```
进制转化：
bin() oct() hex()
int('1100101010',2)
except EOFError:

import sys# 使用 sys.stdin.read() 读取所有输入
data = sys.stdin.read()  # 读取整个输入流
print(data)

complex(a,b)
a+bj

float(input())
输出：
法一：f"{value:.nf}"  #其中n是希望保留的小数位数
百分数
print(f"Percentage: {percentage:.2%}")  # 输出：85.00%
科学计数法：
print(f"Scientific notation: {large_number:.2e}")  # 输出：1.23e+06

格式化输出：
基本模板：f"字符串 {表达式} 字符串"
 print(f"My name is {name} and I am {age} years old.")
三元运算符：
print(f"Adult status: {'Yes' if age >= 18 else 'No'}")

无空格输出：  print(' '.join(map(str, minStep[i]))) #注意这里应该是字符串形式
保护圈：  maze.append([-1] + [int(_) for _ in input().split()] + [-1])

lst.remove(114)
lst.pop(0)


# 创建字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

d=[(1,2),(5,6),(7,'cgy')]
dict(d)

 #通过键来搜索值
法一：print(my_dict['name'])  # 输出：Alice
法二：print(my_dict.get('name'))  # 输出：Alice
 print(my_dict.get('address', 'Not Found'))  # 输出：Not Found

 #通过值来搜索键
找到所有的键：
法一：keys = [key for key, value in my_dict.items() if value == search_value]
法二：keys = list(filter(lambda key: my_dict[key] == search_value, my_dict))
找到第一个符合条件的键：
key = next((key for key, value in my_dict.items() if value == search_value),
None)

 #添加或更新元素（键值对）：
my_dict['age'] = 26  # 更新
```

```
my_dict['country'] = 'USA'  # 添加

#向字典中某一个键下添加元素：
my_dict = {'key1': [1, 2, 3], 'key2': [4, 5]}
 my_dict['key1'].append(4)

#删除键值对
法一：del my_dict['city']
法二：age = my_dict.pop('age')
 print(age)  # 输出：26

 #遍历字典：
    # 遍历键
    for key in my_dict:
     # 删除 'city' 键值对
    # 遍历值
    for value in my_dict.values():
     print(value)
     # 遍历键值对
    for key, value in my_dict.items():
     print(f"{key}: {value}")

 #字典推导式举例：
numbers = [1, 2, 3, 4, 5]
 squared_dict = {n: n**2 for n in numbers}
 print(squared_dict)
 #字典排序：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))

# 创建集合
my_set = {1, 2, 3, 4, 5}
 another_set = set([3, 4, 5, 6, 7])
 #注意：set() 用来创建集合时，它接受一个可迭代对象（如列表、元组、字符串等），因而这里set() 会自动
从列表中提取元素并创建集合，而不能直接set(3, 4, 5, 6, 7)，因为set()括号里只可以有一个参数，而{}
则不同。
# 添加元素
my_set.add(6)
# 删除元素（不存在元素可抛出错误）
my_set.remove(2)
 # 删除不存在的元素，不会抛出错误
my_set.discard(10)

lambda的使用：
基本模板：lambda arguments: expression  #参数：对参数进行的操作

在字典排序中：
sorted_dict = sorted(my_dict.items(), key=lambda x: x[1])
 #按值升序排序，注意sorted得到的是一个列表！
 #如果想要降序并转化为字典格式如下：
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))

与map结合：
# 对列表中的每个元素进行平方操作
squared_numbers = list(map(lambda x: x ** 2, numbers))
```

（1）
sorted() 返回一个新的排序后的列表，原始的可迭代对象不被修改。可以作用于任何可迭代对象，包括列表、元组、字典等。支持自定义排序规则（通过
key 参数）。用于需要保持原始数据不变的情况，
或需要排序结果为列表的场景。
如按长度排序：（字典中的用法见上一条）
```
words = ["banana", "apple", "pear", "cherry"]
 sorted_words = sorted(words, key=len)
```
对列表中的数组按照数组的第一个数字排序：
```
sorted_dist=sorted(dist,key=lambda x:x[0])
```
（2）
.sort() 是 列表对象的方法，只能对 列表 进行排序，原地修改列表，即 排序会直接修改原列表，返回值为
None 。适用于不需要保留原列表的情况，或者希望节省内存的场景

```
alst=[10,20,30]
blst=[3,2,1]
zipped=sorted(list(zip(blst,alst)),key=lambda x:x[0])
blst,alst=zip(*zipped)
print(alst)
print(blst)
t=list(alst)
print(t)
#输出：（30，20，10）（1，2，3）[1,2,3]
```

