工具：
int(str,base)进制转换，如 int('101',2)输出 5
for key, value in dict.items()
for index, value in enumerate(list)
dict.get(key, default) 返回指定键的值，如果不存在则返回 default
list(zip(a, b)) 将两个列表中的值一一对应打包
math.pow(m, n) m 的 n 次方
math.log(m, n) 以 n 为底 m 的对数（默认以 e 为底）
math.ceil()
math.floor()
math.gcd()最大公约数
math.lcd()最小公倍数

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

```
from sys import setrecursionlimit
sys.setrecursionlimit(300000)或者 1<<30
```

str.lstrip() / str.rstrip(): 移除字符串左侧/右侧的空白字符。
str.find(sub): 返回子字符串 sub 在字符串中首次出现的索引，如果未找到，则返回-1。
str.replace(old, new): 将字符串中的 old 子字符串替换为 new。
str.startswith(prefix) / str.endswith(suffix): 检查字符串是否以 prefix 开头或以 suffix 结尾。
str.isalpha() / str.isdigit() / str.isalnum(): 检查字符串是否全部由字母/数字/字母和数字组成。

读取空行：

```
x = input()
if not x:
    balabala
```

多行读取：

```
while True:
    try:
        ...
    except EOFError:
        break
```

找到对应值的键：

```
string = "Words are but wind"
word_order = {el: ind+1 for ind, el in enumerate(string.split())}
print(word_order)
```

sort()中可以加入次级优先级：courses.sort(key = lambda x: (x[2],x[0]), reverse = True)
zip（）函数，将两个列表（字符串）各个元素依次连接为元祖后成为新的列表 list（zip（a， b））
用 extend 连接两个列表

数字	格式	输出	描述
3.1415926	{:.2f}	3.14	保留小数点后两位
3.1415926	{:+.2f}	+3.14	带符号保留小数点后两位
-1	{:-.2f}	-1.00	带符号保留小数点后两位
2.71828	{:.0f}	3	不带小数
5	{:0>2d}	05	数字补零 (填充左边, 宽度为2)
5	{:x<4d}	5xxx	数字补x (填充右边, 宽度为4)
10	{:x<4d}	10xx	数字补x (填充右边, 宽度为4)
1000000	{:,}	1,000,000	以逗号分隔的数字格式
0.25	{:.2%}	25.00%	百分比格式
1000000000	{:.2e}	1.00e+09	指数记法
13	{:>10d}	13	右对齐 (默认, 宽度为10)
13	{:<10d}	13	左对齐 (宽度为10)
13	{:^10d}	13	中间对齐 (宽度为10)

11	`'{:b}'.format(11)`
	`'{:d}'.format(11)`
	`'{:o}'.format(11)`
	`'{:x}'.format(11)`
	`'{:#x}'.format(11)`
	`'{:#X}'.format(11)`
		1011
		11
		13
		b
		0xb
		0XB
			进制

^, <, > 分别是居中、左对齐、右对齐，后面带宽度， : 号后面带填充的字符，只能是一个字符，不指定则默认是用空格填充。

+ 表示在正数前显示 +，负数前显示 -； （空格）表示在正数前加空格

b、d、o、x 分别是二进制、十进制、八进制、十六进制。

Python 实现埃氏筛：

python 复制代码

解释

```python
def sieve(n):
    a = [True] * (n + 1)
    a[0], a[1] = False, False  # 0 和 1 不是素数
    for i in range(2, int(n ** 0.5) + 1):
        if a[i]:
            for j in range(i * i, n + 1, i):
                a[j] = False
    return [x for x in range(2, n + 1) if a[x]]
```

