用空格分隔输入多个字符串，key：input().split()

- `split()` 是字符串对象的一个方法，用于将字符串分割成一个列表。
- 默认情况下，`split()` 会根据空白字符（空格、制表符等）分割字符串，如果你给 `split()` 提供参数，例如 `split(',')`，则会根据指定的字符进行分割。

```
# 输入多个字符串，并用空格分隔
input_string = input("请输入字符串（用空格分隔）：")
strings = input_string.split()
print(strings)
```

移除一个字符串右侧的指定字符， .rstrip()

如果希望移除两侧指定字符集合，则使用： .strip()

```
text = "Hello, World!!!"
cleaned_text = text.rstrip('!')
print(cleaned_text)  # 输出：'Hello, World'
```

在Python中，`chr()` 函数的作用是将一个整数（通常是一个Unicode代码点）转换为对应的字符。它接收一个范围在0到1114111之间的整数作为参数（即有效的Unicode代码点），并返回该代码点对应的字符。

语法

```
chr(i)
```

- `i`：一个整数，表示Unicode代码点。

示例

```
# 示例 1: 数字对应的小写字母
print(chr(97))  # 输出: 'a'
# 示例 2: 汉字
print(chr(20320))  # 输出: '你好'
# 示例 3: 大写字母
print(chr(65))  # 输出: 'A'
```

- 当你需要从编码值（如ASCII或Unicode代码点）生成相应字符时，可以使用`chr()`。
- 与 `ord()` 函数结合使用，`ord()` 可以将字符转换为对应的整数。

示例结合 `ord()`

```
# 使用 ord() 获取字符的 Unicode 码点
char = 'A'
code_point = ord(char)
print(code_point)  # 输出: 65
# 再用 chr() 将其转换回字符
print(chr(code_point))  # 输出: 'A'
```

集合

往集合内部添加已经存在的元素不会改变什么，因为集合特点是无序，唯一

以下是一些有关集合的操作的例子：

```
# 创建集合
my_set = {1, 2, 3}
# 添加元素
my_set.add(4)
print(my_set)  # 输出：{1, 2, 3, 4}
# 移除元素
my_set.remove(2)
print(my_set)  # 输出：{1, 3, 4}
# 检查元素是否在集合中
print(3 in my_set)  # 输出：True
print(5 in my_set)  # 输出：False
```

print函数的使用

如果在 `print()` 函数中传入多个变量，Python 会自动用空格隔开它们进行输出。

例如：

```
a = "Hello"
b = 5.0
c = True
print(a, b, c)  # 输出： Hello 5.0 True
```

如果你想改变输出时的分隔符，可以使用 `sep` 参数：

```
print(i, j, k, l, sep=", ")  # 输出： 1, 2, 3, 4
```

math库

Python 的标准库中的 `math` 模块提供了一个方便的函数 `gcd()` 来计算两个数的最大公因数。

```
import math # 引入数学库
a = 24
b = 36
result = math.gcd(a, b)
print(result)  # 输出：12
```

`isqrt()` 是 Python 中 `math` 模块的一个函数，全称是 **integer square root**，用于计算非负整数的**整数平方根**。与常见的 `sqrt()` 函数不同，`isqrt()` 返回的是**向下取整的整数**结果，而不是浮点数。

sort方法（只能对列表使用）

1. **原地排序**：`sort()` 方法会在原列表上进行排序，不会创建新的列表。
2. **默认升序**：默认情况下，`sort()` 方法会将列表按升序排序。
3. 可选参数：

- `reverse`：一个布尔值参数，默认为 `False`，表示升序。如果设置为 `True`，则按降序排序。
- `key`：一个函数，指定一个用于排序的准则。常用于自定义排序规则。

如果想要降序排列，需要使用 `reverse=True` 参数，例如：

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6]
my_list.sort(reverse=True)
print(my_list)
```

```
[9, 6, 5, 4, 3, 2, 1, 1]#输出
```

通过 `key` 参数，指定一个函数来确定排序顺序。例如，对字符串按长度排序：

```
words = ["apple", "banana", "cherry", "date"]
words.sort(key=len)
print(words)
```

```
['date', 'apple', 'banana', 'cherry']#输出
```

注意：`sort()` 方法直接修改原列表，而不返回新的列表。如果想保留原列表，可以使用 `sorted()` 函数，它返回排序后的新列表，而不修改原列表。

值得一提的是

