除字典内所有元素 dict.copy() 返回一个字典的浅复制

dict.fromkeys() 创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值

dict.get(key, b) 返回指定键的值，如果键不在字典中返回 b 的值

key in dict 如果键在字典 dict 里返回 true，否则返回 false

dict.setdefault(key, default=None) 和 get()类似, 但如果键不存在于字典中，将会添加键并将值设为 default

dict.update(dict2) 把字典 dict2 的键/值对更新到 dict 里

dict.pop(key[,default]) 删除字典 key（键）所对应的值，返回被删除的值。

dict.popitem() 返回并删除字典中的最后一对键和值。

for k,v in dict.items(): 用于遍历整个字典，返回的每一项都是元组（键，值）

集合：set=set()　或者 set={1,2,3}
add()　为集合添加元素　clear()　移除集合中的所有元素　copy()　拷贝一个集合
difference()　返回多个集合的差集
discard() 删除集合中指定的元素，没有不报错　remove()　移除指定元素，没有会报错
intersection() 返回集合的交集　union()　返回两个集合的并集
isdisjoint()　判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
sa.issubset(sb)判断 sa 是否为 sb 的子集
sa.issuperset(sb)　判断 sb 是否为 sa 的子集
symmetric_difference() 返回两个集合中不重复的元素集合
update() 给集合添加元素　len() 计算集合元素个数　pop() 随机移除元素

栈：stack 在一端取和放 stack = []列表自带属性
run in O (1) time:
empty (): checks if the stack is empty
size (): returns the number of elements in the stack
top () / peek (): shows the top element without removing it
push(a): adds an element a at the top
pop (): removes the top element

队列：queue 可用双端序列 deque 实现，两边操作 O(1)
from collections import deque
q=deque([])
q.popleft()　q.pop()时间复杂度均为 O(1)
q.appendleft()　q.append()
q = deque([0] * 5, maxlen=5)　# deque([0, 0, 0, 0, 0], maxlen=5)
maxlen：可选参数，指定 deque 的最大长度。当元素数超过此值时，旧元素会被自动移除（从另一端）

堆：heapq 完全二叉树结构 小顶堆：堆顶元素是整个堆中最小的元素（heapq 默认实现）
高效操作：插入元素（heappush）和删除堆顶元素（heappop）的时间复杂度均为 O (log n)
import heapq
heap = []
heappush(heap, item)　向堆中插入元素 item，并维持堆的特性（小顶堆）
heappop(heap)　删除并返回堆顶元素（最小元素），自动调整堆结构以维持特性。空堆 IndexError
heap[0]　访问堆顶元素（最小元素），但不删除（注意：直接修改堆顶元素会破坏堆结构）
heapreplace(heap, item)　先弹出堆顶元素，再插入 item（适合替换堆顶元素的场景）

heapq 默认只支持小顶堆，若需要大顶堆（堆顶为最大元素），可以通过插入元素的相反数实现
Top K 问题：从大量数据中快速找到最大（或最小）的 K 个元素（用堆效率高于排序）。
```
def find_top_k(nums, k):
    # 用小顶堆存储最大的 k 个元素（堆顶为 k 个中最小的，便于替换）
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)  # 超过 k 个则弹出最小的
    return heap  # 堆中剩余的就是最大的 k 个元素
```

```
nums = [3, 1, 4, 1, 5, 9, 2, 6]
print(find_top_k(nums, 3))  # 输出：[5,6,9]（堆结构，实际最大的 3 个）
```

列表推导式 [表达式 for 变量 in 列表（ if 条件）]

```
new_names = [name.upper()for name in names if len(name)>3]
```

字典推导式 { 键: 值 for 值 in 列表 if 条件 }

```
listdemo = ['Google','Runoob', 'Taobao']
>>> newdict = {key:len(key) for key in listdemo}
>>> newdict ={'Google': 6, 'Runoob': 6, 'Taobao': 6}
```