calendar

1.calendar.month(年，月)：返回一个月份的日历字符串。它接受年份和月份作为参数，并以多行字符串的形式返回该月份的日历。

2.calendar.calendar(年)：返回一个年份的日历字符串。这个函数生成整个年份的日历，格式化为多行字符串。

3.calendar.monthrange(年，月)：返回两个整数，第一个是该月第一天是周几（0-6 表示周一到周日），第二个是该月的天数。

4.calendar.weekday(年，月，日)：返回给定日期是星期几。0-6 的返回值分别代表星期一到星期日。

5.calendar.isleap(年)：返回一个布尔值，指示指定的年份是否是闰年。

6.calendar.leapdays(年 1，年 2)：返回在指定范围内的闰年数量，不包括第二个年份。

7.calendar.monthcalendar(年，月)：返回一个整数矩阵，表示指定月份的日历。每个子列表表示一个星期；天数为 0 表示该月份此天不在该星期内。

8.calendar.setfirstweekday(星期)：设置日历每周的起始日。默认情况下，第一天是星期一，但可以通过这个函数更改。

9.calendar.firstweekday()：返回当前设置的每周起始日。

counter：计数

```
from collections import Counter
a=['red', 'blue', 'red', 'green', 'blue', 'blue']
a=Counter(a)
Counter()
```

主要功能：可以支持方便、快速的计数，将元素数量统计，然后计数并返回一个字典，键为元素，值为元素个数。

```
from collections import Counter
list1 = ["a", "a", "a", "b", "c", "c", "f", "g", "g", "g", "f"]
dic = Counter(list1)
print(dic)
    #结果:次数是从高到低的
    #Counter({'a': 3, 'g': 3, 'c': 2, 'f': 2, 'b': 1})
print(dict(dic))
    #结果:按字母顺序排序的
    #{'a': 3, 'b': 1, 'c': 2, 'f': 2, 'g': 3}
print(dic.items())    #dic.items()获取字典的 key 和 value
    #结果:按字母顺序排序的
    #dict_items([('a', 3), ('b', 1), ('c', 2), ('f', 2), ('g', 3)])
print(dic.keys())
    #结果:
    #dict_keys(['a', 'b', 'c', 'f', 'g'])
print(dic.values())
    #结果：
    #dict_values([3, 1, 2, 2, 3])
print(sorted(dic.items(), key=lambda s: (-s[1])))
    #结果:按统计次数降序排序
    #[('a', 3), ('g', 3), ('c', 2), ('f', 2), ('b', 1)]
```

most_common()

返回一个列表，包含 counter 中 n 个最大数目的元素，如果忽略 n 或者为 None，most_common()将会返回 counter 中的所有元素，元素有着相同数目的将会选择出现早的元素

```
list1 = ["a", "a", "a", "b", "c", "f", "g", "g", "c", "11", "g", "f", "10", "2"]
print(Counter(list1).most_common(3))
    #结果：[('a', 3), ('g', 3), ('c', 2)]
```

```
    #"c"、"f"调换位置，结果变化
list2 = ["a", "a", "a", "b", "f", "c", "g", "g", "c", "11", "g", "f", "10", "2"]
print(Counter(list2).most_common(3))
    #结果：[('a', 3), ('g', 3), ('f', 2)]
```

update()

从一个可迭代对象（可迭代对象是一个元素序列，而非(key,value)对构成的序列）中或者另一个映射（或 counter）中所有元素相加，是数目相加而非替换它们

```
dic1 = {'a': 3, 'b': 4, 'c': 0, 'd': -2, "e": 0}
dic2 = {'a': 3, 'b': 4, 'c': 0, 'd': 2, "e": -1, "f": 6}
a = Counter(dic1)
print(a)
    #结果:Counter({'b': 4, 'a': 3, 'c': 0, 'e': 0, 'd': -2})
b = Counter(dic2)
print(b)
    #结果:Counter({'f': 6, 'b': 4, 'a': 3, 'd': 2, 'c': 0, 'e': -1})
a.update(b)
print(a)
#结果：Counter({'b': 8, 'a': 6, 'f': 6, 'c': 0, 'd': 0, 'e': -1})
```

subtract()