```
ans.sort(key=lambda x: (x[0], -x[1]))
#把列表中的元素列表先按照第一个元素升序，再按照第二个元素降序
```

这里在这个例子中，`data` 是一个包含多个子列表的列表。我们使用 `sorted` 函数，并传递一个 `lambda` 函数作为 `key`，该函数返回每个子列表的第二个元素 `x[1]`。排序完成后，将得到按第二个元素升序排列的新列表 `sorted_data`。

```
# 示例列表
data = [[1, 3], [2, 1], [3, 2]]
# 按每个子列表的第二个元素进行升序排序
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

输出为：

```
[[2, 1], [3, 2], [1, 3]]
```

`indexed_data = list(enumerate(data))`：使用 `enumerate` 函数将数据和它们的原始索引配对，创建一个包含 `(index, value)` 元组的列表。 索引从 0 开始。

```
indexed_data = list(enumerate(data))  # Creates a list of tuples (index, value)
```

如果想求一个 列表 中指定元素的数量，可以使用列表的 `count()` 方法。这个方法会返回列表中某个指定元素出现的次数。

```
my_list = [1, 2, 3, 2, 4, 2, 5]
count_of_2 = my_list.count(2)
print(count_of_2)  # 输出: 3
```

如果知道要移除元素的索引，可以使用 `del` 关键字。

```
my_list = [1, 2, 3, 4]
del my_list[1]  # Removes the element at index 1 (value 2)
print(my_list)  # Output: [1, 3, 4]
```

如何输出一个列表中所有的元素并且相邻元素之间不空格呢？有以下两种方法：

1.join()函数：

```
my_list = ['a', 'b', 'c', 'd']
output = ''.join(my_list)
print(output)#输出abcd
```

2.使用 print() 的 sep 参数：

```
my_list = ['a', 'b', 'c', 'd']
print(*my_list, sep='')#输出abcd
```

循环输入总量未知的数据方法：try，except

代码示例如下：

```
while True:
    try:
        a=input()
        blablabla
    except EOFError:
        break
```

二进制转换

在 Python 中，可以使用内置的 `bin()` 函数来获取一个数的二进制表示。`bin()` 函数会返回一个以 `'0b'` 开头的字符串，表示该数的二进制形式。

```
# 定义整数
num = 10
# 转换为二进制
berjinzhi= bin(num)
# 输出结果
print(berjinzhi)
```

输出结果为：

```
0b1010 #注意这是个字符串
```

保留特定位小数

1. 使用字符串格式化

使用 f-string（Python 3.6+）

```
number = 3.1415926
formatted_number = f"{number:.2f}"  # 保留两位小数
print(formatted_number)  # 输出: 3.14
```

优点：这些方法返回的是字符串，可以更灵活地控制输出格式。

2. 使用 `Decimal` 模块

对于需要高精度的小数计算，可以使用 `decimal` 模块。

```
from decimal import Decimal, ROUND_HALF_UP
number = Decimal('3.1415926')
rounded_number = number.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)  # 保留两位小数
print(rounded_number)  # 输出: 3.14
```

3. 使用 `math` 模块中的 `floor` 或 `ceil`

```
import math
number = 3.1415926
# 保留两位小数并向下取整
floored = math.floor(number * 100) / 100
print(floored)  # 输出: 3.14
# 保留两位小数并向上取整
ceiled = math.ceil(number * 100) / 100
print(ceiled)  # 输出: 3.15
```

`all()` 是 Python 内置的一个函数，**用于检查一个可迭代对象（如列表、元组、集合等）中的所有元素是否都为 True**。如果所有元素都为 True，则返回 `True`；如果有任何一个元素为 `False`，则返回 `False`。

使用格式

```
all(iterable)
```

其中，`iterable` 表示一个可迭代对象，比如列表、元组等。

工作原理

- `all()` 会依次检查 `iterable` 中的每一个元素，只要有一个元素为 `False`，就会立即返回 `False`。
- 如果 `iterable` 为空，`all()` 默认返回 `True`，因为没有元素可以让它返回 `False`。

举例

```
# 示例2：包含 False 元素
list2 = [True, False, True]
print(all(list2))  # 输出：False
# 示例3：空列表
list3 = []
print(all(list3))  # 输出：True
# 示例4：数字列表（非零数字视为 True，零视为 False）
list5 = [1, 0, 3]
print(all(list5))  # 输出：False
```

实用场景

`all()` 在检查条件时非常实用。例如，可以用它来检查一个列表中的所有元素是否都满足特定条件：

```
# 检查一个列表中的所有元素是否都为正数
numbers = [1, 2, 3, 4, 5]
print(all(n > 0 for n in numbers))  # 输出：True
```