字符串格式化与浮点数精度

num = 3.14159　　formatted_num = "%.2f" % num　　formatted_num："3.14"

F-String 方法

```
num = 3.14159
formatted_num = f"{num:.2f}"
print(formatted_num)  # 输出："3.14"
```

```
>>> name = 'Runoob'
>>> f'Hello {name}'
```

格式化字符串、变量替换

```
print("The minimum amount of money in the piggy-bank is {}.".format(least))
```

不定行输入

1、import sys

sys.stdin.read　返回一个字符串，包含所有输入的内容

sys.stdin.readlines　返回一个列表，每一行是一个字符串元素，保留每行末尾的换行符需要 strip

2、try-except

```
lines = []
try:
    while True:
        line = input()
        lines.append(line)
except EOFError:    #EOFError 文件结束符，KeyError 字典中不存在的键 ValueError 传递给函数或操作的值无效或不正确
    Pass          #占位符，此处无操作
```

输入中存在空行——跳过空行

```
while True:
    try:
        inp=input()
        if inp!="":
            break
    except EOFError:
            exit()
num,diameter=map(int,inp.split())
```

处理大规模输入输出：

```
import sys
input = sys.stdin.read
data = input().split()  # 读入所有数据并分割为列表

import sys
sys.stdout.write('\n'.join(map(str, results)) + '\n')
```

输入一串数字批量转换：list(map(int,input().split()))
字符串转为列表：a=list(input().strip()) #’12345678’ ['1', '2', '3', '4', '5', '6', '7', '8']

进制转换：

```
num = 10
binary_num = bin(num)  #二进制 0b
print(binary_num)  # 输出: 0b1010  如果需要去掉前缀 0b，可以使用切片
octal_num = oct(num)  #八进制 0o
hex_num = hex(num) #十六进制 0x
```

其他进制转为十进制一律 decimal_num = int(binary_num, 2) 要注明基数 2,8,16
其他的转换不能一步到位，需要用十进制过渡

对列表进行操作小心序号移动

```
for island in uncovered :
    if radar >=2*island[1]-island[2]:
        uncovered.remove(island)
```

这是错误操作，例如第 0 项被删除后第 1 项就变成了新的第 0 项，但是 for 指向的下一个 index=1，这样就导致一个点被跳过
正确示例：

```
for i in range(len(uncovered)) :
    island=uncovered[0]
    if island[0]-island[2]<=radar<=island[0]+island[2] :
        uncovered.remove(island)
```

矩阵创建：不要用其他方式以免浅拷贝
简明版 `matrix = [[0 for _ in range(n)] for _ in range(m)]`
复杂情况

```
matrix = []
    for i in range(m):
    row = []
    for j in range(n):
    row.append(0)
    matrix.append(row)
    print(matrix)
```

正则表达式　首先 import re 从左往右挨着写
re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match() 就返回 None
re.match(pattern, string, flags=0)
pattern　匹配的正则表达式　　string　　要匹配的字符串
flags 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。
匹配成功 re.match 方法返回一个匹配的对象，否则返回 None。
re.search 扫描整个字符串并返回第一个成功的匹配
re.search(pattern, string, flags=0)
re.findall
在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果有多个匹配模式，则返回元组列表，如果没有找到匹配的，则返回空列表。
注意： match 和 search 是匹配一次 findall 匹配所有。
pattern.findall(string[, pos[, endpos]])
参数：
pattern 匹配模式 string 待匹配的字符串
pos 可选参数，指定字符串的起始位置，默认为 0
endpos 可选参数，指定字符串的结束位置，默认为字符串的长度
re.sub(pattern, repl, string, count=0, flags=0)查找替换
pattern : 正则中的模式字符串。
repl : 替换的字符串，也可为一个函数。
string : 要被查找替换的原始字符串。
count : 模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。
flags : 编译时用的匹配模式，数字形式。

