1. 动态规划

T1：购物单

描述

王强决定把年终奖用于购物，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机，扫描仪
书柜	图书
书桌	台灯，文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件，且每件物品只能购买一次。
每个主件可以有 0 个、 1 个或 2 个附件。附件不再有从属于自己的附件。
王强查到了每件物品的价格（都是 10 元的整数倍），而他只有 N 元的预算。除此之外，他给每件物品规定了一个重要度，用整数 1 ~ 5 表示。他希望在花费不超过 N 元的前提下，使自己的满意度达到最大。
满意度是指所购买的每件物品的价格与重要度的乘积的总和，假设设第i件物品的价格为$v[i]$，重要度为$w[i]$，共选中了k件物品，编号依次为$j_1, j_2, ..., j_k$，则满意度为：$v[j_1]*w[j_1]+v[j_2]*w[j_2]+...+v[j_k]*w[j_k]$。（其中 * 为乘号）
请你帮助王强计算可获得的最大的满意度。

输入描述：

输入的第 1 行，为两个正整数N，m，用一个空格隔开：
（其中 N （ N<32000 ）表示总钱数， m （m <60 ）为可购买的物品的个数。）

从第 2 行到第 m+1 行，第 j 行给出了编号为 j-1 的物品的基本数据，每行有 3 个非负整数 v p q

（其中 v 表示该物品的价格（ v<10000 ）， p 表示该物品的重要度（ 1 ~ 5 ）， q 表示该物品是主件还是附件。如果 q=0 ，表示该物品为主件，如果 q>0 ，表示该物品为附件， q 是所属主件的编号）

输出描述：

输出一个正整数，为张强可以获得的最大的满意度。

示例1

输入：
```
1000 5
800 2 0
400 5 1
300 5 1
400 3 0
500 2 0
```
复制

输出：
```
2200
```
复制

示例2

输入：
```
50 5
20 3 5
20 3 5
10 3 0
10 2 0
10 1 0
```
复制

输出：
```
130
```
复制

说明：由第1行可知总钱数N为50以及希望购买的物品个数m为5；
第2和第3行的q为5，说明它们都是编号为5的物品的附件；
第4~6行的q都为0，说明它们都是主件，它们的编号依次为3~5；
所以物品的价格与重要度乘积的总和的最大值为10*1+20*3+20*3=130

关键点

状态：dp[i][j]为最大使用金额i时有j件物品可选择时的最大价值；

转移方程：当前最大价值=max(不买这个物品（买前一个物品）的价值，不买前一个物品(退钱)买这个物品的价值)

解法

```python
money, number = list(map(int, input().strip().split()))
money = money // 10
main = [[[0, 0], [0, 0], [0, 0]] for _ in range(number + 2)]  # 依次存放第一个主件和它的附件
for i in range(1, 1 + number):
    v, p, q = list(map(int, input().strip().split()))
    if q == 0:  # 主件
        main[i][0] = [v // 10, v * p]
    else:
        if main[q][1][0] == 0:
            main[q][1] = [v // 10, v * p]
        else:
            main[q][2] = [v // 10, v * p]
main = [i for i in main if i[0][0] != 0]  # 去掉编号是空的主件
dp = [[0 for i in range(1 + money)] for j in range(1 + len(main))]
for num in range(1, 1 + len(main)):
    for mon in range(1 + money):
        dp[num][mon] = dp[num - 1][mon]  # 当前物品继承上一个物品的最大价值
        (p1, v1), (p2, v2), (p3, v3) = main[num - 1]
        # 不买这个物品（买前一个物品） 和 不买前一个物品买这个物品的最大值
        if mon >= p1 + p2 + p3:  # 这里必须是dp[num][mon]不能是dp[num-1][mon]，因为这个值是在每个if更新的
            dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p2 - p3] + v1 + v2 + v3)
        if mon >= p1 + p3:
            dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p3] + v1 + v3)
        if mon >= p1 + p2:
            dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1 - p2] + v1 + v2)
        if mon >= p1:
            dp[num][mon] = max(dp[num][mon], dp[num - 1][mon - p1] + v1)

print(dp[-1][-1])
```

T2：合唱队

描述

N 位同学站成一排，音乐老师要请最少的同学出列，使得剩下的 K 位同学排成合唱队形。
设K位同学从左到右依次编号为 1，2…，K，他们的身高分别为$T_1, T_2, \ldots, T_K$，若存在$i(1 \leq i \leq K)$使得$T_1 < T_2 < \ldots\ldots < T_{i-1} < T_i$且$T_i > T_{i+1} > \ldots\ldots > T_K$，则称这$K$名同学排成了合唱队形。
通俗来说，能找到一个同学，他的两边的同学身高都依次严格降低的队形就是合唱队形。
例子：
123 124 125 123 121 是一个合唱队形
123 123 124 122不是合唱队形，因为前两名同学身高相等，不符合要求
123 122 121 122不是合唱队形，因为找不到一个同学，他的两侧同学身高递减。

你的任务是，已知所有N位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

注意：不允许改变队列元素的先后顺序 且 不要求最高同学左右人数必须相等

数据范围：$1 \leq n \leq 3000$

输入描述：

用例两行数据，第一行是同学的总数 N，第二行是 N 位同学的身高，以空格隔开

输出描述：

最少需要几位同学出列

示例1

输入：8
186 186 150 200 160 130 197 200
复制

输出：4
复制

说明：由于不允许改变队列元素的先后顺序，所以最终剩下的队列应该为186 200 160 130或150 200 160 130

关键点

合唱队最多人数实质是从左往右和从右往左的每个位置最长递增子序列个数的和的最大值。

状态：dp[i]为第i个位置的最长递增子序列个数。

转移方程：当前位置最长递增子序列为=当前位置之前位置里，小于当前位置的最大值，它的子序列个数+1

解法

```python
def left_to_right_max_count(ls):
    """
    找出从左往右每个位置左侧最长递增子序列元素数量。
    1、循环列表每个元素i。
    2、循环列表当前元素的前面的元素j。
    3、如果当前元素i比前面的元素j大，且dp[i]较小，则dp更新当前元素i位置的数量为j元素数量+1
    :param ls:
    :return:
    """
    count = len(ls)
    dp = [1 for _ in range(count)]
    for i in range(0, len(ls)):
        for j in range(0, i):
            if ls[i] > ls[j] and dp[i] <= dp[j]:  # 注意and后面进行判断再赋值速度更快
                dp[i] = dp[j] + 1
    return dp

n = int(input())
hight_ls = list(map(int, input().split(' ')))
left = left_to_right_max_count(hight_ls)
right = left_to_right_max_count(hight_ls[::-1])[::-1]
```

```
total = [left[i] + right[i] - 1 for i in range(len(left))]
print(len(hight_ls) - max(total))
```

进阶

```
def left_to_right_max_count2(in_arr):
    """
    1.先取出列表第一个元素放在待定的最长递增子序列列表里，再循环其他元素，
    2.如果i大于最长递增子序列就添加到其最后,当前位置的最长递增子序列数量为列表长度
    3.如果i小于就把i替换掉最长递增子序列比它大的第一个数，当前位置的最长递增子序列数量和前一个一致
    :param in_arr:
    :return:
    """
    longest = [1] * len(in_arr)
    result = [in_arr[0]]
    arr_ls = list(enumerate(in_arr, 0))
    for idx, value in arr_ls[1:]:
        if value > result[-1]:
            result.append(value)
            longest[idx] = len(result)
        else:
            idx = bisect.bisect_left(result, value)
            result[idx] = value
            longest[idx] = longest[idx - 1]
    return longest

n = int(input())
hight_ls = list(map(int, input().split(' ')))
left = left_to_right_max_count2(hight_ls)
right = left_to_right_max_count2(hight_ls[::-1])[::-1]
total = [left[i] + right[i] - 1 for i in range(len(left))]
print(len(hight_ls) - max(total))
```