从一个可迭代对象中或者另一个映射（或 counter）中，元素相减，是数目相减而不是替换它们

```
dic1 = {'a': 3, 'b': 4, 'c': 0, 'd': -2, "e": 0}
dic2 = {'a': 3, 'b': 4, 'c': 0, 'd': 2, "e": -1, "f": 6}
a = Counter(dic1)
print(a)
    #结果：Counter({'b': 4, 'a': 3, 'c': 0, 'e': 0, 'd': -2})
b = Counter(dic2)
print(b)
    #结果：Counter({'f': 6, 'b': 4, 'a': 3, 'd': 2, 'c': 0, 'e': -1})
a.subtract(b)
print(a)
    #结果：Counter({'e': 1, 'a': 0, 'b': 0, 'c': 0, 'd': -4, 'f': -6})
```

permutations：全排列

```
from itertools import permutations as per
elements = [1, 2, 3]
permutations = list(per(elements))
```

combinations：组合

```
from itertools import combinations as com
elements = ['A', 'B', 'C', 'D']# 生成所有长度为 2 的组合
combinations = list(com(elements, 2))
```

bisect

```
import bisect
# 创建一个已排序的列表
sorted_list = [1, 3, 3, 6, 7, 9]
# 使用 bisect_left 查找元素应插入的位置
```

```
insert_index = bisect.bisect_left(sorted_list, 4)
print("Insert at index:", insert_index)
# 使用 insort_left 插入元素并保持有序
bisect.insort_left(sorted_list, 4)
print("Updated list:", sorted_list)
```

dijkstra

这个版本的 Dijkstra 算法使用了一个集合 visited 来记录已经访问过的节点，这样可以避免对同一个节点的重复处理。当我们从优先队列中取出一个节点时，如果这个节点已经在 visited 集合中，那么我们就跳过这个节点，处理下一个节点。这样可以提高算法的效率。

此外，这个版本的 Dijkstra 算法还在找到目标节点 t 时就立即返回结果，而不是等到遍历完所有节点。这是因为 Dijkstra 算法保证了每次从优先队列中取出的节点就是当前距离最短的节点，所以当我们找到目标节点 t 时，就已经找到了从起始节点 s 到 t 的最短路径，无需再继续搜索。

堆中存储的所有节点的距离是递增的，且不会再变小。

```
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
```

走山路

```
import heapq
def dijkstra():
    heap = []
    heapq.heappush(heap, (0, start_x, start_y))
    min_cost = [[float('inf')] * n for _ in range(m)]
    min_cost[start_x][start_y] = 0
    while heap:
        num, x, y = heapq.heappop(heap)
```

```
            if num > min_cost[x][y]:
                continue
            if x == end_x and y == end_y:
                return num
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n and matrix[nx][ny] != '#':
                    cost = num + abs(int(matrix[nx][ny]) - int(matrix[x][y]))
                    if cost < min_cost[nx][ny]:
                        min_cost[nx][ny] = cost
                        heapq.heappush(heap, (cost, nx, ny))
    return 'NO'

m, n, p = map(int, input().split())
matrix = [input().split() for _ in range(m)]
directions = [(1, 0), (0, 1), (-1, 0), (0, -1)]
for _ in range(p):
    start_x, start_y, end_x, end_y = map(int, input().split())
    if matrix[start_x][start_y] == '#' or matrix[end_x][end_y] == '#':
        print('NO')
    else:
        print(dijkstra())
```

冒泡排序

```
def BubbleSort(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

dp

1、k-Tree

双列表 dp

```
    n, k, d = map(int, input().split())
    mod = 10**9 + 7
    dp1 = [1] + [0] * n
    dp2 = [1] + [0] * n
    for i in range(1, n+1):
        for j in range(1, min(i, k) + 1):
            dp1[i] = (dp1[i] + dp1[i-j]) % mod
        for j in range(1, min(d, i + 1)):
            dp2[i] = (dp2[i] + dp2[i-j]) % mod
    print((dp1[n]-dp2[n])%mod)
```