元字符：具有特殊意义的字符，用于表示复杂的模式。
. 匹配除换行符外的任意字符。
^ 表示匹配输入字符串的开始位置。
$ 表示匹配输入字符串的结束位置。
* 和{0,}一样，匹配前面的子表达式零次或多次。
+ 和{1,}一样，匹配前面的子表达式一次或多次。
? 和{0,1}一样，匹配前面的子表达式零次或一次。`
{n} 匹配确定的 n 次。
[abc] 字符集，匹配方括号内的任意字符。
\d 匹配一个数字字符，等价于 [0-9] 。
\w 匹配包括下划线的任何单词字符，等价于 [A-Za-z0-9_] 。
| 例如： a|b 匹配 a 或者 b
() 匹配括号里面内容
[^xyz]，匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”任一字符
如果要查找特殊字符，需要进行转义，在其前加一个 \

函数可以执行一些操作，不一定要有 return

深度优先搜索 dfs 栈实现

```
def dfs(x, y, c):
    stack = [(x, y)]
    while stack:
        x, y = stack.pop()
        if m[x][y] != c:
            continue
        m[x][y] = '#'
        for dx, dy in dire:
            tx, ty = x + dx, y + dy
            if m[tx][ty] == c:
                stack.append((tx, ty))
```

并查集

```
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_x] = root_y
        size[root_y] += size[root_x]

n, m = map(int, input().split())
parent = list(range(n + 1))
size = [1] * (n + 1)

for _ in range(m):
    a, b = map(int, input().split())
    union(a, b)
#classes = [size[find(x)] for x in range(1, n + 1) if x == parent[x]]
classes = [size[x] for x in range(1, n + 1) if x == parent[x]]
print(len(classes))
```

enumerate(iterable, start=0)

iterable：必需，可迭代对象（列表、元组、字符串等）

start：可选，起始索引值，默认从 0 开始

返回值：一个枚举对象，包含索引和对应元素的元组（(index, value)）

for i,j in enumerate():

子集 递归回溯与系统栈

```
def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans, sol = [], []

        def backtrack(i):
            # 终止条件：处理完所有元素
            if i == n:
                ans.append(sol[:])
                return

            # 分支 1：不选择 nums[i]
            backtrack(i + 1)

            # 分支 2：选择 nums[i]
            sol.append(nums[i]) #选择
            backtrack(i + 1)   #递归
            sol.pop()   # 回溯

        backtrack(0)
        return ans
```

质数欧拉筛

```
def euler_sieve(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 i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes
n = 50
print(euler_sieve(n))
```

二分查找源代码

```
def bisect_left(a, x, lo=0, hi=None, *, key=None):
if lo < 0:
        raise ValueError('lo must be non-negative')
 if hi is None:
        hi = len(a)
 if key is None:
        while lo < hi:
            mid = (lo + hi) // 2
            if a[mid] < x:
                lo = mid + 1
            else:
                hi = mid
 else:
        while lo < hi:
            mid = (lo + hi) // 2
            if key(a[mid]) < x:
                lo = mid + 1
            else:
                hi = mid
 return lo
# 二分查找函数
def binary_search(arr, target):
 index = bisect_left(arr, target)
 if index != len(arr) and arr[index] == target:
        return index    # 返回目标值的索引
 else:
        return -1    # 如果未找到目标值，返回 -1
```