从列表中删除元素
法一：my_list = [1, 2, 3, 4, 2, 5]
```
 my_list.remove(2)  # 删除第一个 2
```
法二：my_list = [1, 2, 3, 4, 5]
```
 removed_element = my_list.pop(2)  # 删除索引为 2 的元素 (即 3)
```

```
ord()    chr()

enumerate(lst,start=0)
for index,num in enumerate(lst)
```

3. 迪杰斯特拉算法（最短权值路径）
（1）最短权值路径（现有一个共n个顶点（代表城市）、m条边（代表道路）的无向图（假设顶点编号为从0到n-1），每条边有各自的边权，代表两个城市之间的距离。求从s号城市出发到达t号城市的最短距离。）
```
import heapq
def dijkstra(n, edges, s, t):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))
    pq = [(0, s)]  # (distance, node)
    visited = set()
    distances = [float('inf')] * n
    distances[s] = 0
    while pq:
        dist, node = heapq.heappop(pq)
        if node == t:
            return dist
        if node in visited:
```

```
            continue
        visited.add(node)
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                new_dist = dist + weight
                if new_dist < distances[neighbor]:
                    distances[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))
    return -1
n, m, s, t = map(int, input().split())
edges = [list(map(int, input().split())) for _ in range(m)]
result = dijkstra(n, edges, s, t)
print(result)
```

双dp：土豪购物

```
def max_value(s):
    n=len(s)
    dp1=[0 for _ in range(n)]
    dp2=[0 for _ in range(n)]
    dp1[0]=s[0]
    dp2[0]=s[0]
    for i in range(1,n):
        dp1[i]=max(dp1[i-1]+s[i],s[i])#不放回
        dp2[i]=max(dp2[i-1]+s[i],dp1[i-1],s[i])#放回之前某个，放回现在也就是第i个，从现在开始取
    return max(max(dp1),max(dp2))
s=list(map(int,input().split(',')))
max_num=max_value(s)
print(max_num)
```

类似思路：最大摆动子序列

```
def max_len(n,nums):
 if n==1:
 return 1
 else:
 up=1
 down=1
 for i in range(1,n):
 if nums[i]>nums[i-1]:
 up=down+1
 elif nums[i]<nums[i-1]:
 down=up+1
 return max(up,down)
 n=int(input())
 nums=list(map(int,input().split()))
 print(max_len(n,nums))
```

小偷背包：01背包

```
n,b=map(int,input().split())
nedan=list(map(int,input().split()))
weight=list(map(int,input().split()))
dp=[[0 for _ in range(b+1)] for _ in range(n+1)]
for i in range(n+1):
    for j in range(b+1):
        if weight[i-1]<=j:
            dp[i][j]=max(dp[i-1][j],dp[i-1][j-weight[i-1]]+nedan[i-1])
```

```
print(dp[n][b])
```

完全背包：

```
def complete_knapsack(n, C, weights, values):
    # 初始化 dp 数组，dp[j] 表示容量为 j 时的最大价值
    dp = [0] * (C + 1)

    for i in range(n):
        for j in range(weights[i], C + 1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

    return dp[C]

# 示例
n = 3
C = 10
weights = [2, 3, 5]
values = [1, 2, 3]

max_value = complete_knapsack(n, C, weights, values)
print(max_value)  # 输出: 8
```