T3：素数伴侣

描述

题目描述
若两个正整数的和为素数，则这两个正整数称之为“素数伴侣”，如2和5、6和13，它们能应用于通信加密。现在密码学会请你设计一个程序，从已有的 N （ N 为偶数）个正整数中挑选出若干对组成“素数伴侣”，挑选方案多种多样，例如有4个正整数：2，5，6，13，如果将5和6分为一组中只能得到一组“素数伴侣”，而将2和5、6和13编组将得到两组“素数伴侣”，能组成“素数伴侣”最多的方案称为“最佳方案”，当然密码学会希望你寻找出“最佳方案”。
输入:
有一个正偶数 n ，表示待挑选的自然数的个数。后面给出 n 个具体的数字。
输出:
输出一个整数 K ，表示你求得的“最佳方案”组成“素数伴侣”的对数。

数据范围： $1 \leq n \leq 100$ ，输入的数据大小满足 $2 \leq val \leq 30000$

输入描述:

输入说明
1 输入一个正偶数 n
2 输入 n 个整数

输出描述:

求得的“最佳方案”组成“素数伴侣”的对数。

示例1

输入：4
2 5 6 13 复制

输出：2 复制

示例2

输入：2
3 6 复制

输出：0 复制

关键点

和为素数必为偶数+奇数；将输入分为偶数和奇数；提前好计算奇数和偶数所有组合能否构成素数，避免循环中重复计算；用匈牙利算法求最大匹配

1、遍历偶数，寻找可以组成素数伴侣的奇数。

2、给奇数创建访问列表，确保当前偶数对每个奇数只会访问一遍。

3、如该偶数找到且奇数未被访问。如未被匹配，则匹配。如奇数已匹配，则给被匹配的偶数重新递归找一个。

解法

```python
def fun2():
    def is_prime(_value):
        if _value == 1:
            return False
        for _i in range(2, 1 + int(_value ** 0.5)):
            if _value % _i == 0:
                return False
        return True

    # 遍历每一个左边节点，给它找一个右边节点
    def find(i, _left_visit_right):
        for j, right in enumerate(right_lis):  # 遍历右
            if matchable[i][j] and _left_visit_right[j] is False:  # 如果左右能匹配,且没访问过
                _left_visit_right[j] = True
                if match_dict.get(j) is None:  # 右边没有匹配
                    match_dict[j] = i  # 给右边匹配左边
                    return True
```

```
                elif find(match_dict.get(j), _left_visit_right[::]):  # 右边已经匹配上了左某，试着给左某的再找过一个
                    match_dict[j] = i  # 如果能给左某找到，则右现匹配左现
                    return True
        return False

    n = input()
    arr = list(map(int, input().split()))
    left_lis = [i for i in arr if i % 2 == 0]  # 二分图左边
    right_lis = [i for i in arr if i % 2 != 0]  # 二分图右边
    matchable = [[is_prime(i + j) for j in right_lis] for i in left_lis]  # 二分图里所有可能的路径能否匹配的结果
    match_dict = dict()
    for i, left in enumerate(left_lis):
        left_visit_right = [False for j in right_lis]  # 每一个左边保存一个它右边是否访问列表,必须在此处初始化。每次递归都是新开始
        find(i, left_visit_right)
    print(len(match_dict))


fun2()
```

T4：最长回文子串

描述

Catcher是MCA国的情报员，他工作时发现敌国会用一些对称的密码进行通信，比如像这些ABBA，ABA，A，123321，但是他们有时会在开始或结束时加入一些无关的字符以防止别国破解。比如进行下列变化 ABBA->12ABBA,ABA->ABAKK,123321->51233214 。因为截获的串太长了，而且存在多种可能的情况（abaaab可看作是aba,或baaab的加密形式），Cathcer的工作量实在是太大了，他只能向电脑高手求助，你能帮Catcher找出最长的有效密码串吗?

数据范围：字符串长度满足 $1 \leq n \leq 2500$

输入描述:

输入一个字符串（字符串的长度不超过2500）

输出描述:

返回有效密码串的最大长度

示例1

输入：ABBA 复制

输出：4 复制

示例2

输入：ABBBA 复制

输出：5 复制

示例3

输入：12HHHHA 复制

输出：4 复制

关键点

遍历所有可能的子串长度，遍历字符串，不断规划各个长度的i到j子串是否回文。

状态：dp[i][j]表示从索引i到j的子串是否回文。

转移方程：i到j是否回文=(i到j长度小于等于2 or i+1到j-1是否回文) and i位置是否等于j位置

解法

```python
def longest_palindrome(s):
    # 如果字符串长度为0，返回0
    if len(s) == 0:
        return 0
    # 获取字符串长度
    n = len(s)
    # 初始化最长回文子串长度为0
    longest_len = 0
    # 创建一个二维数组，用于存储每个可能的子串是否是回文
    dp = [[False] * n for _ in range(n)]
    # 遍历所有可能的子串长度
    for length in range(1, n + 1):
        # 对每个子串进行检查
        for start in range(n):
            # 计算子串的结束位置
            end = start + length - 1
            # 如果结束位置超过字符串长度，跳出循环
            if end >= n:
                break
            # 判断子串是否是回文
            dp[start][end] = (length == 1 or length == 2 or dp[start + 1][end - 1]) and s[start] == s[end]
            # 如果子串是回文，并且长度大于当前最长回文子串长度，更新最长回文子串长度
            if dp[start][end] and length > longest_len:
                longest_len = length
    # 返回最长回文子串长度
    return longest_len
```

T5：统计每个月兔子个数

描述

有一种兔子，从出生后第3个月起每个月都生一只兔子，小兔子长到第三个月后每个月又生一只兔子。
例子：假设一只兔子第3个月出生，那么它第5个月开始会每个月生一只兔子。
一月的时候有一只兔子，假如兔子都不死，问第n个月的兔子总数为多少?
数据范围：输入满足 $1 \leq n \leq 31$

输入描述:

输入一个int型整数表示第n个月

输出描述:

输出对应的兔子总数

示例1

输入：3 复制
输出：2 复制

关键点

状态：dp[i]为第i个月兔子数量，前两个月数量为1，dp[0]=dp[1]=dp[2]=1。

转移方程：dp[i] = dp[i-1] + dp[i-2]，i月为上个月的兔子 + 上上个月的兔子数量(会再生一个)

解法

```python
def fun():
    n = int(input())
    dp = [0 for _ in range(n+1)]  # 第n个月的兔子数量
    dp[0] = dp[1] = dp[2] = 1  # 0月不算 前2个月兔子数量是1
    for i in range(3, 1+n):  # 从第3个月开始计算
        dp[i] = dp[i-1] + dp[i-2]
    print(dp[n])


fun()
```

T6：计算字符串的最小编辑距离

描述

Levenshtein 距离，又称编辑距离，指的是两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。许可的编辑操作包括**将一个字符替换成另一个字符，插入一个字符，删除一个字符**。编辑距离的算法是首先由俄国科学家 Levenshtein 提出的，故又叫 Levenshtein Distance 。
例如：
字符串A: abcdefg
字符串B: abcdef
通过增加或是删掉字符 "g" 的方式达到目的。这两种方案都需要一次操作。把这个操作所需要的次数定义为两个字符串的距离。
要求：
给定任意两个字符串，写出一个算法计算它们的编辑距离。