缓存：

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

防爆递归栈：

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

数据量过大直接 dp，lru_chche 无论如何都没有 dp 速度快

最大公约数：

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

最小公倍数（LCM）可以通过它们的乘积除以最大公约数来计算，公式为：LCM(a, b) = |a * b| //GCD(a, b)

bisect 模块：针对已升序排列的列表进行操作，二分查找（O (log n)）

bisect.bisect_left(lst, x) 查找 x 在 lst 中的插入位置，确保插入后列表仍有序（左侧插入，即返回第一个大于等于 x 的位置）

bisect.bisect_right(lst, x) 返回第一个大于 x 的位置（右侧插入，允许重复元素时放在已有相同元素的右边）

bisect.insort_left(lst, x) 将 x 插入到 lst 中，插入位置由 bisect_left 确定，保持列表有序

bisect.insort_right(lst, x) 将 x 插入到 lst 中，插入位置由 bisect_right 确定，保持列表有序

内置排列组合：

```
import itertools
```

排列：itertools.permutations(iterable, r=None) r 为长度，不写默认为全排列，按位置生成因此无去重

如果 iterable 本身有重复元素，去重需要 list(set())操作一次

组合：itertools.combinations(iterable, r) r 为长度必须要写，同样按位生成需要去重

iterable：待生成组合的可迭代对象（列表、字符串、元组、集合等）

```
nums = [1, 2, 3]
perm = itertools.permutations(nums)
print(list(perm)) # 输出：[(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)]
```

广度优先搜索 bfs——走迷宫　队列实现

```
from collections import deque
move=[(1,0),(0,1),(0,-1),(-1,0)]
def bfs(x,y):
    q=deque()
    q.append((0,x,y))
    accessed = {(x,y)}      #已访问集合
    while q:
        step,nowx,nowy=q.popleft()   #先进先出
        if nowx==row and nowy==col:      #终止条件
            return step
        for dx,dy in move:
            nx=nowx+dx
            ny=nowy+dy
            if matrix[nx][ny]==0 and (nx,ny) not in accessed:
                accessed.add((nx,ny))
                q.append((step+1,nx,ny))        #存储到达该点的最短路线长、该点坐标
    return -1   #无法到达时的情况

row,col = map(int, input().split())
matrix=[[1]*(col+2)]
for i in range(row):
    matrix.append([1]+list(map(int, input().split()))+[1])
matrix.append([1]*(col+2))
print(bfs(1,1))
```

矩阵问题智障错误速查：
加保护圈以及输入转换的格式
访问下一点位注意起止点等特殊字符，SE 也属于能访问类型，如果移动对象占据多个格子，自身也可访问
xy 坐标不要打反了，先行后列
起点坐标是否标记为已访问，这点很重要！

bfs 算法逐步扩展，“穿墙”问题不需要考虑入队顺序，路线更短的一定先出
路径回溯：用字典存储某点对应的前一个坐标
last_arrival[(nx,ny)]=(nowx,nowy)

动态规划 dp
不拘一格，dp 数组与状态转移方程，随机应变
务必考虑清楚初始化情况，float(“inf”)还是 0
dp 一般比 recursion 快但是没那么简明，数据量过大直接 dp

bisect 优化的最长连续递增子序列

```
import bisect
n = int(input())
lis =list( map(int, input().split()))
dp = [1e9]*n
for i in lis:
    dp[bisect.bisect_left(dp, i)] = i
print(bisect.bisect_left(dp, 1e8))
```

有权重网格 bfs：堆优化 Dijkstra

```
from heapq import heappush, heappop
move=[(1,0),(0,1),(-1,0),(0,-1)]
def bfs(sx,sy,tx,ty,mapp):
    q=[(0,sx,sy)]            # 优先队列：存储(当前总消耗, 横坐标 x, 纵坐标 y)，堆顶是消耗最小的路径
    accessed={(sx,sy)}    # 访问集合：记录已确定「最小消耗」的单元格，避免重复处理
    while q:
        energy,nowx,nowy=heappop(q)    # 步骤 1：弹出堆顶（当前消耗最小的路径）
        accessed.add((nowx,nowy))        # 标记该单元格为已访问（其最小消耗已确定）
        if nowx==tx and nowy==ty:   # 步骤 2：到达终点，直接返回当前总消耗（堆的特性保证最小值）
            return energy
        for dx,dy in move:                # 步骤 3：遍历上下左右四个方向
            nx,ny=nowx+dx,nowy+dy
            # 合法性校验：①不越界 ②未被访问过 ③可通行（非#）
            if (0<=nx<m and 0<=ny<n) and (nx,ny) not in accessed :
                if mapp[nx][ny]!="#":
                    cost=abs(int(mapp[nx][ny])-int(mapp[nowx][nowy]))
                    heappush(q,(cost+energy,nx,ny))   # 将新路径推入优先队列（允许重复入队）
    return "NO"

m,n,p=map(int,input().split())
Map=[]
for i in range(m):
    Map.append(list(input().split()))
for i in range(p):
    sx,sy,tx,ty=map(int,input().split())
    if Map[sx][sy]=="#" or Map[tx][ty]=="#":   # 提前剪枝：起点/终点是#，直接不可达
        print('NO')
        continue
    print(bfs(sx,sy,tx,ty,Map))
```