总结来说，`all()` 是一个非常方便的函数，用于在一个可迭代对象中进行全量检查，帮助我们快速判断所有元素是否都符合条件。

要将输入的字符串 `matrix = [[1,2,3],[4,5,6],[7,8,9]]` 转化为一个 Python 中的列表，你可以使用 `ast.literal_eval()` 或者 `eval()` 函数来将字符串转换为真实的列表对象。这里推荐使用 `ast.literal_eval()`，因为它更安全，不会执行任何恶意代码。

示例代码如下：

```
import ast
# 输入字符串
matrix_str = '[[1,2,3],[4,5,6],[7,8,9]]'
# 使用 ast.literal_eval 转换字符串为列表
matrix = ast.literal_eval(matrix_str)
print(matrix)
```

输出：

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这段代码将字符串 `matrix_str` 转化为 Python 中的二维列表对象。如果你使用 `eval()`，也会得到类似的结果，但是 `eval()` 的安全性较差，因此一般建议使用 `ast.literal_eval()`。

heapq堆数据结构

它提供了一个 **堆** 数据结构的实现。堆（Heap）是一种完全二叉树（Complete Binary Tree），并且具有以下两个关键特点

在 Python 中，`heapq` 默认实现的是 **最小堆（Min-Heap）**

`heapq`模块提供的功能

1. `heapq.heappush(heap, item)`：
   - 将一个元素 `item` 添加到堆 `heap` 中，保持堆的性质。
2. `heapq.heappop(heap)`：
   - 移除并返回堆中的最小元素（对于最小堆是堆顶元素），同时维持堆的性质。
3. `heapq.heappushpop(heap, item)`：
   - 将元素 `item` 添加到堆中，并弹出并返回堆中的最小元素。这个操作比先执行 `heappush` 再执行 `heappop` 更高效。
4. `heapq.heapify(iterable)`：
   - 将一个可迭代对象转换为堆。这个操作的时间复杂度是 `O(n)`，比逐个 `heappush` 更高效。
5. `heapq.nlargest(n, iterable)`：
   - 返回可迭代对象中最大的 `n` 个元素，按照从大到小的顺序。输出的是列表
6. `heapq.nsmallest(n, iterable)`：
   - 返回可迭代对象中最小的 `n` 个元素，按照从小到大的顺序。输出的是列表

主要特性和优点

- **堆的插入与删除操作效率高**：`heapq` 实现了最小堆，插入（`heappush`）和删除（`heappop`）的时间复杂度是 `O(log n)`，比普通的列表操作要高效。
- **快速访问最小元素**：堆顶元素可以在 `O(1)` 时间复杂度内获取，因为它总是最小的元素（对于最小堆而言）。
- **适用于优先队列**：最小堆非常适合实现 **优先队列**，因为它可以在 `O(log n)` 的时间内获取当前最小的元素，并且可以动态地插入新的元素。`heapq` 是基于元组的第一个元素进行排序的。如果元组中的第一个元素相同，它会继续比较第二个元素，如果还相同，再比较第三个元素。

例子：使用 `heapq` 实现优先队列

假设我们有一组任务，每个任务都有一个优先级，我们希望按照优先级从高到低的顺序处理这些任务。我们可以使用 `heapq` 来实现一个简单的优先队列。

示例 1：使用 `heapq` 实现最小堆优先队列

```
import heapq
# 创建一个空堆
heap = []
# 添加任务（优先级，任务名）
heapq.heappush(heap, (2, 'Task A'))  # 任务A的优先级为2
heapq.heappush(heap, (1, 'Task B'))  # 任务B的优先级为1
heapq.heappush(heap, (3, 'Task C'))  # 任务C的优先级为3
# 查看最小元素（即优先级最高的任务）
print("最先处理的任务:", heapq.heappop(heap))  # 输出: (1, 'Task B')
# 查看接下来最先处理的任务
print("接下来处理的任务:", heapq.heappop(heap))  # 输出: (2, 'Task A')
# 输出剩下的任务
print("剩下的任务:", heap)  # 输出: [(3, 'Task C')]
```

deque双端队列

`deque` 是 Python 中 `collections` 模块下的一个双端队列（Double-Ended Queue）实现，使得从两端进行操作（插入或删除）都具有很高的性能。

相对于**列表（list）**来说，`deque` 主要具有以下几个优势和特性：

---

1. 两端高效操作

- `deque` **在两端插入和删除元素的时间复杂度是 O(1)**，这意味着无论队列中有多少个元素，从头部或尾部进行操作的速度都非常快。
- `list` **在头部插入或删除的时间复杂度是 O(n)**，因为它需要移动所有其他元素以保持索引顺序，而尾部操作可以达到 O(1)。

