2023 年 CCF 大学生计算机系统与程序设计竞赛

CCSP 2023

时间：2023 年 10 月 25 日 09:00 ~ 21:00

题目名称	装修	摸球	次元波动平衡	简易类型系统	可容错实时流处理系统
题目类型	传统型	传统型	传统型	传统型	传统型
输入	标准输入	标准输入	标准输入	标准输入	标准输入
输出	标准输出	标准输出	标准输出	标准输出	标准输出
每个测试点时限	1.0 秒	1.0 秒	1.5 秒	1.0 秒	2.0 秒
内存限制	512 MiB	512 MiB	512 MiB	512 MiB	512 MiB
子任务数目	20	20	6	20	4
测试点是否等分	是	是	否	否	否

装修（decorate）

【题目描述】

在一个热门的元宇宙游戏中，小 b 购置了一座豪华的虚拟房产，现在他需要为这个房子添置若干家具。为了制作这些家具，他需要 N $(1 \leq N \leq 100)$ 种特定的材料。

游戏中总共有 M $(1 \leq M \leq 10^4)$ 种材料，它们的获取分为两类：

1. **直接获取的材料**：这些材料只需要付出相应的成本就可以获得。
2. **通过合成获取的材料**：这些材料无法直接获取，必须通过合成的方式获得。

除了以上两种方式，小 b 还可以通过以下两种途径获取材料：

1. **与其他玩家交换**：小 b 了解到小区内有 P $(0 \leq P \leq 5)$ 个邻居也在准备家具，并且他们有一些多出来的材料。对于邻居 j，他多出来了材料 r_j，并且希望小 b 可以用他所需的材料 s_j 来换取 r_j。小 b 只能与每个邻居交换**至多一次**。
2. **购买官方售卖的材料礼包**：官方售卖的材料礼包共有 Q $(0 \leq Q \leq 5)$ 种，这些礼包包含多种材料，购买礼包将会获得礼包中的所有材料。每个礼包只能购买**至多一次**。

小 b 想要知道，如何以最小的成本来获得所需的 N 种材料。

【输入格式】

第一行包含四个正整数 N, M, P, Q，其中 N $(1 \leq N \leq 100)$ 表示小 b 所需的材料种数，M $(1 \leq M \leq 10^4)$ 表示素材的总种类数，P $(0 \leq P \leq 5)$ 表示邻居的个数，Q $(0 \leq Q \leq 5)$ 表示礼包个数。素材编号为 $1, 2, \cdots, M$。

接下来一行描述了小 b 所需的材料列表，它包含 N 个正整数素材编号。

接下来 M 行描述了每种素材的获取方式。

- 如果这种素材是直接获取的，则输入为 $0, c_i$ $(1 \leq c_i \leq 100)$，其中 c_i 是这种素材的成本。
- 如果这种素材是需要合成的，则第一个数为一个正整数 a_i $(a_i > 0)$，表示合成所需的素材个数。然后紧跟这 a_i 个正整数表示合成所需素材的编号列表。保证**每种素材至多只会作为一种素材的合成材料，并且素材之间的合成关系不会成环**。

之后的 P 行每行有 2 个整数 s_j, r_j $(1 \leq s_j, r_j \leq M)$，表示可以用素材 s_j 交换素材 r_j。注意交换是**单向的**，不可以使用素材 r_j 来交换素材 s_j。

最后的 Q 行每行描述了一个礼包，第一个数为一个正整数 u_k $(u_k \leq 100)$，表示礼包所包含的素材个数，然后是一个正整数 w_k $(1 \leq w_k \leq 10^4)$，表示礼包的成本。之后跟随 u_k 个正整数表示礼包包含的材料编号列表。

【输出格式】

输出一个整数，表示小 b 获取所需材料的最小成本。

【样例 1 输入】

```
1 7 0 0
1
3 2 3 4
3 5 6 7
0 2
0 3
0 5
0 6
0 3
```

【样例 1 输出】

```
19
```

【样例 1 解释】

最优方案为：

- 获取素材 $5,6,7$ 并合成为素材 2，花费 $5+6+3=14$；
- 获取素材 $3,4$，花费 $2+3=5$；
- 使用素材 $2,3,4$ 来合成素材 1。

总成本为 $14+5=19$。

【样例 2 输入】

```
3 6 2 2
1 2 3
2 2 3
1 4
0 2
0 6
0 3
0 8
6 1
3 4
2 9 4 6
```

```
12 2 6 5 6
```

【样例 2 输出】

```
1 10
```

【样例 2 解释】

最优方案为：

- 购买素材礼包 2，获取素材 5,6，花费 6，然后通过素材 6 交换得到素材 1；
- 获取素材 3，交换得到素材 4，然后合成素材 2，花费 2；
- 获取素材 3，花费 2。

总成本为 $6+2+2=10$。

【子任务】

对于所有的数据，满足 $1 \le N \le 100$，$1 \le s_j, r_j \le M \le 10^4$，$0 \le P, Q \le 5$，$1 \le c_i, u_k \le 100$，$1 \le w_k \le 10^4$。