欧拉筛

```
# 返回小于r的素数列表
def oula(r):
    # 全部初始化为0
    prime = [0 for i in range(r+1)]
    # 存放素数
    common = []
    for i in range(2, r+1):
        if prime[i] == 0:
            common.append(i)
        for j in common:
            if i*j > r:
                break
            prime[i*j] = 1
            #将重复筛选剔除
            if i % j == 0:
                break
    return common
 prime = oula(20000)
 print(prime)

def oula(a):
    zhishu=[]
    zhishu1=[True]*(a+1)
    for i in range(2,a+1):
        if zhishu1[i]:
            zhishu.append(i)
        for h in zhishu:
            if h*i<=a:
                zhishu1[h*i]=False
    zhishu=set(zhishu)
    return zhishu

import heapq
```

```
heapq.heapify(lst)
heapq.heappush(lst,114)
x=heapq.heappop(lst)
最小元素：lst[0]

双端队列
from collections import deque
pq=deque([(x0,y0)])

3. bisect:用于在已排序的列表中找到插入点，或者对列表进行插入操作，同时保持列表的顺序。常见的应用包
括二分查找（binary search）、在排序列表中插入元素等。
1.bisect_left(arr, x):（bisect_right(arr, x)同理）
返回一个索引，该索引是列表 arr 中插入元素 x 的位置，并且会确保 x 插入后，列表仍然保持升序排列。
如果 x 已经存在于列表中，则返回左边的插入位置（即 x 的第一个位置）。
import bisect
 arr = [1, 3, 4, 10, 12]
 index = bisect.bisect_left(arr, 5)
 print(index)  # 输出 3，因为 5 应该插入在 10 之前，索引 3
 2.insort(arr, x):（insort_left(arr,x) 保持升序，若x已经存在，插入左边，
insort_right(arr,x)同理）
将元素 x 插入到列表 arr 中，并保持列表的顺序。
等效于使用 bisect_right 找到插入位置，然后插入元素。
import bisect
 arr = [1, 3, 4, 10, 12]
 bisect.insort(arr, 5)  # 将 5 插入到正确的位置
print(arr)  # 输出 [1, 3, 4, 5, 10, 12]
二分查找实现：
import bisect
 arr = [1, 3, 4, 10, 12]
 x = 4
 pos = bisect.bisect_left(arr, x)
 if pos < len(arr) and arr[pos] == x:
 print(f"元素 {x} 存在于列表中，位置为 {pos}")
 else:
 print(f"元素 {x} 不在列表中")
区间查找：
import bisect
 intervals = [1, 5, 10, 15, 20]  # 代表的区间为 [1, 5), [5, 10), [10, 15), [15, 20)
 value = 12
 index = bisect.bisect_right(intervals, value)
 print(f"数值 {value} 在区间 {intervals[index-1]} 和 {intervals[index]} 之间")

import itertools
a=[1,2,3]
p=itertools.permutations(a, 2)
for pp in p:
    print(pp)
#输出：(1, 2)(1, 3)(2, 1)(2, 3)(3, 1)(3, 2)
```

下表中变量 a 为 60，b 为 13二进制格式如下：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

运算符	描述	实例
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1。`~x` 类似于 `-x-1`	(~a) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	a >> 2 输出结果 15 ，二进制解释： 0000 1111

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)	如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1。 **Python 3 已废弃，使用 (x>y)-(x<y) 替换**。
exp(x)	返回e的x次幂(e^x),如math.exp(1) 返回2.718281828459045
fabs(x)	以浮点数形式返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数 x 的四舍五入值，如给出 n 值，则代表舍入到小数点后的位数。 **其实准确的说是保留值将保留到离上一位更近的一端。**
sqrt(x)	返回数字x的平方根。

三角函数

Python包括以下三角函数:

函数	描述
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2) ， 返回90.0
radians(x)	将角度转换为弧度

```
import time

for i in range(101):
    print("\r{:3}%".format(i),end=' ')
    time.sleep(0.05)

>>> list=[1,2,3,4]
>>> it = iter(list)    # 创建迭代器对象
>>> print (next(it))   # 输出迭代器的下一个元素
1
>>> print (next(it))
2
>>>

from functools import lru_cache
@lru_cache(maxsize=128)

#### 日期与时间
import calendar, datetime print(calendar.isleap(2020)) # 输出: True
print(datetime.datetime(2023, 10, 5).weekday()) # 输出: 3 (星期四)
```

```
背包问题（Knapsack Problem）是组合优化中的经典问题，通常的目的是从若干个物品中选择一些物品装入背包，使得背包中物品的总价值最大，且物品的总重量不超过背包的承重限制。

背包问题的变种主要有以下几种：0-1背包、完全背包、多重背包，及其二进制优化。

1. 0-1背包问题
定义： 在0-1背包问题中，给定n个物品和一个容量为C的背包。每个物品有一个重量和价值。每个物品只能选择放入背包或不放入背包，不能部分放入或重复放入。

状态转移方程： 设dp[i][j]表示前i个物品放入容量为j的背包所能获得的最大价值。状态转移方程为：

dp[i][j] = dp[i-1][j]：不选择第i个物品。
```

dp[i][j] = max(dp[i][j], dp[i-1][j-w[i]] + v[i])：选择第i个物品。
代码实现：

python
复制代码
```
def knapsack_01(n, C, weights, values):
    dp = [0] * (C + 1)
    for i in range(n):
        for j in range(C, weights[i] - 1, -1):  # 从后往前遍历，避免重复选择
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[C]
```