2. 双端队列的灵活性

`deque` 支持从两端进行操作，包括：

- `append()` 和 `appendleft()`：分别在右端和左端插入元素。
- `pop()` 和 `popleft()`：分别从右端和左端弹出元素。

3. 适合队列和栈操作

- `deque` 可以方便地用作**队列**（FIFO，先进先出）和**栈**（LIFO，后进先出）。
- 使用 `append` 和 `popleft` 实现队列功能。
- 使用 `append` 和 `pop` 实现栈功能。

示例（队列与栈）：

```
dq = deque()
# 队列示例（FIFO）
dq.append(1)
dq.append(2)
dq.append(3)
print(dq.popleft())  # 输出 1
print(dq.popleft())  # 输出 2
# 栈示例（LIFO）
dq.append(1)
dq.append(2)
dq.append(3)
print(dq.pop())      # 输出 3
print(dq.pop())      # 输出 2
```

---

4. 时间复杂度总结

操作	`deque` 时间复杂度	`list` 时间复杂度
右端插入/删除	O(1)	O(1)
左端插入/删除	O(1)	O(n)
随机访问元素	O(n)	O(1)

结论

`deque` 更适合需要频繁在两端进行插入和删除操作的场景，例如：

- 实现队列（FIFO）或栈（LIFO）。
- 需要固定长度队列的场景（使用 `maxlen`）。

而 `list` 更适合需要随机访问元素的情况，因为 `list` 支持 O(1) 的索引访问。

1. 什么是 Timsort?

Timsort 是一种混合排序算法，结合了 **归并排序** 和 **插入排序** 的优点。它由 Python 的核心开发者 **Tim Peters** 在 2002 年为 Python 编程语言设计，并成为 Python 中的内置排序算法。Timsort 也是 Java 7 及以后版本中 `Arrays.sort()` 方法的实现方式。

2. Timsort 的主要特点

- **稳定性**：Timsort 是稳定的排序算法，即相等的元素在排序后保持相对位置不变。这在某些应用场景下非常重要。
- **时间复杂度**：
  - **最坏情况**：O(n log n)
  - **最佳情况**：O(n)（当数据已经部分排序时）
- **空间复杂度**：O(n)
- **自适应性**：Timsort 能够检测输入数据中的有序子序列（称为“运行”），并利用这些有序部分来优化排序过程，从而在实际应用中表现出色。

3. 如何在你的代码中应用 Timsort?

我们可以利用 Python 的内置 `sort()` 方法，结合 `functools.cmp_to_key` 来定义自定义的比较规则。这样，排序过程将由高效的 Timsort 算法处理，显著加快排序速度。

```
from functools import cmp_to_key

# 自定义比较函数
def compare(a, b):
    # 如果 a + b > b + a，a 应该排在 b 前面
    if a + b > b + a:
        return -1  # a 在前
    elif a + b < b + a:
        return 1   # b 在前
    else:
        return 0   # 不变

# 使用内置排序函数进行排序
lst.sort(key=cmp_to_key(compare))
```

解释:

1. **自定义比较函数** `compare(a, b)`:
   - 比较两个字符串 `a` 和 `b`，通过拼接 `a + b` 和 `b + a` 的大小关系决定它们的排序顺序。
   - 如果 `a + b` 大于 `b + a`，则 `a` 应该排在 `b` 前面，返回 `-1`。
   - 如果 `a + b` 小于 `b + a`，则 `b` 应该排在 `a` 前面，返回 `1`。
   - 如果两者相等，返回 `0`，保持原有顺序。
2. **使用** `sorted()` **或** `list.sort()`:
   - `lst.sort(key=cmp_to_key(compare))` 会根据自定义的比较函数对列表 `lst` 进行排序。
   - 这种方法利用了 Timsort 的高效性，将时间复杂度从 $O(n^2)$ 降低到 $O(n \log n)$，大大提升了排序速度。

算法

质数:

欧拉筛（线性筛）

```
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
```

dp

最长公共子序列

```
for i in range(len(A)):
    for j in range(len(B)):
        if A[i] == B[j]:
            dp[i][j] = dp[i-1][j-1]+1
        else:
            dp[i][j] = max(dp[i-1][j],dp[i][j-1])
```

最长单调子序列

```
dp = [1]*n
for i in range(1,n):
    for j in range(i):
        if A[j]<A[i]:
            dp[i] = max(dp[i],dp[j]+1)
ans = sum(dp)
```