数据范围：给定的字符串长度满足 $1 \leq len(str) \leq 1000$

输入描述：

每组用例一共2行，为输入的两个字符串

输出描述：

每组用例输出一行，代表字符串的距离

示例1

输入：abcdefg
abcdef
复制

输出：1
复制

关键点

状态：dp[i][j]为字符串s1的前i位和字符串s2的前j位的最小编辑距离，i或j为0时最小编辑距离为另一个字符串的长度。

转移方程：s1少一位编辑到s2和s2少一位编辑到s1的距离+1 和 s1与s2都少一位时加s1[i]和s2[j]是否不相等；

dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + (s1[i-1] != s2[j-1]))

解法

```python
def fun():
    n = input()
    m = input()
    dp = [[0 for i in range(len(m)+1)] for j in range(len(n)+1)]  # i和j位时的编辑距离，因为有0位，所以要+1
    for i in range(len(m)+1):  # 给默认值当一个字符串是空串时，编辑距离就是另一个字符串的长度
        dp[0][i] = i
```

```
    for i in range(len(n)+1):  # 给默认值当一个字符串是空串时，编辑距离就是另一个字符串的长度
        dp[i][0] = i
    for i in range(1, len(n)+1):  # 因为0字符的时候已经有默认值了，所以从1个字符开始遍历
        for j in range(1, len(m)+1):
            # 有i个或者j个字符的时候对应的是字符串索引是i-1或者j-1
            dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1] + (n[i-1] != m[j-1]))
    print(dp[-1][-1])


fun()
```

T7：盘子分苹果的分法

描述

把m个**同样**的苹果放在n个**同样**的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？
注意：如果有7个苹果和3个盘子，（5，1，1）和（1，5，1）被视为是同一种分法。

数据范围：$0 \leq m \leq 10$，$1 \leq n \leq 10$。

输入描述：

输入两个int整数

输出描述：

输出结果，int型

示例1

输入：7 3 复制

输出：8 复制

关键点

状态：dp[i][j]为盘子i个苹果j个时的分法。

转移方程：

盘子多，苹果少的情况：i个盘子和i-1个盘子是一样的次数。dp[i][j] = dp[i - 1][j]。

盘子和苹果相等的情况：等于少一个盘子的次数+1(多一个盘子多一种次数)。dp[i][j] = dp[i - 1][j] + 1。

盘子少，苹果多的情况：等于空一个盘子的次数+不空盘子的次数(不空盘子时，每个盘子必有一个，为多余苹果放置的次数)。 dp[i][j] = dp[i - 1][j] + dp[i][j - i]。

解法

```
def fun_dp():
    apple, dish = list(map(int, input().split(' ')))
    dp = [[0 for _ in range(apple + 1)] for _ in range(dish + 1)]  # 初始化dp数组
    for i in range(dish + 1):
        dp[i][0] = 1  # 0个苹果的放法只有1种
    for i in range(1, dish + 1):
        for j in range(1, apple + 1):
            if j < i:  # 盘子多 苹果少
                dp[i][j] = dp[i - 1][j]
            elif i == j:  # 盘子相等苹果，等于少一个盘子的放法+多一个盘子多一种放法
```

```
                dp[i][j] = dp[i - 1][j] + 1
            else:  # 盘子少 苹果多
                dp[i][j] = dp[i - 1][j] + dp[i][j - i]  # 盘子空着的，加盘子不空，每个盘子放一个，多余的果放盘子的方法数
    print(dp[-1][-1])
```

T8：求最长公共子串

描述

查找两个字符串a,b中的最长公共子串。若有多个，输出在较短串中最先出现的那个。
注：子串的定义：将一个字符串删去前缀和后缀（也可以不删）形成的字符串。请和“子序列”的概念分开！

数据范围：字符串长度$1 \leq length \leq 300$
进阶：时间复杂度：$O(n^3)$，空间复杂度：$O(n)$

输入描述：

输入两个字符串

输出描述：

返回重复出现的字符

示例1

输入：abcdefghijklmnop
abcsafjklmnopqrstuvw
复制

输出：jklmnop
复制

关键点

状态：dp[i][j]为s1到i位和s2到j位的最长公共子串长度。

转移方程：s1[i]等于s2[j]时，如果是s1或s2开头，那么dp[i][j]为1，否则dp[i][j]为dp[i-1][j-1]+1。

解法

```
def fun_dp():
    s1, s2 = input(), input()
    s1, s2 = (s1, s2) if len(s1) < len(s2) else (s2, s1)  # s1是短的
    dp = [[0 for _ in s2] for _ in s1]  # s1的i位和s2的j位时的最大公共子串长度
    max_len, max_com_sub_s = 0, ''
    for i in range(0, len(s1)):  # 遍历s1的每个位置
        for j in range(0, len(s2)):  # 遍历s2的每个位置
            if s1[i] == s2[j]:  # 当前位置相等时，
                if i == 0 or j == 0:  # 如果是开头，那自然是1
                    dp[i][j] = 1
                else:   # 动态转移方程为s1 s2上一位的dp值+1
                    dp[i][j] = dp[i-1][j-1] + 1
                if dp[i][j] > max_len:  # 更新最大值，因为最大值不是dp[-1][-1]
                    max_len = dp[i][j]
                    max_com_sub_s = s2[j-max_len+1:j+1]
    print(max_com_sub_s)
```

T9：走方格的方案数

描述

请计算n*m的棋盘格子（n为横向的格子数，m为竖向的格子数）从棋盘左上角出发沿着边缘线从左上角走到右下角，总共有多少种走法，要求不能走回头路，即：只能往右和往下走，不能往左和往上走。

注：沿棋盘格之间的边缘线行走

数据范围：$1 \leq n, m \leq 8$

输入描述：

输入两个正整数n和m，用空格隔开。(1≤n,m≤8)

输出描述：

输出一行结果

示例1

输入：2 2 复制

输出：6 复制

关键点

状态：dp[i][j]为第i行j列的格子的方法数。

转移方程：dp[i][j] = dp[i - 1][j] + dp[i][j - 1] if i != 0 and j != 0 else 1，等于左边来的方法数+上面来的方法数。

解法

```
def fun():
    n, m = list(map(int, input().strip().split(' ')))
    dp = [[1 for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for j in range(m + 1):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1] if i != 0 and j != 0 else
1
    print(dp[-1][-1])

fun()
```

T10：TSP问题

有n个城市，每个城市去其他城市距离为输入。求一个城市经过其他城市到目标城市的最短距离。

假设有4个城市，距离为dis。dis[i][j]为城市i到城市j的距离，求城市0经过城市123又回到0的最短距离。

```
dis = [[0, 3, 6, 7],
       [5, 0, 2, 3],
       [6, 4, 0, 2],
       [3, 7, 5, 0]]
```

关键点

此类问题属于状态压缩动态规划，记录的状态比较复杂需要压缩。

状态：dp[i][j]为在城市i，已经去过了城市集合j的情况下的最短距离。

转移方程：在城市i，去过了城市集合j的最短距离 = 从城市集合j里来城市i的最短的城市的距离 = min([在城市k，去过了城市集合j-i的距离 for i in j])