2. 完全背包问题
定义： 在完全背包问题中，给定n个物品和一个容量为C的背包。每个物品有一个重量和价值，并且每个物品可以选择无限个。

状态转移方程： 设dp[i][j]表示前i个物品放入容量为j的背包所能获得的最大价值。状态转移方程为：

dp[i][j] = dp[i-1][j]：不选择第i个物品。
dp[i][j] = max(dp[i][j], dp[i][j-w[i]] + v[i])：选择第i个物品。
与0-1背包的区别在于，完全背包问题允许物品多次选择，因此状态转移时内层循环应从前向后遍历。

代码实现：

python
复制代码
```
def knapsack_complete(n, C, weights, values):
    dp = [0] * (C + 1)
    for i in range(n):
        for j in range(weights[i], C + 1):  # 完全背包，从前往后遍历
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[C]
```

3. 多重背包问题
定义： 多重背包问题是完全背包问题的一种扩展。每种物品有一个数量限制，物品的数量不能超过指定的最大数量。

状态转移方程： 设dp[i][j]表示前i种物品放入容量为j的背包所能获得的最大价值。多重背包问题的转移方式稍有变化，我们需要考虑每种物品的数量限制。

通常，可以使用“二进制分解”方法来简化多重背包问题，将问题转化为多个0-1背包问题的组合。

代码实现（分解法）：

python
复制代码
```
def knapsack_multiple(n, C, weights, values, counts):
    dp = [0] * (C + 1)
    for i in range(n):
        # 物品i最多能使用counts[i]次
        for k in range(1, counts[i] + 1):
            for j in range(C, weights[i] * k - 1, -1):
                dp[j] = max(dp[j], dp[j - weights[i] * k] + values[i] * k)
    return dp[C]
```

4. 二进制优化（优化多重背包）
定义： 对于多重背包问题，我们可以通过“二进制优化”将其转化为多个0-1背包问题。二进制优化方法的核心思想是将每种物品的数量限制分解为若干个数量为2^k的子问题，这样可以减少状态转移的次数。

二进制优化思想：

将每个物品的数量count[i]分解为若干个2^k个物品。
例如，如果某种物品有3个，我们可以把它分解为1 + 2个物品，并分别处理这些“新物品”。
这种分解方法保证了每次只需要考虑“0个、1个、2个、4个……”物品，而不需要考虑所有物品数量的组合。
代码实现（分解法）：

python
复制代码
```python
def knapsack_binary(n, C, weights, values, counts):
    dp = [0] * (C + 1)
    for i in range(n):
        k = 1
        while k <= counts[i]:  # 二进制分解
            for j in range(C, weights[i] * k - 1, -1):
                dp[j] = max(dp[j], dp[j - weights[i] * k] + values[i] * k)
            counts[i] -= k
            k *= 2
        # 处理剩余部分
        if counts[i] > 0:
            for j in range(C, weights[i] * counts[i] - 1, -1):
                dp[j] = max(dp[j], dp[j - weights[i] * counts[i]] + values[i] *
counts[i])
    return dp[C]
```
总结
0-1背包：每个物品只能选择一次。
完全背包：每个物品可以选择任意次数，物品的选择没有限制。
多重背包：每个物品的选择次数有限制，通常通过“二进制优化”将其转化为多个0-1背包问题。
二进制优化：通过将物品的数量限制分解成二进制数次方的形式，从而优化多重背包问题。