背包问题

```
#01背包（每个物品限放1个）
def zero_one_knapsack(W, weights, values, n):
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(W, weights[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[W]

#完全背包（每个物品不限量放置）
def complete_knapsack(W, weights, values, n):
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(weights[i], W + 1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[W]

#多重背包（每个物品限制数量）
def multiple_knapsack_direct(W, weights, values, counts, n):
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(W, weights[i] -1, -1):
            for k in range(1, counts[i] +1):
                if j >= weights[i] * k:
                    dp[j] = max(dp[j], dp[j - weights[i] * k] + values[i] * k)
    return dp[W]
#也可以使用二进制分解
p = [0]*T
for i in range(n):
    all_num = nums[i]
    k = 1
    while all_num>0:
        use_num = min(k,all_num) #处理最后剩不足2的幂的情形
        for t in range(T,use_num*time[i]-1,-1):
            dp[t] = max(dp[t-use_num*time[i]]+use_num*value[i],dp[t])
```

```
        k *= 2
        all_num -= use_nume
```

dfs

```
dx=[2,2,-2,-2,1,-1,1,-1]
dy=[1,-1,1,-1,2,2,-2,-2]
def dfs(chess,x,y,cnt):
    chess[x][y] = 1#先标记
    #也可以引入辅助visited空间orreached集合
    if cnt==m*n-1:
        vnt+=1
    for i in range(8):
        nx=x+dx[i]
        ny=y+dy[i]
        if 0<=nx<=n-1 and 0<=ny<=m-1 and chess[nx][ny]==0:
            dfs(chess,nx,ny,cnt+1)
    chess[x][y]=0#再回溯

#用stack模拟
        while ans:
            z = ans.pop()
            if z[0]==I and z[1]==J:
                c=True
                break
            if land[z[0] - 1][z[1] - 1] < land[I - 1][J - 1]:
                continue
            for i in range(4):
                nx = z[0] + dx[i]
                ny = z[1] + dy[i]
                if 0 <= nx < M and 0 <= ny < N:
                    if land[nx - 1][ny - 1] <h:
                        land[nx - 1][ny - 1] = h
                        ans.append([nx,ny])
```

bfs

```
#双端队列模拟（两座孤岛距离）
sta = deque()
sta.append((i, j))
visited[i][j] = True
    while sta:
        lenth = len(sta)
        for _ in range(lenth):
            ne = sta.popleft()
            if island[ne[0]][ne[1]] == "1":
                island1.append((ne[0], ne[1]))
            for a in range(4):
                nx = ne[0] + dx[a]
                ny = ne[1] + dy[a]
                if 0 <= nx < n and 0 <= ny < m:
                    if visited[nx][ny] ==False and island[nx][ny] == "1":
                        sta.append((nx, ny))
                        visited[nx][ny] = True
```

```
#函数模拟(螃蟹采蘑菇)
directions=[(0,1),(1,0),(-1,0),(0,-1)]
def bfs(sx,sy,maze):
    visited = set()
    reach=deque([(sx,sy)])
    visited.add((sx,sy))
    while reach:
        x,y=reach.popleft()
        if maze[y][x] == 9 or maze[y + ay][x + ax] == 9:
            return "yes"
        for dx,dy in directions:
            nx=x+dx
            ny=y+dy
            if 0<=nx<n-ax and 0<=ny<n-ay and (nx,ny) not in visited and maze[ny][nx]!=1
and maze[ny+ay][nx+ax]!=1:
                visited.add((nx,ny))
                reach.append((nx,ny))
```

Dijkstra

```
#走山路
import heapq
directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]
m,n,p=map(int,input().split())
mount=[]
for _ in range(m):
    mount.append(list(input().split()))
def dijkstra(sx,sy,ex,ey):
    heap=[]
    distinction=[[float("inf") for _ in range(n)] for _ in range(m)]
    #distinction[ny][nx]表示的是点(nx,ny)到点(sx,sy)的最短距离
    distinction[sy][sx]=0#初始化，起点到起点的距离自然为0
    if mount[sy][sx]=="#":
        return "NO"
    heapq.heappush(heap,(0,sx,sy))
    while len(heap)>0:
        d,x,y = heapq.heappop(heap)#弹出堆顶元素（最小）
        if x==ex and y==ey:
            return d
        h=int(mount[y][x])
        for dx,dy in directions:
            nx=x+dx
            ny=y+dy
            if 0<=nx<n and 0<=ny<m and mount[ny][nx]!="#":
                if distinction[ny][nx]>d+abs(int(mount[ny][nx])-h):
                    distinction[ny][nx]=d+abs(int(mount[ny][nx])-h)
                    heapq.heappush(heap,(distinction[ny][nx],nx,ny))
    return "NO"#全部搜完都搜不到终点，说明终点为"#"
for _ in range(p):
    sy,sx,ey,ex=map(int,input().split())
    print(dijkstra(sx,sy,ex,ey))
```