解法

```python
def tsp1(dis):
    # n 是城市的数量
    n = len(dis)
    # INF 是一个表示无穷大的值，用于初始化 dp 数组
    inf = float('inf')
    # dp[i][j] 表示访问过的城市集合为 j，当前在城市 i 的最短路径长度
    dp = [[inf] * (1 << n) for _ in range(n)]
    # 初始化 dp 数组，如果我们直接从城市 0 出发到城市 i，那么路径长度就是 dis[i][0]
    for i in range(n):
        dp[i][0] = dis[i][0]

    # 遍历所有可能的城市集合 j
    for j in range(1, 1 << n):
        # 对于每个集合，我们遍历每个城市 i
        for i in range(n):
            # 如果 i 不在 j 中，我们就跳过
            if ((j >> i) & 1) == 0:
                continue
            # 我们尝试从一个不同的城市 k 转移到 i
            for k in range(n):
                # 如果 k 和 i 是同一个城市，我们就跳过。当前在的城市i一定在已访问的集合j中，不在则无意义
                if k == i:
                    continue
                # dp[k][j ^ (1 << i)]为在城市k去过了城市j里不包含i的的最小距离
                # 我们更新 dp[i][j] 为所有可能的 dp[k][j^(1<<i)] + dis[k][i] 的最小值
                dp[i][j] = min(dp[i][j], dp[k][j ^ (1 << i)] + dis[k][i])
    # 我们返回从任何一个城市出发，访问所有城市并返回原点的最短路径长度
    return min(dp[i][(1 << n) - 1] + dis[i][0] for i in range(n))

dis = [[0, 3, 6, 7],
       [5, 0, 2, 3],
       [6, 4, 0, 2],
       [3, 7, 5, 0]]
print(tsp1(dis))
```

2. 深度优先搜索

T1：迷宫问题

描述

定义一个二维数组 N*M ，如 5 × 5 数组下所示:

int maze[5][5] = {
0, 1, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 1, 0,
};

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的路线。入口点为[0,0],既第一格是可以走的路。

数据范围： $2 \le n,m \le 10$ ， 输入的内容只包含 $0 \le val \le 1$

输入描述:

输入两个整数，分别表示二维数组的行数，列数。再输入相应的数组，其中的1表示墙壁，0表示可以走的路。数据保证有唯一解,不考虑有多解的情况，即迷宫只有一条通道。

输出描述:

左上角到右下角的最短路径，格式如样例所示。

示例1

输入：5 5
0 1 0 0 0
0 1 1 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0

复制

输出：(0,0)
(1,0)
(2,0)
(2,1)
(2,2)
(2,3)
(2,4)
(3,4)
(4,4)

复制

示例2

输入：5 5
0 1 0 0 0
0 1 0 1 0
0 0 0 0 1
0 1 1 1 0
0 0 0 0 0

复制

输出：(0,0)
(1,0)
(2,0)
(3,0)
(4,0)
(4,1)
(4,2)
(4,3)
(4,4)

复制

说明：注意：不能斜着走！！

关键点

递归搜索每个位置x,y能通行的各个位置，每次路过位置标记为已访问，如最后无路可走则返回False，回退到上一个能通行的支路口。

解法

```
def fun():
    def dfs(x, y):
        visit[x][y] = True
        if x == n - 1 and y == m - 1:  # 找到了迷宫出口
            maze_path.append((x, y))
            return True
        passable = [True, True, True, True]  # 上下左右是否可以通行
```

```
        if x == 0:  # 第一行
            passable[0] = False
        if x == n - 1:  # 最后一行
            passable[1] = False
        if y == 0:  # 第一列
            passable[2] = False
        if y == m - 1:  # 最后一列
            passable[3] = False
        if passable[0] and maze[x - 1][y] == '0' and (visit[x - 1][y] is False):  # 往上可行
            if dfs(x - 1, y):
                maze_path.append((x, y))
                return True
        if passable[1] and maze[x + 1][y] == '0' and (visit[x + 1][y] is False):  # 往下可行
            if dfs(x + 1, y):
                maze_path.append((x, y))
                return True
        if passable[2] and maze[x][y - 1] == '0' and (visit[x][y - 1] is False):  # 往左可行
            if dfs(x, y - 1):
                maze_path.append((x, y))
                return True
        if passable[3] and maze[x][y + 1] == '0' and (visit[x][y + 1] is False):  # 往右可行
            if dfs(x, y + 1):
                maze_path.append((x, y))
                return True
        return False

    n, m = list(map(int, input().split(' ')))  # n行， m列
    maze = [input().split(' ') for _ in range(n)]
    maze_path = []
    visit = [[False for _ in range(m)] for _ in range(n)]  # 当前点是否走过
    dfs(0, 0)
    maze_path = maze_path[::-1]  # 倒放
    for path in maze_path:
        print(f'({path[0]},{path[1]})')


fun()
```

T2：求数独

描述

问题描述：数独（Sudoku）是一款大众喜爱的数字逻辑游戏。玩家需要根据9X9盘面上的已知数字，推算出所有剩余空格的数字，并且满足**每一行、每一列、每一个3X3粗线宫内**的数字均含1-9，并且不重复。

示例1

输入：
```
0 9 2 4 8 1 7 6 3
4 1 3 7 6 2 9 8 5
8 6 7 3 5 9 4 1 2
6 2 4 1 9 5 3 7 8
7 5 9 8 4 3 1 2 6
1 3 8 6 2 7 5 9 4
2 7 1 5 3 8 6 4 9
3 8 6 9 1 4 2 5 7
0 4 5 2 7 6 8 3 1
```

输出：
```
5 9 2 4 8 1 7 6 3
4 1 3 7 6 2 9 8 5
8 6 7 3 5 9 4 1 2
6 2 4 1 9 5 3 7 8
7 5 9 8 4 3 1 2 6
1 3 8 6 2 7 5 9 4
2 7 1 5 3 8 6 4 9
3 8 6 9 1 4 2 5 7
9 4 5 2 7 6 8 3 1
```

关键点

遍历所有格子，当前格子有值则跳过。当前格子是空的则求出当前格子当前数独棋盘的所有合法值，遍历所有合法值，每次遍历时dfs下一个格子。如果最后没有合法值了，则返回False，同时将上一步填入的值回溯为0。

解法

```python
def fun():
    def find_legal_value(x, y):
        # 查找当前坐标点的合法可填值
        x_axis = set(board[x])
        y_axis = {board[i][y] for i in range(9)}
        sub_board = {board[_x][_y] for _x in [x - x % 3, x - x % 3 + 1, x - x % 3 + 2] for _y in
                     [y - y % 3, y - y % 3 + 1, y - y % 3 + 2]}
        legal_value = set(range(1, 10)) - x_axis - y_axis - sub_board
        if legal_value:
            return legal_value
        else:
            return False

    # 从0,0开始，遍历它的所有合法取值，即行列九宫格都不包含的数字
    def dfs(x, y):
        # 如果已经遍历到了最后一个(8,8),则返回True
        if x == 8 and y == 8:
            value = find_legal_value(8, 8)
            # 最后一个格子为空则填入
            if value:
                for i in value:
                    board[8][8] = i
            return True
        # 超出换行
        y = y + 1 if x == 9 else y
        x = x % 9
        # 找下一个为0的格子
        while board[x][y] != 0:
            x += 1
            # 超出换行
            y = y + 1 if x == 9 else y
```

```
            x = x % 9
            if x == 8 and y == 8:
                value = find_legal_value(8, 8)
                # 最后一个格子为空则填入
                if value:
                    for i in value:
                        board[8][8] = i
                return True
        # 检查当前格子是否可以填入
        legal_value = find_legal_value(x, y)
        if legal_value:
            # 可以填入则遍历所有可以填入的值
            for value in legal_value:
                board[x][y] = value  # 填入
                if dfs(x + 1, y):  # dfs下一个
                    return True
                else:
                    board[x][y] = 0  # 回溯
        else:  # 不可以则返回False
            return False

board = [list(map(int, input().split(' '))) for _ in range(9)]
dfs(0, 0)
for i in board:
    i = list(map(str, i))
    print(' '.join(i))
```