单调栈

```
def next_greater_element(arr):
    n = len(arr)
    result = [-1]*n
    stack = []
    for i in range(n):
        #持续处理栈中所有比当前小的元素
        while stack and arr[i] > arr[stack[-1]]:
            top_idx = stack.pop()
            result[top_idx] = arr[i]
        stack.append(i)
    return result

arr = [4,5,2,25]
print(next_greater_element(arr))     # [5,25,25,-1]
```

接水双指针写法

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = left = pre_max = suf_max = 0
        right = len(height) - 1
        while left < right:
            pre_max = max(pre_max, height[left])
            suf_max = max(suf_max, height[right])
            if pre_max < suf_max:
                ans += pre_max - height[left]
                left += 1
            else:
                ans += suf_max - height[right]
                right -= 1
        return ans
```

柱状图接水问题

```
n=int(input())
columns=list(map(int, input().split()))
stack=[]
capacity=0
for i in range(n):
    while stack and columns[stack[-1]]<columns[i]:
        ind=stack.pop()
        bot=columns[ind]
        if not stack:
            break
        distance=i-stack[-1]-1
        capacity+=distance*(min(columns[i],columns[stack[-1]])-bot)
    stack.append(i)
print(capacity)
```

位运算

按位与（&）　a & b

规则：两个数对应二进制位都为 1 时，结果位为 1，否则为 0。

a = 6　# 二进制：0110

b = 3　# 二进制：0011

6 & 3 = 2（二进制 0010）

按位或（|）　a | b

规则：两个数对应二进制位至少一个为 1 时，结果位为 1，否则为 0。

6 | 3 = 7（二进制 0111）

按位异或（^）　a ^ b

规则：两个数对应二进制位不同 时为 1，相同 时为 0（可理解为“无进位加法”）

result = a ^ b

6 ^ 3 = 5（二进制 0101）

按位取反（~）　~a

规则：对数值的每一位二进制位取反（0 变 1，1 变 0）；Python 中整数是补码存储，因此结果等价于 ~a = -(a + 1)。

~6 = -7（推导：~6 = -(6+1) = -7）

左移（<<）　a << n

规则：将 a 的二进制位向左移动 n 位，右侧补 0；等价于 a * (2^n)（效率远高于乘法）

6 << 1 = 12（二进制 1100，6*2=12）

右移（>>）　a >> n

规则：将 a 的二进制位向右移动 n 位，左侧补符号位（正数补 0，负数补 1）；等价于 a // (2^n)（整数除法）

6 >> 1 = 3（二进制 0011，6//2=3）

置某一位为 1	1 << n	10|(1<<3)
置某一位为 0	~(1 << n)	26 & ~(1<<3)
反转某一位	1 << n	10 ^ (1<<3)

旅行商问题——状态压缩 dp

```
def solve():
    n = int(input().strip())
    cost = []
    for _ in range(n):
        cost.append(list(map(int, input().split())))