辅助栈

```
#快速堆猪
pig=[]#主栈，存放所有数据
min_stack=[]#辅助栈，存放最小值
while True:
    try:
        a=input()
        if a[:3]=="pus":
            num=int(a[5:])
            pig.append(num)
            if len(min_stack)==0 or min_stack[-1]>=num:#新加入的如果比辅助栈的栈顶小，那么压入
                min_stack.append(pig[-1])
        elif a[:3]=="pop":
            if len(pig)!=0:
                toppig=pig.pop()
                if toppig==min_stack[-1]:
                    min_stack.pop()
        else:
            if len(pig)!=0:
                print(min_stack[-1])
    except EOFError:
        break
```

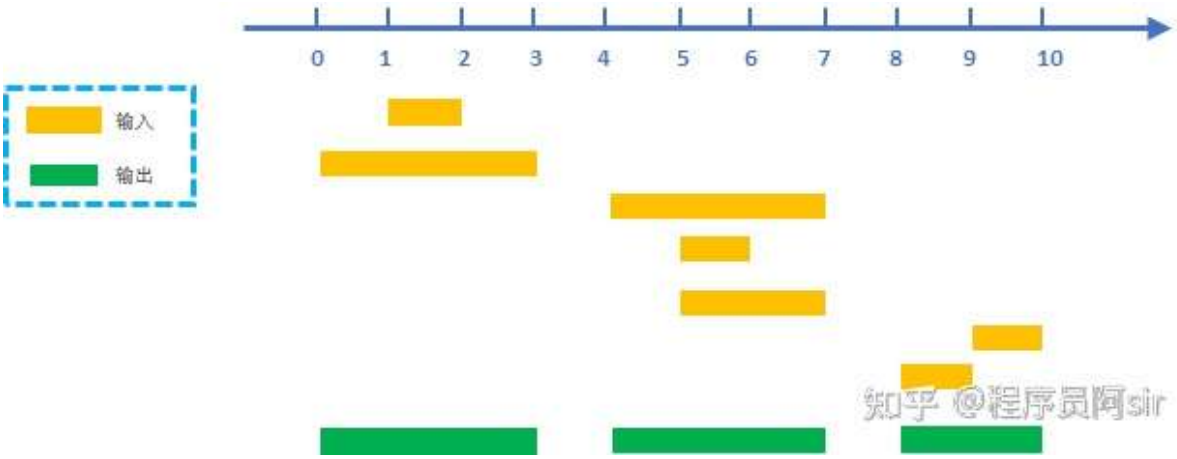
Kadane（最大连续子数组和）

```
def max_subarray_sum(arr):
    if not arr:
        return 0
    max_current=max_global=arr[0]#max_current当前位置的最大子数组和，max_global遍历到目前为止最大子数组和
    for num in arr[1:]:
        max_current =max(num,max_current+num)
        if max_current>max_global:
            max_global= max_current
    return max_global
#最大子矩阵和
步骤 1：初始化
输入：一个二维矩阵 matrix，大小为 rows x cols。
目标：找到和最大的子矩阵。
步骤 2：选择左右列边界
遍历所有可能的列对 (left, right)，其中 left 从 0 到 cols-1，right 从 left 到 cols-1。
对于每一对 (left, right)，将 matrix 中 left 到 right 列之间的所有元素的和压缩为一个一维数组 temp，长度为 rows。
步骤 3：应用Kadane算法
对于每个压缩后的数组 temp，应用Kadane算法找到其最大子数组和 current_max。
如果 current_max 大于当前的全局最大和 max_sum，则更新 max_sum。
同时，记录对应的子矩阵的上下边界（起始行和结束行）以及左右边界。
步骤 4：输出结果
在遍历所有列对后，max_sum 即为最大子矩阵的和。
根据记录的边界，可以提取出对应的子矩阵。
```