T3：盘子分苹果的分法

描述

把m个**同样**的苹果放在n个**同样**的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？
注意：如果有7个苹果和3个盘子，（5，1，1）和（1，5，1）被视为是同一种分法。

数据范围：$0 \leq m \leq 10$，$1 \leq n \leq 10$。

输入描述：

输入两个int整数

输出描述：

输出结果，int型

示例1

输入：7 3 复制

输出：8 复制

关键点

递归i个苹果和j个碟子时的次数，分3种情况。

盘子多，苹果少的情况：i个盘子和i-1个盘子是一样的次数。

盘子和苹果相等的情况：等于少一个盘子的次数+1(多一个盘子多一种次数)。

盘子少，苹果多的情况：等于空一个盘子的次数+不空盘子的次数(不空盘子时，每个盘子必有一个，为多余苹果放置的次数)。

解法

```python
def fun_dfs():
    def dfs(apple, dish):
        if apple == 1 or dish == 1:
            return 1
        elif apple < dish:  # 苹果数量少
            return dfs(apple, dish - 1)
        elif apple == dish:
            return dfs(apple, dish - 1) + 1  # 相等时，等于少一个盘子的时候 + 多一个盘子的时候多1个选择
        elif apple > dish:
            # 空一个盘子 + (不空时必有apple - dish要放在dish个里)
            return dfs(apple, dish - 1) + dfs(apple - dish, dish)

    _apple, _dish = list(map(int, input().split(' ')))
    ret = dfs(_apple, _dish)
    print(ret)
```

T4：求是否能计算为24点

描述

给出4个1-10的数字，通过加减乘除运算，得到数字为24就算胜利,除法指实数除法运算,运算符仅允许出现在两个数字之间,本题对数字选取顺序无要求，但每个数字仅允许使用一次，且需考虑括号运算
此题允许数字重复，如3 3 4 4为合法输入，此输入一共有两个3，但是每个数字只允许使用一次，则运算过程中两个3都被选取并进行对应的计算操作。

输入描述:

读入4个[1,10]的整数，数字允许重复，测试用例保证无异常数字。

输出描述:

对于每组案例，输出一行表示能否得到24点，能输出true，不能输出false

示例1

输入：7 2 1 10 复制
输出：true 复制

关键点

由于不限顺序，所以要枚举所有顺序组合；

递归一种组合n，和当前的值value，以及递归到n的位置i，依次递归加减乘除4种情况，如果i到了最后且不等于24则返回False，否则返回True。

注意存在(1+1) x (10+2)的情况，需要单独递归。

解法

```python
from itertools import permutations


def fun():
    num = list(map(int, input().split(' ')))

    def dfs(n, value, i):
        if i == 3:  # 当前列表已经递归完
            return True if value == 24 else False  # 当前递归计算的值是不是24
        # 递归当前列表的每一种运算
```

```
        if dfs(n, value + n[i + 1], i + 1) or dfs(n, value - n[i + 1], i + 1) or dfs(n, value * n[i + 1], i + 1) or dfs(n, value / n[i + 1], i + 1):
            return True
        elif i == 1 and (dfs(n, value * (n[i+1] + n[i+2]), i+2) or dfs(n, value * (n[i+1] - n[i+2]), i+2)):  # 后两位是括号
            return True
        elif i == 1 and ((n[i+1] - n[i+2] != 0 and dfs(n, value / (n[i+1] - n[i+2]), i+2)) or dfs(n, value / (n[i+1] + n[i+2]), i+2)):  # 后两位是括号
            return True
        return False
    # 循环输入的4位数的排列
    for _n in list(permutations(num)):
        if dfs(_n, _n[0], 0):  # 只要有一种满足则停止
            print('true')
            return
    print('false')


fun()
```

T5：字符串的通配符

描述

问题描述：在计算机中，通配符一种特殊语法，广泛应用于文件搜索、数据库、正则表达式等领域。现要求各位实现字符串通配符的算法。
要求：
实现如下2个通配符：
*：匹配0个或以上的字符（注：能被*和?匹配的字符仅由英文字母和数字0到9组成，下同）
？：匹配1个字符
注意：匹配时不区分大小写。

输入：
通配符表达式；
一组字符串。
输出：
返回不区分大小写的匹配结果，匹配成功输出true，匹配失败输出false
数据范围：字符串长度：$1 \leq s \leq 100$
进阶：时间复杂度：$O(n^2)$，空间复杂度：$O(n)$

输入描述：

先输入一个带有通配符的字符串，再输入一个需要匹配的字符串

输出描述：

返回不区分大小写的匹配结果，匹配成功输出true，匹配失败输出false

示例1

输入：
```
te?t*.*
txt12.xls
```
复制

输出：
```
false
```
复制

示例2

输入：z
zz 复制

输出：false 复制

示例3

输入：pq
pppq 复制

输出：false 复制

示例4

输入：**z
0QZz 复制

输出：true 复制

示例5

输入：?*Bc*?
abcd 复制

输出：true 复制

示例6

输入：h*?*a
h#a 复制

输出：false 复制

说明：根据题目描述可知能被*和?匹配的字符仅由英文字母和数字0到9组成，所以?不能匹配#，故输出false

示例7

输入：p*p*qp**pq*p**p***ppq 复制
pppppppqppqqppqppppqqqppqppqpqqqppqpqpppqpppqpqqqpqqp

输出：false 复制

关键点

递归两个字符串s和rule:

1、如果两者0位相等则递归s[1:]和rule[1:];

2、否则如果两者0位不相等但是rule是?且s是数字字母，则双方递归下一位;

3、否则如果0位rule是*则分为递归s匹配和s不匹配的情况。s匹配则递归s下一位，rule不动；s不匹配则递归s不动rule下一位。

4、双方都刚好结束返回True，或者s结束rule只剩下*也返回True，否则都是False。

解法

```python
def fun2():
    def dfs(s: str, rule: str):
        if (s, rule) in visit:  # 当前组合已经访问过了就抬走
            return False
        visit.append((s, rule))
        if s == '' and rule == '':  # s完毕且rule刚好完毕，ok
            return True
        elif s != '' and rule == '':  # s未完毕，rule完毕，gg
            return False
        elif s == '' and rule != '':  # s完毕，rule未完毕，看rule是否只剩下*
            if rule.replace('*', '') == '':  # 只剩下*，ok
                return True
            else:  # 还有其他rule值，gg
```

```
                return False
        # 直接就相等抬走下一位
        if s[0] == rule[0]:
            return dfs(s[1:], rule[1:])
        else:
            # 是？的时候必须要是数字字母才进入匹配
            if rule[0] == '?' and s[0].isalnum():
                return dfs(s[1:], rule[1:])
            # 是*的时候分进入和跳过
            elif rule[0] == '*':
                # * 匹配，s往后移
                if s[0].isalnum() and dfs(s[1:], rule):
                    return True
                # * 不匹配， rule往后移
                elif dfs(s, rule[1:]):
                    return True
                else:
                    return False

    rule, s, visit = input().lower(), input().lower(), []
    print('true' if dfs(s, rule) else 'false')

fun2()
```

T6：火车进站

描述

给定一个正整数N代表火车数量，0<N<10，接下来输入火车入站的序列，一共N辆火车，每辆火车以数字1-9编号，火车站只有一个方向进出，同时停靠在火车站的列车中，只有后进站的出站了，先进站的才能出站。
要求输出所有火车出站的方案，以字典序排序输出。
数据范围：$1 \leq n \leq 10$
进阶：时间复杂度：$O(n!)$，空间复杂度：$O(n)$