描述

给定一组n种不同面额的硬币，以及要支付的总金额

计算并返回可以凑成总金额所需的 **最少的硬币个数** 。如果没有任何一种硬币组合能组成总金额，返回 -1。

你可以认为每种硬币的数量是无限的。

输入

输入为两行

第一行为两个整数n（1 ≤ n ≤ 100），m（0 ≤ m ≤ 10^6），其中n表示硬币的种类数，m表示要凑的总金额

第二行为n个整数，表示硬币的面值，所有硬币面值均小于m

输出

可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，则输出 -1。

```python
n,m=map(int,input().split())
mianzhilist=list(map(int,input().split()))
dp=[float("inf") for _ in range(m+1)]
dp[0]=0
for i in range(0,m+1):
    for j in range(0,n):
        if i>=mianzhilist[j]:
            dp[i]=min(dp[i],dp[i-mianzhilist[j]]+1)
if dp[m]==float("inf"):
    print(-1)
else:
    print(dp[m])
```

```python
#迪杰斯特拉伪代码
import heapq

def dijkstra(graph, start):
    # 初始化
    n = len(graph)  # 顶点个数
    dist = [float('inf')] * n  # 最短路径数组，初始为无穷大
    dist[start] = 0  # 起点到自身的距离为0
    visited = [False] * n  # 记录顶点是否被访问
    pq = [(0, start)]  # 优先队列，存储(距离, 顶点)

    while pq:
        current_dist, u = heapq.heappop(pq)

        if visited[u]:
            continue  # 如果当前顶点已被访问，跳过

        visited[u] = True  # 标记当前顶点为已访问

        # 松弛操作
        for v, weight in graph[u]:
            if not visited[v] and current_dist + weight < dist[v]:
                dist[v] = current_dist + weight
                heapq.heappush(pq, (dist[v], v))

    return dist
```

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

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

    # 检测负权环
    for u, v, weight in graph:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
```

```python
                print("Graph contains a negative-weight cycle")
                return None

    return dist

# 图的边表示为 (起点, 终点, 权重)
graph = [
    (0, 1, 1),
    (1, 2, 3),
    (0, 2, 4),
    (2, 3, 2),
    (3, 1, -6)
]
V = 4  # 顶点数量
start = 0  # 源点

distances = bellman_ford(graph, start, V)
if distances:
    print("Vertex Distances from Source:", distances)

#输出: Graph contains a negative-weight cycle
```

Floyd-Warshall算法基于**动态规划**的思想:

- 如果顶点 `k` 是路径 `(i → j)` 的中间顶点之一，那么最短路径要么不经过 `k`，要么经过 `k`。
- 使用一个二维数组 `dist[i][j]` 来存储顶点 `i` 到顶点 `j` 的最短路径距离。
- 对每个中间顶点 `k`，更新所有顶点对 `(i, j)` 的最短路径，判断是否经过 `k` 会使路径更短。

状态转移方程:

dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])

- `dist[i][j]`: 表示从顶点 `i` 到顶点 `j` 的当前最短路径。
- `dist[i][k] + dist[k][j]`: 表示从顶点 `i` 到 `j` 经过 `k` 顶点的路径长度。
- 取两者的较小值，更新最短路径。

初始化:

- 如果存在边 `i → j`，设置 `dist[i][j]` 为该边的权重。
- 如果 `i == j`，设置 `dist[i][j] = 0`。
- 如果不存在边 `i → j`，设置 `dist[i][j] = ∞`。

动态规划:

- 对每个顶点 `k`，将其作为中间节点，遍历所有的顶点对 `(i, j)`。
- 如果经过 `k` 可以缩短 `i → j` 的路径，则更新 `dist[i][j]`。

检测负权环:

- 如果存在 `dist[i][i] < 0`，说明图中存在**负权环**。

输出结果:

- 如果没有负权环，输出所有顶点对之间的最短路径。
- 如果有负权环，报告存在负权环。

```
def floyd_warshall(graph):
    V = len(graph)  # 顶点数量
    dist = [[float('inf')] * V for _ in range(V)]

    # 初始化距离矩阵
    for i in range(V):
        for j in range(V):
            if i == j:
                dist[i][j] = 0  # 自己到自己的距离为0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]  # 存在边则设为权重

    # 动态规划核心
    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    # 检测负权环
    for i in range(V):
        if dist[i][i] < 0:
            print("Graph contains a negative-weight cycle")
            return None

    return dist

graph = [
    [0, 3, float('inf')],
    [float('inf'), 0, 1],
    [-2, float('inf'), 0]
]

distances = floyd_warshall(graph)
if distances:
    for row in distances:
        print(row)

输出:
[0, 3, 4]
[float('inf'), 0, 1]
[-2, 1, 0]
```