区间问题

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。**



区间合并问题示例：合并结果包含3个区间

【步骤一】：按照区间**左端点**从小到大排序。

【步骤二】：维护前面区间中最右边的端点为ed。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。

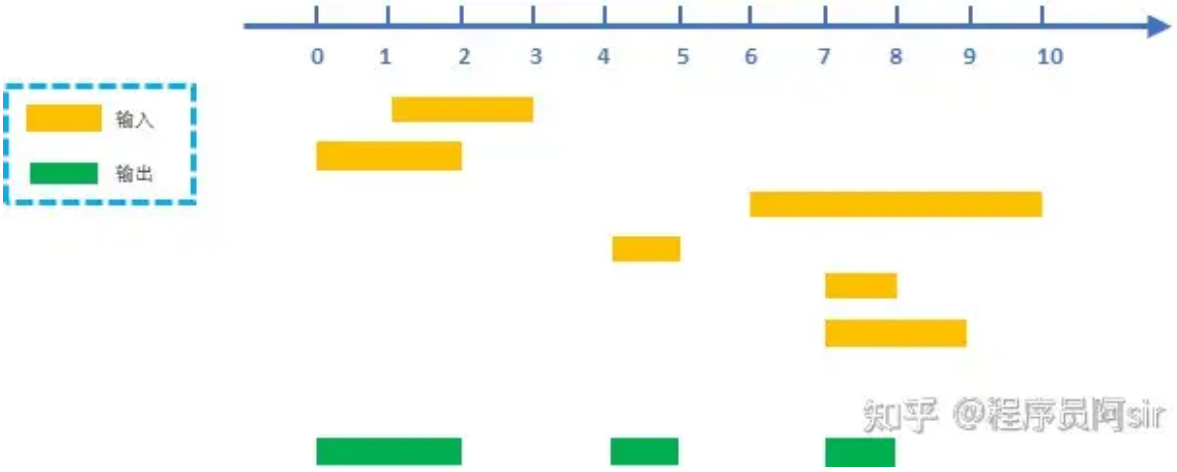
假设当前遍历到的区间为第i个区间 [l_i,r_i]，有以下两种情况：

- l_i <=ed：说明当前区间与前面区间**有交集。**因此**不需要**增加区间个数，但需要设置 ed = max(ed, r_i)。
- l_i > ed: 说明当前区间与前面**没有交集。**因此**需要**增加区间个数，并设置 ed = max(ed, r_i)。

```
list.sort(key=lambda x:x[0])
st=list[0][0]
ed=list[0][1]
ans=[]
for i in range(1,n):
    if list[i][0]<=ed:
        ed=max(ed,list[i][1])
    else:
        ans.append((st,ed))
        st=list[i][0]
        ed=list[i][1]
ans.append((st,ed))
```

2 选择不相交区间

给出一堆区间，要求选择**尽量多**的区间，使得这些区间**互不相交**，求可选取的区间的**最大数量。**这里端点相同也算有重复。



选择不相交区间问题示例：结果包含3个区间

【步骤一】：按照区间**右端点**从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

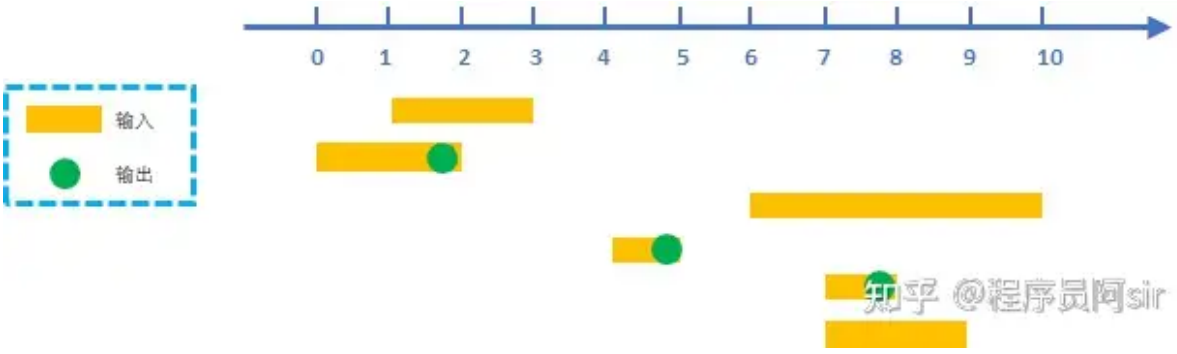
假设当前遍历到的区间为第i个区间 [l_i,r_i]，有以下两种情况：

- l_i <= ed：说明当前区间与前面区间有交集。因此直接跳过。
- l_i > ed: 说明当前区间与前面没有交集。因此选中当前区间，并设置 ed = r_i。

```
list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[list[0]]
for i in range(1,n):
    if list[i][0]<=ed:
        continue
    else:
        ans.append(list[i])
        ed=list[i][1]
```