输入描述：

第一行输入一个正整数N（0 < N <= 10），第二行包括N个正整数，范围为1到10。

输出描述：

输出以字典序从小到大排序的火车出站序列号，每个编号以空格隔开，每个输出序列换行，具体见sample。

示例1

输入：3 复制
1 2 3

输出：1 2 3 复制
1 3 2
2 1 3
2 3 1
3 2 1

说明：第一种方案：1进、1出、2进、2出、3进、3出
第二种方案：1进、1出、2进、3进、3出、2出
第三种方案：1进、2进、2出、1出、3进、3出
第四种方案：1进、2进、2出、3进、3出、1出
第五种方案：1进、2进、3进、3出、2出、1出
请注意，[3,1,2]这个序列是不可能实现的。

关键点

进站的记为一个列表left，站外的记为一个列表right，出站了的记为一个列表当前递归方案res_temp，总的方案数量res。

递归3个列表，分为3种情况：

1、left是空，则只能进站，站外的进站：right=right[1:]。

2、right是空，则只能出站，进站的出站left=left[1:]，res_temp记录出站的车。

3、其他情况可进可出，分别进行递归。

4、边界条件，站内没有了，站外也没有了，此时将res_temp记录到总数据。

解法

```
def fun():
    def dfs(left: list, right: list, res_temp: list):
        # left代表站内的，right代表站外的,res_temp代表出战的
        if not right and not left:
            return res_list.append(res_temp)
        if not left:  # 没进过只能进
            dfs(left + [right[0]], right[1:], res_temp)
        elif not right:  # 进完了只能出
            dfs(left[:-1], right, res_temp + [left[-1]])
        else:
            dfs(left + [right[0]], right[1:], res_temp)  # 想进就进
            dfs(left[:-1], right, res_temp + [left[-1]])  # 想出就出

    n, train, res_list = input(), input().split(' '), []
    dfs([], train, [])
    [print(' '.join(i)) for i in sorted(res_list)]


fun()
```

T7：数组分组

描述

输入int型数组，询问该数组能否分成两组，使得两组中各元素加起来的和相等，并且，所有5的倍数必须在其中一个组中，所有3的倍数在另一个组中（不包括5的倍数），不是5的倍数也不是3的倍数能放在任意一组，可以将数组分为空数组，能满足以上条件，输出true；不满足时输出false。

数据范围：每个数组大小满足 $1 \leq n \leq 50$ ，输入的数据大小满足 $|val| \leq 500$

输入描述：

第一行是数据个数，第二行是输入的数据

输出描述：

返回true或者false

示例1

输入：4
1 5 -5 1 复制

输出：true 复制

说明：第一组：5 -5 1
第二组：1

示例2

输入：3
3 5 8 复制

输出：false 复制

说明：由于3和5不能放在同一组，所以不存在一种分法。

关键点

将数据分为3组，被5整除的arr5，被3整除的arr3，和其他数据放在arr_other。

递归arr3和arr5和arr_other的第i位，递归将第i位加到arr3或者arr5，直到递归完，边界条件i==len(arr_other)，判断arr3是否==arr5，等于则返回True，否则False。

解法

```
def fun():
    def dfs(_arr_3, _arr_5, i):
        if i == len(arr_other):  # 结算
            return True if sum(_arr_3) == sum(_arr_5) else False
        if dfs(_arr_3 + [arr_other[i]], _arr_5, i + 1) or dfs(_arr_3, _arr_5 + [arr_other[i]], i + 1):
            return True
        else:
            return False

    n, arr = input(), list(map(int, input().strip().split(' ')))
    arr_3, arr_5, arr_other = [i for i in arr if i % 3 == 0 and i % 5 != 0], [i for i in arr if i % 5 == 0], \
        [i for i in arr if i % 3 != 0 and i % 5 != 0]
    print('true' if dfs(arr_3, arr_5, 0) else 'false')
```

T8：机器人的最大活动区域

现有一个机器人，可放置于 M × N的网格中任意位置，每个网格包含一个非负整数编号。当相邻网格的数字编号差值的绝对值小于等于 1 时，机器人可在网格间移动

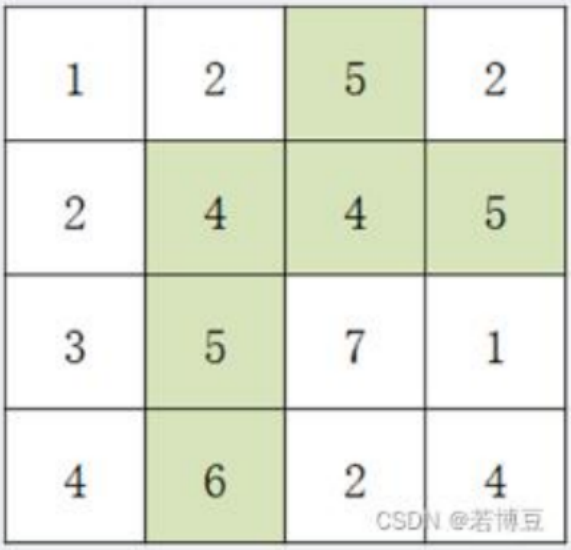
问题：求机器人可活动的最大范围对应的网格点数目。

说明：

1）网格左上角坐标为 (0, 0)，右下角坐标为 (m-1, n-1)

2）机器人只能在相邻网格间上、下、左、右移动

示例1，输入如下网格



输出：6

说明：图中绿色区域，相邻网格差值绝对值都小于等于1，且为最大区域，对应网格点数目为6

```
m, n = 4, 4  # 区域总大小
maze = [[1, 2, 5, 2], [2, 4, 4, 5], [3, 5, 7, 1], [4, 6, 2, 4]]  # 网格值
```

关键点

遍历每一个网格，从网格出发，递归每一个方向，如果相邻方向值大于1则返回False。

解法

```
def fun1():
    def dfs(i, j, value):
        nonlocal step
        if abs(maze[i][j] - value) > 1 or [i, j] in step_visit:
            return False
        else:
            step_visit.append([i, j])
            step += 1
        if i - 1 >= 0:  # 向上
            dfs(i - 1, j, maze[i][j])
```

```
        if i + 1 < m:  # 向下
            dfs(i + 1, j, maze[i][j])
        if j - 1 >= 0:  # 向左
            dfs(i, j - 1, maze[i][j])
        if j + 1 < n:  # 向右
            dfs(i, j + 1, maze[i][j])

    max_step = 1
    for i in range(m):
        for j in range(n):
            step = 0  # 本次能活动的最大的区域
            step_visit = []  # 已访问的活动区域
            dfs(i, j, maze[i][j])
            max_step = max(max_step, step)
    print(max_step)


fun1()
```

3. 栈

T1：矩阵乘法计算量计算

描述

矩阵乘法的运算量与矩阵乘法的顺序强相关。
例如：
A是一个50×10的矩阵，B是10×20的矩阵，C是20×5的矩阵
计算A*B*C有两种顺序：((AB)C)或者(A(BC))，前者需要计算15000次乘法，后者只需要3500次。
编写程序计算不同的计算顺序需要进行的乘法次数。

数据范围：矩阵个数：$1 \leq n \leq 15$，行列数：$1 \leq row_i, col_i \leq 100$，**保证给出的字符串表示的计算顺序唯一。**
进阶：时间复杂度：$O(n)$，空间复杂度：$O(n)$