    INF = 10**9
    # 位运算 1：计算所有可能的状态数（2^n）
    total_mask = (1 << n)  # 等价于 2^n，比如 n=4 时，1<<4=16（0b10000）
    # dp[mask][i]：状态定义
    # mask=访问过的所有城市，i=当前所在城市，值=到达该状态的最小开销
    dp = [[INF] * n for _ in range(total_mask)]

    # 初始化：从城市 0 出发，仅访问城市 0，开销为 0
    # mask=1（0b0001）表示仅访问城市 0
    dp[1][0] = 0

    # 遍历所有可能的状态（mask）
    for mask in range(total_mask):
        # 遍历当前所在的城市 i
        for i in range(n):
            # 跳过无法到达的状态
            if dp[mask][i] == INF:
                continue
            # 尝试前往所有未访问的城市 j
            for j in range(n):
                # 位运算 2：判断城市 j 是否已被访问（mask 的第 j 位是否为 1）
                if mask & (1 << j):
                    continue  # 已访问，跳过
                # 位运算 3：更新状态（将城市 j 加入已访问集合）
                new_mask = mask | (1 << j)  # 把 mask 的第 j 位设为 1
                # 计算新开销：当前开销 + 从 i 到 j 的机票价格
                new_cost = dp[mask][i] + cost[i][j]
                # 更新 dp：保留更小的开销
                if new_cost < dp[new_mask][j]:
                    dp[new_mask][j] = new_cost

    # 所有城市都访问完的状态（mask=0b111...1）
    full_mask = total_mask - 1  # 比如 n=4 时，16-1=15=0b1111
    ans = INF
    # 从任意城市 i 返回起点 0，计算总开销
    for i in range(n):
        if dp[full_mask][i] != INF:
            ans = min(ans, dp[full_mask][i] + cost[i][0])
    print(ans)
```

```
if __name__ == "__main__":
    solve()
```

全排列的位运算写法

```
letters=input()
n=len(letters)
ans=[];now=[]
state=0
def backtrack():
    global ans,now,state
    if len(now)==n:
        ans.append(now[:])
    for i in range(n):
        if state&(1<<i):
            continue
        now.append(letters[i])
        state=state|(1<<i)
        backtrack()
        now.pop()
        state=state&(~(1<<i))
backtrack()
for a in ans:
    print("".join(a))
```

子集的位运算写法

```
n=int(input())
ans=[[]]
for i in range(1,1<<n):
    now=[]
    for j in range(n):
        if i&(1<<j):
            now.append(j+1)
    ans.append(now)
for a in ans:
    print(*a)
```

defaultdict 会为不存在的键自动创建一个默认值，不会报错，默认值的类型由你初始化时指定。
初始化时必须传入一个默认值工厂函数（用来生成默认值的方法），常见的工厂函数有 list、int、str 等内置类型（本质是可调用对象）。

```
from collections import defaultdict
# 初始化：默认值是空列表
student_scores = defaultdict(list)

# 向不存在的键追加元素（自动创建键，值为 []，再追加元素）
student_scores['小明'].append(90)
student_scores['小明'].append(85)
student_scores['小红'].append(95)
print(student_scores)
# 输出：defaultdict(<class 'list'>, {'小明': [90, 85], '小红': [95]})
dict(student_scores))可转为普通字典
```

dict_with_list = defaultdict(list) 默认值为列表（list() 会生成空列表 []）
dict_with_int = defaultdict(int) 默认值为整数 0（int() 会生成 0）可用于计数
dict_with_str = defaultdict(str) 默认值为空字符串（str() 会生成 ""）
也可以用 lambda 定义

```
num_dict = defaultdict(lambda: 100)
print(num_dict['a'])  # 输出：100
```

NBA 门票问题（大数据多重背包）