3 区间选点问题

给出一堆区间，取**尽量少**的点，使得每个区间内**至少有一个点**（不同区间内含的点可以是同一个，位于区间端点上的点也算作区间内）。



区间选点问题示例，最终至少选择3个点

这个题可以转化为上一题的**求最大不相交区间**的数量。

【步骤一】：按照区间右端点从小到大排序。

【步骤二】：从前往后依次枚举每个区间。

假设当前遍历到的区间为第i个区间 [l_i,r_i]，有以下两种情况：

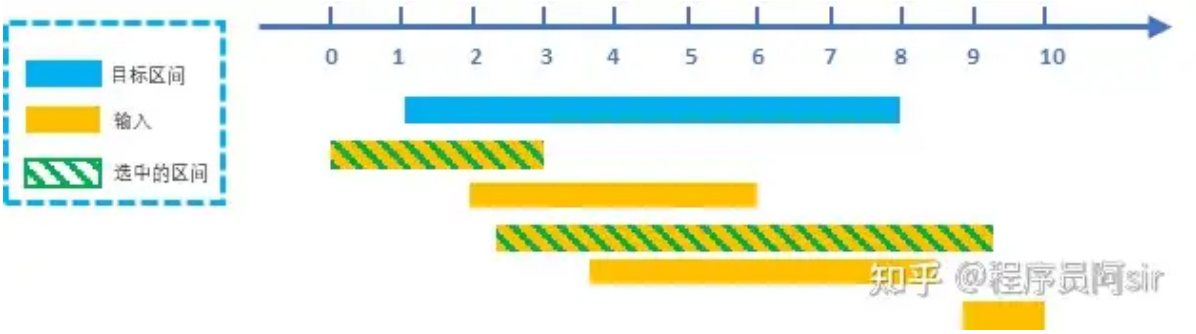
- l_i <=ed：说明当前区间与前面区间有交集，前面已经选点了。因此直接跳过。
- l_i > ed: 说明当前区间与前面没有交集。因此选中当前区间，并设置 ed = r_i。

```
list.sort(key=lambda x:x[1])
ed=list[0][1]
ans=[list[0][1]]
for i in range(1,n):
    if list[i][0]<=ed:
        continue
    else:
        ans.append(list[i][1])
        ed=list[i][1]
```

4 区间覆盖问题

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。

如下图所示：



区间覆盖问题示例，最终至少选择2个区间才能覆盖目标区间

【步骤一】：按照区间左端点从小到大排序。

步骤二】：**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。

假设右端点最大的区间是第i个区间，右端点为 r_i。

最后将目标区间的start更新成r_i

```
q.sort(key=lambda x:x[0])
#start,end 给定
ans=0
ed=q[0][1]
for i in range(n):
    if q[i][0]<=start<=q[i][1]:
        ed=max(ed,q[i][1])
        if ed>=end:
            ans+=1
            break
    else:
        ans+=1
        start=0
        start+=ed
```

5 区间分组问题

给出一堆区间，问最少可以将这些区间分成多少组使得每个组内的区间互不相交。

区间分组问题示例，最少分成3个组

【步骤一】：按照区间左端点从小到大排序。

【步骤二】：从**前往后**依次枚举每个区间，判断当前区间能否被放到某个现有组里面。

（即判断是否存在某个组的右端点在当前区间之中。如果可以，则不能放到这一组）

假设现在已经分了 m 组了，第 k 组最右边的一个点是 r_k，当前区间的范围是 [L_i,R_i] 。则：

如果L_i <r_k 则表示第 i 个区间无法放到第 k 组里面。反之，如果 L_i > r_k， 则表示可以放到第 k 组。

- 如果所有 m 个组里面没有组可以接收当前区间，则当前区间新开一个组，并把自己放进去。
- 如果存在可以接收当前区间的组 k，则将当前区间放进去，并更新当前组的 r_k = R_i。

注意：

为了能快速的找到能够接收当前区间的组，我们可以使用**优先队列 （小顶堆）**。

优先队列里面记录每个组的右端点值，每次可以在 O(1) 的时间拿到右端点中的的最小值。

```
import heapq
list.sort(key=lambda x: x[0])
min_heap = [list[0][1]]
for i in range(1, n):
    if list[i][0] >= min_heap[0]:
        heapq.heappop(min_heap)
    heapq.heappush(min_heap, list[i][1])
num=len(min_heap)
```