输入描述：

输入多行，先输入要计算乘法的矩阵个数n，每个矩阵的行数，列数，总共2n的数，最后输入要计算的法则
计算的法则为一个字符串，仅由左右括号和大写字母（'A'~'Z'）组成，保证括号是匹配的且输入合法！

输出描述：

输出需要进行的乘法次数

示例1

输入：
```
3
50 10
10 20
20 5
(A(BC))
```
复制

输出：
```
3500
```
复制

关键点

遍历计算法则字符串，如果不是右括号则入栈，遇到右括号则出栈栈顶元素2个，进行计算后再入栈。

解法

```
def fun():
    n = int(input())
    arr = [list(map(int, input().split())) for i in range(n)]
    order, res = [], 0
```

```
    f = input()
    for i in f:
        if i.isalpha():
            order.append(arr[ord(i) - 65])  # 将字母转换成第几个矩阵的处理信息
        elif i == ')' and len(order) >= 2:  # 读到右括号就处理最近的两个矩阵相乘的结果
            b = order.pop()
            a = order.pop()
            res += a[1] * b[1] * a[0]  # 累计到res中
            order.append([a[0], b[1]])
    print(res)


fun()
```

T2：求去除重复数字能保留下的最大数字

给定一个数字字符串，求它去除重复数字之后能保留下来的最大值。

输入：

5445795045

输出：

79504

关键点

用一个栈来保存合法的数字，栈空的时候进栈，非空的时候判断栈内元素是否合法，不合法则出栈。

用一个长度为10的列表，提前计算记录数字0~9出现的次数。

1、遍历输入，当前元素i，如果栈内i的数量已经有1个了，则跳过。

2、如果栈是空，则入栈。

3、如果栈非空，则结算栈内元素是否合法。循环，判断当前栈顶元素是否比i小且数量大于1，是则出栈，否则代表栈内元素是合法的，则把i入栈。

解法

```
def fun3():
    n = '9855408780'
    st = time.time()
    stack = []
    n = list(map(int, n))
    count = [n.count(i) for i in range(0, 10)]
    for i in n:
        if stack.count(i) == 1:  # 当前元素已经在栈内了,栈内已经容纳满了该元素
            count[i] -= 1
        elif stack:  # 栈非空则判断栈顶是否合法
            while True:  # '5445795045'
                # 栈顶的元素，如果比当前元素小并且有重复并且当前元素不在栈内，则需要出栈
                if stack and stack[-1] <= i and count[stack[-1]] >1:
                    out = stack.pop()
                    count[out] -= 1
```

```
                else:
                    # 如果当前元素不在栈内则入栈
                    stack.append(i)
                    break
        else:
            stack.append(i)  # 栈是空则入栈
    return ''.join(map(str, stack)), time.time() - st
```

T3：求解压缩的值

现需要实现一种算法，能将一组压缩字符串还原成原始字符串，还原规则如下：
1、字符后面加数字N，表示重复字符N次。例如：压缩内容为A3，表示原始字符串为AAA。
2、花括号中的字符串加数字N，表示花括号中的字符串重复N次。例如：压缩内容为{AB}3，表示原始字符串为ABABAB。
3、字符加N和花括号后面加N，支持任意的嵌套，包括互相嵌套。例如：压缩内容可以{A3B1{C}3}3。

输入：

{A3B1{C}3}3

输出：

AAABCCCAAABCCCAAABCCC

关键点

如果不是"}"则入栈，是数字，则将栈顶元素出栈翻倍再入栈。

是"}"则结算，结算时弹出栈顶，临时字符串记录累加，直到栈顶弹出为"{"停止结算。再将临时字符串入栈。

解法

```
def fun():
    s = '{A3B1{CD}3}3'
    stack = []
    for idx, i in enumerate(s):
        # 结算完，判断结算后右括号后有没有触发翻倍
        if i != '}':
            if i.isdigit():  # 是数字则将最后一个入栈的出栈，然后翻倍再入栈
                stack.append(stack.pop() * int(i))
            else:
                stack.append(i)
        else:
            sub_res = ''
            # 遇到右括号进入括号结算
            while True:
                last = stack.pop()  # 出栈
                sub_res = last + sub_res
                if last == '{':
                    break
```

```
            stack.append(sub_res)
    print(''.join(stack).replace('{', ''))


fun()
```

4. 数学

T1：质数因子

描述

功能:输入一个正整数，按照从小到大的顺序输出它的所有质因子（重复的也要列举）（如180的质因子为2 2 3 3 5 ）

数据范围：$1 \leq n \leq 2 \times 10^9 + 14$

输入描述:

输入一个整数

输出描述:

按照从小到大的顺序输出它的所有质数的因子，以空格隔开。

示例1

输入：180 复制

输出：2 2 3 3 5 复制

关键点：

求一个数的因子，只用求到它的根号+1即可。

解法1：

```
def fun1():
    def is_prime(x):
        if x == 1:
            return False
        for _i in range(2, 1 + int(x ** 0.5)):
            if x % _i == 0:
                return False
        return True
    n = int(input())
    res = []
    i = 2
    while i < 1 + int(n ** 0.5):
        # 如果是因子，且是质数，则不断除以此数
        if n % i == 0:  # and is_prime(i):  # 由于n已经对比i小的所有数取余过了，所以如果能对i取余为0，则i必是质数
            while n % i == 0:
                n //= i
                res.append(str(i))
        i += 1
    if is_prime(n):
        res.append(str(n))
    print(' '.join(res))
```

解法

```python
def fun2():
    n = int(input())
    if 1 < n <= 3:
        print(n)
    i = 2
    while i < int(n ** 0.5) + 1:
        while n % i == 0:  # 由于n已经对比i小的所有数取余过了，所以如果能对i取余为0，则i必是质数
            n = n // i
            print(i, end=' ')
        i += 1
if n > 1:
    print(n)
```

T2：求解立方根

描述

计算一个浮点数的立方根，不使用库函数。
保留一位小数。

数据范围：$|val| \leq 20$

输入描述：

待求解参数，为double类型（一个实数）

输出描述：

输出参数的立方根。保留一位小数。

示例1

输入：19.9 复制
输出：2.7 复制

示例2

输入：2.7 复制
输出：1.4 复制

关键点

负数可转为正数处理，结果只需加一个负号；0~1的小数可以取倒数处理，结果只需再取倒数；

题目即可变为求1~inf的浮点数的立方根，使用二分法即可。

解法

```python
def fun():
    n = float(input())
    if n == 0:
        return print(0.0)
    # 负数结果和正数一样，转换为正数
    flag = -1 if n < 0 else 1
    n = flag * n
    # 小于1的取倒数转换为大于1的数来处理，结果再取倒数倒回来
    flag2 = '/' if 0 < n < 1 else ''
    n = 1 / n if 0 < n < 1 else n

    # 由于没有负数和小于1的数，答案应在的范围大于1
```

```
    ans_range = (1, n)
    # 默认答案从二分中间开始
    ans = (ans_range[0] + ans_range[1]) / 2
    while abs(ans * ans * ans - n) > 0.0001:
        # 大于时更新最大值，小于时更新最小值
        ans_range = [ans_range[0], ans] if ans * ans * ans > n else [ans, ans_range[1]]
        ans = (ans_range[0] + ans_range[1]) / 2

    # 是不是0到1之间的数，倒数回来
    if flag2 == '/':
        print(f'{1 / ans * flag:.1f}')
    else:
        print(f'{ans * flag:.1f}')


fun()
```

T3：求解最小公倍数

描述

正整数A和正整数B 的最小公倍数是指 能被A和B整除的最小的正整数值，设计一个算法，求输入A和B的最小公倍数。

数据范围：$1 \leq a, b \leq 100000$

输入描述：

输入两个正整数A和B。

输出描述：

输出A和B的最小公倍数。

示例1

输入：5 7 复制

输出：35 复制

示例2

输入：2 4 复制

输出：4 复制

关键点

暴力遍历即可，注意以下3点可以大幅减少时间复杂度：

1、a、b的最小公倍数最大是a*b，只需遍历到a*b.