```
prices=[50,100,250,500,1000,2500,5000]
money=int(input())
nums=list(map(int,input().split()))
min_num={0:0}
shoplist={0:[0,0,0,0,0,0,0]}
for m in range(money):
    if m in min_num:
        for tick in range(7):
            if shoplist[m][tick]<nums[tick]:
                if m+prices[tick] in min_num:
                    if min_num[m+prices[tick]]>min_num[m]+1:
                        min_num[m+prices[tick]]=min_num[m]+1
                        shoplist[m+prices[tick]]=shoplist[m].copy()
                        shoplist[m+prices[tick]][tick]+=1
                else:
                    min_num[m+prices[tick]]=min_num[m]+1
                    shoplist[m+prices[tick]]=shoplist[m].copy()
                    shoplist[m+prices[tick]][tick]+=1
if money in min_num:
    print(min_num[money])
else:
    print("Fail")
```

Candy 问题：需要穷举，注意问题的拆分

```
n,m = map(int,input().split())
nums = []
s = []
for i in range(n):
    x,y = map(int,input().split())
    nums.append(x)
    s.append(x+y)
minsum = min(s)
ans = 0
nums.sort()
pre = [0]
for ni in nums:
    pre.append(pre[-1]+ni)
for i,p in enumerate(pre):
    ans = max(ans,((m-p)//minsum)*2+i)
print(ans)
```

辅助栈和懒删除

```
pig, pigmin = [], []
while True:
```

```
    try:
        *line, = input().split()
        if "pop" in line:
            if len(pig) == 0:
                continue

            val = pig.pop()
            if len(pigmin) > 0 and val == pigmin[-1]:
                pigmin.pop()
        elif "push" in line:
            val = int(line[1])
            pig.append(val)
            if len(pigmin) == 0 or val <= pigmin[-1]:
                pigmin.append(val)
        elif "min" in line:
            if len(pig) == 0:
                continue
            else:
                print(pigmin[-1])
    except EOFError:
        break
```

滑动窗口与单调队列

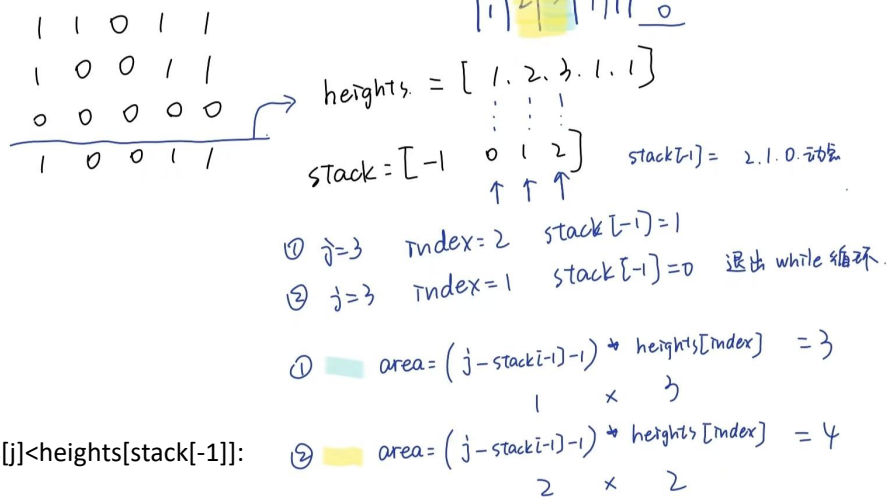
```
from collections import deque
from typing import List
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums or k == 0:
            return []
        n = len(nums)
        if k == 1:
            return nums
        deque_index = deque()  # 存储索引，保持双端队列中的值递减
        res = []
        for i in range(n):
            # 移除滑出窗口的元素（队首元素）
            if deque_index and deque_index[0] < i - k + 1:
                deque_index.popleft()
            # 移除所有小于当前元素的队尾元素
            while deque_index and nums[deque_index[-1]] < nums[i]:
                deque_index.pop()
            # 将当前元素的索引加入队列
            deque_index.append(i)
            # 从第 k 个元素开始记录结果，队首始终是窗口的最大值
            if i >= k - 1:
                res.append(nums[deque_index[0]])
```

```
        return res
if __name__ == "__main__":
    sol = Solution()
    print(sol.maxSlidingWindow([1,3,-1,-3,5,3,6,7], 3)) # [3,3,5,5,6,7]
```

网格最大面积问题（二维单调栈）

```
import sys
row,col=map(int,input().split())
forest_map=sys.stdin.readlines()
matrix=[]
for line in forest_map:
    matrix.append(list(line.strip().split()))
heights=[0]*(col+1)
max_area=float('-inf')

for i in range(row):
    stack=[-1]
    for j in range(col+1):
        if j!=col:
            if matrix[i][j]=='1':
                heights[j]=0
            else:
                heights[j]+=1
        while stack[-1]!=-1 and heights[j]<heights[stack[-1]]:
            index=stack.pop()
            now_area=heights[index]*(j-stack[-1]-1)
            if now_area>max_area:
                max_area=now_area
        stack.append(j)
print(max_area)
```



合并后的最大最小值、最小最大值：二分查找

本质上不是 greedy，而是枚举法猜测答案并进行验证

跳房子——最大化最小值

```
def solve():
    import sys
    end,stone_num,max_remove=map(int,input().split())
    stones=list(map(int,sys.stdin.read().strip().split()))
    stones=[0]+stones+[end]
    ans=-1

    def is_valid(min_distance,stone_num,max_remove,stones):
        former=0;remove=0
        for i in range(1,stone_num+2):
            now=stones[i]
```

```
            if now-former<min_distance:
                remove+=1
                if remove>max_remove:
                    return False
            else:
                former=now
        return True

    left,right=0,end    #预设取值范围，最小距离大于 0 并且不超过起止点之间的距离
    while left<right:
        mid=(left+right)//2
        if is_valid(mid,stone_num,max_remove,stones):
            left=mid+1
            ans=mid
        else:
            right=mid
    print(ans)
if __name__=='__main__':
    solve()
```

月度开销——最小化最大值

```
def solve():
    import sys
    n,m=map(int,input().split())
    months=list(map(int,sys.stdin.read().strip().split()))

    def can_divide(max_cost,m):
        former_sum=0
        divide=1                            #注意正确设置初始值
        for i in range(n):
            if months[i]>max_cost:          #剪枝
                return False
            if months[i]+former_sum>max_cost:
                divide+=1
                if divide>m:
                    return False
                former_sum=months[i]
            else:
                former_sum+=months[i]
        return True
    ans=-1                                  #注意初始化数值，如果 ans 未被修改说明 mid=left=right
    left,right=max(months),sum(months)+1 #预设可能范围，最大月开销不小于最大单个月数值，小于总和
    while left<right:
        mid=(left+right)//2
        if can_divide(mid,m):               #可行则可以试试更多的划分，对应更小的花销，取左侧
            right=mid
            ans=mid
```

```
            else:
                left=mid+1
        print(ans) if ans!=-1 else print(right)

if __name__=='__main__':
        solve()
```

区间问题 先画图想清楚思路

1、区间合并：给出一堆区间，要求合并所有有交集的区间 （端点处相交也算有交集）。最后问合并之后的区间。（校门外的树）

按照区间左端点从小到大排序。

从前往后枚举每一个区间，判断是否应该将当前区间视为新区间。

2、最多不相交区间、区间选点（radar installation）

按照区间右端点从小到大排序。

从前往后依次枚举每个区间，判断与前面的是否有交叉。

3、区间覆盖：给出一堆区间和一个目标区间，问最少选择多少区间可以覆盖掉题中给出的这段目标区间。（世界杯只因）

按照区间左端点从小到大排序。

从前往后依次枚举每个区间，在所有能覆盖当前目标区间起始位置 start 的区间之中，选择右端点最大的区间。

4、区间分组

转化为时间轴问题求解