2、a、b的最小公倍数一定是a的倍数也一定是b的倍数，用range遍历时步长可选a或者b。

3、选取a、b的较大值作为步长。

T4：素数筛

求小于等于n的素数的个数。

输入n，为正整数。

关键点

遍历到n，是素数则累加，此法时间复杂度太高。

优化：

1、只需要遍历到根号n。

2、创建长度为n+1的列表prime，索引i代表数字i是否是素数。

3、如果i是素数，则从i开始，以i为步长的索引的值，都标记为不是素数。

解法

```
def fun(n):
    prime = [1] * (n + 1)
    prime[0] = prime[1] = 0  # 0和1不是质数
    i = 2
    while i * i <= n:
        if prime[i]:  # 从2开始，2是质数，将2的倍数都标记为非质数。
            prime[i * i:n + 1:i] = [0] * len(prime[i * i:n + 1:i])
        i += 1
    print(sum(prime))

fun(1000)
```

进阶

```
def fun(n):
    prime = bytearray([1] * (n + 1))
    prime[0] = prime[1] = 0  # 0和1不是质数
    i = 2
    while i * i <= n:
        if prime[i]:  # 从2开始，2是质数，将2的倍数都标记为非质数。
            prime[i * i:n + 1:i] = bytearray((n - i * i) // i + 1)
        i += 1
    print(prime.count(1))

fun(1000)
```

5. 区间合并

T1：最多等且不相交连续子序列

求一个数组中，和相等的，连续的子序列中，不相交的子序列的最多的个数。

输入：

8 8 9 1 9 6 3 9 1 0

输出：

4

解析：最多的为和为9的子序列有9和9和3 6和9.

关键点

因为要不相交，所以必须记录每个子序列的区间，且需要和最多，所以要记录和的值。可用一个字典，key为和的值，value为组成和的值的区间，最后再求不相交区间的最多区间的值即可。

1、从0索引遍历数组，位置为i，再遍历i到数组末尾，求得所有的子序列的和及其区间，存储到字典。

2、求不相交的区间的最多区间，只需遍历每个和，求出每个和的每个区间不相交的区间数量。

3、求某个和的区间里不相交的区间的数量，只需把区间按区间结束索引排序，再遍历每个区间，下一个区间的开始索引小于上一个区间的结束索引则相交。

解法

```python
def find_max_equal_sum_subsequences(arr):
    # 创建一个字典来存储子序列的和及其对应的索引
    sum_to_subsequence_indices = {}

    # 遍历数组中的每个元素
    for i in range(len(arr)):
        sum = 0
        # 计算所有可能的连续子序列的和
        for j in range(i, len(arr)):
            sum += arr[j]
            # 如果这个和在字典中还不存在，就创建一个新的键值对
            if sum not in sum_to_subsequence_indices:
                sum_to_subsequence_indices[sum] = []
            # 将子序列的开始和结束索引添加到对应的列表中
            sum_to_subsequence_indices[sum].append((i, j))

    max_count = 0
    # 遍历字典中的每个值（即索引列表）
    for indices in sum_to_subsequence_indices.values():
        # 根据子序列的结束索引进行排序
        indices.sort(key=lambda x: x[1])
        count = 1
        end = indices[0][1]
        # 遍历索引列表中的每个元素
        for i in range(1, len(indices)):
            # 如果当前子序列的开始索引大于前一个子序列的结束索引，就增加计数
            if indices[i][0] > end:
                count += 1
                end = indices[i][1]
        # 更新最大计数
        max_count = max(max_count, count)
    return max_count


arr = [-1, 0, 4, -3, 6, 5, -6, 5, -7, -3]
print(find_max_equal_sum_subsequences(arr))
```

6. 双指针

T1：求符合要求的最长子串

给定一个字符串s，找出符合要求的最长子串，子串要求：

1、该子串中任意一个字符最多出现2次。

2、该子串不包含指定某个字符。

输入：

D

ABACA123D

解析：满足题意的子串为BACA123，长度为7.

关键点

用左右两个指针截取子串，移动右指针，当不满足要求1时，移动左指针，满足要求后继续移动右指针。

当不满足要求2时，左右指针都跳到右指针后面。直到右指针到末尾，过程中一直更新最大长度。

*可用字典记录移动过程中字符的数量，减少字符数量计算。左指针移动时，消失的字符对应数量-1，右指针移动时，多出来的字符的数量+1。跳过时，清空字典。

解法

```python
def fun():
    char = 'D'
    s = 'ABACA123DC'
    left, right = 0, 0
    res = 0
    char_count = {}
    while True:
        # 现在的字符
        now = s[right]
        # 现在的字符为题目不允许的字符时，从这个字符后面继续开始
        if now == char:
            left = right = right + 1
            char_count.clear()
            continue
        # 当当前字符的数量超过2时，先将当前左指针的字符的次数减1，再移动左指针
        if char_count.setdefault(now, 1) > 2:
            char_count[s[left]] -= 1
            left += 1
            continue
        # 更新最大值
        now_long = right - left + 1
        if now_long > res:
            res = now_long
        right += 1
        if right < len(s):
            now = s[right]
            char_count[now] = char_count.get(now, 0) + 1
        else:
            break
    print(res)
```

7. 枚举

T1：红黑图染色方案

给定一张红黑图，其中有些节点是相邻的。所有节点可以染为红色或黑色。

要求：不可以有两个相邻的红色节点。问有多少种染色方案。

输入：

n m，表示n个节点和m条边。接下来输出m行，代表连接x节点和y节点的边，意味着x和y相邻。

```
4 3
0 1
0 2
1 2
输出：8
```

关键点

枚举所有可能的染色方案，共2^n种，去除掉不满足条件的即可。

解法

```
def fun():
    n, m = map(int, input().split())
    # 相邻的节点
    adjoin_point = [list(map(int, input().split())) for _ in range(m)]
    # 所有染色的方案总数
    total = 2 ** n
    # 合格方案数
    res = 0
    for i in range(total):
        # 某种染色方案，红为1，黑为0
        i = bin(i)[2:].rjust(n, '0')
        # 遍历相邻节点
        legal = True
        for point in adjoin_point:
            # 相邻的节点都为红色时代表不合题意，退出
            if i[point[0]] == i[point[1]] == '1':
                legal = False
                break
        if legal:
            print(i)
        res += legal
    print(res)
```

进阶

```
# 用二进制来计算提高速度
def fun():
    # 读取节点数和边数
    n, m = map(int, input().split())
```

```
    # 使用列表推导式创建一个空的邻接矩阵
    edges = [list(map(int, input().split())) for _ in range(m)]
    # 计算总的染色方案数
    total = 1 << n
    # 初始化结果为总的染色方案数
    res = total
    # 遍历所有的染色方案
    for i in range(total):
        # 使用位运算快速计算每个节点的颜色
        colors = [(i >> j) & 1 for j in range(n)]
        print(colors)
        # 检查每条边的两个节点是否都被染成红色
        for u, v in edges:
            if colors[u] == colors[v] == 0:
                # 如果两个节点都被染成红色，则该染色方案不符合要求
                res -= 1
                break
    # 输出结果
    print(res)
```