<table>
<tr><th>测试点</th><th>N</th><th>M</th><th>P</th><th>Q</th></tr>
<tr><td>1</td><td>$=10$</td><td>$=20$</td><td rowspan="4">$=0$</td><td rowspan="8">$=0$</td></tr>
<tr><td>2</td><td>$=20$</td><td>$=10^3$</td></tr>
<tr><td>3</td><td>$=50$</td><td>$=2,000$</td></tr>
<tr><td>$4 \sim 6$</td><td>$=10^2$</td><td>$=10^4$</td></tr>
<tr><td>7</td><td>$=10$</td><td>$=20$</td><td rowspan="8">$=5$</td></tr>
<tr><td>8</td><td>$=20$</td><td>$=10^3$</td></tr>
<tr><td>9</td><td>$=50$</td><td>$=2,000$</td></tr>
<tr><td>$10 \sim 12$</td><td>$=10^2$</td><td>$=10^4$</td></tr>
<tr><td>13, 14</td><td>$=10$</td><td>$=20$</td><td rowspan="4">$=5$</td></tr>
<tr><td>15, 16</td><td>$=20$</td><td>$=10^3$</td></tr>
<tr><td>17, 18</td><td>$=50$</td><td>$=2,000$</td></tr>
<tr><td>19, 20</td><td>$=10^2$</td><td>$=10^4$</td></tr>
</table>

摸球（ball）

【题目描述】

小 C 最近迷上了摸球。

小 C 不喜欢重复，因此球有颜色和编号两种属性，只要有一个不同就视为不同的球。

小 C 不喜欢重复，因此他买了 $n+m$ 种颜色的球。其中前 n 种颜色的球各有 a 个，编号为 1 到 a。后 m 种颜色的球各有 b 个，编号为 1 到 b。$(0 \leq n, m, a, b \leq 10^3)$

小 C 不喜欢重复，因此他每次会从中摸出 k 个颜色互不相同的球。

小 C 不喜欢重复，因此他希望被摸出来的球的编号互不相同。

当然，小 C 学过生日悖论，他知道当 k 足够大时，这个概率是很低的。但小 C 还是不喜欢重复，因此他希望知道，给定一个**不超过** 2 的正整数 s，在所有大小为 k 且颜色互不相同的球的集合中，有多少个集合满足任意一个编号的出现次数不超过 s。

这个数字可能很大，你只需要输出答案 $\bmod 998,244,353$ 的结果即可。

【输入格式】

从标准输入读入数据。

第一行包含 6 个正整数 n, m, a, b, k, s，含义如题面所示。$(0 \leq n, m, a, b \leq 10^3$，$0 \leq k \leq n+m$，$s \in \{1, 2\})$

【输出格式】

输出到标准输出。

输出一个非负整数，表示满足条件的集合个数 $\bmod 998,244,353$ 的结果。

【样例 1 输入】

```
1 2 1 2 2 1
```

【样例 1 输出】

```
4
```

【样例 1 解释】

假设球 (x, y) 表示颜色为 x，编号为 y 的球。大小为 2，颜色互不相同，任意一个编号的出现次数不超过 1 的球的集合有 $\{(1,1),(2,2)\},\{(1,1),(3,2)\},\{(2,1),(3,2)\},\{(2,2),(3,1)\}$，一共 4 个。

【样例 2 输入】

```
1 2 1 2 3 2
```

【样例 2 输出】

```
3
```

【样例 2 解释】

大小为 3，颜色互不相同，任意一个编号的出现次数不超过 2 的球的集合有 $\{(1,1),(2,1),(3,2)\},\{(1,1),(2,2),(3,1)\},\{(1,1),(2,2),(3,2)\}$，一共 3 个。

【样例 3 输入】

```
32 64 32 64 16 2
```

【样例 3 输出】

```
643230309
```

【子任务】

对于所有的数据，满足 $0 \le n,m,a,b \le 10^3$，$0 \le k \le n+m$，$s \in \{1,2\}$。

测试点	n,m,a,b	s	特殊性质
$1 \sim 3$	≤ 5	$=1$	无
$4,5$	$\le 10^3$		$a=b$
$6 \sim 10$			无
$11 \sim 13$	≤ 5	$=2$	无
$14,15$	$\le 10^3$		$a=b$
$16 \sim 20$			无

次元波动平衡（surge）

【题目描述】

欢迎来到异次元世界的波动预测局（也被称为“未来视窗”）！在这里，我们有一台神奇的仪器——时间风波镜。它能预测到次元能量指数未来 n 个时间点的波动。这组波动用数学表示为 $A = a_1, a_2, \cdots, a_n$。

不过别高兴得太早，事情并不那么简单。由于暗黑神力的影响，波动可能异常剧烈，甚至有可能引发次元风暴！

但别担心，你作为一名波动调节师（还有着二次元头像的那种），有权在不超过 K 个时间点上使用你的神秘力量——次元调谐器进行干预。次元调谐器能让你选择某个时间点的波动值减去 D。注意，一个时间点**最多**可以被干预 1 次。

你的任务，如果选择接受的话，是通过合理地选择最多 K 个时间点进行干预，从而最小化“**次元波动指数**”，**次元波动指数**是指未来任意时间段的波动之和的最大值。具体来说你需要**最小化**：

$$\max_{1\le i\le j\le n} \sum_{k=i}^{j} a'_k$$

其中 a'_k 是在对波动数组 A 做至多 K 次干预后得到的数组 A' 中的波动值。

【输入格式】

第一行包含三个整数 n、K 和 D，其中 n 表示未来可以预测的时间点数量，K 表示你能进行干预的最大次数，D 是你的次元调谐器的削减量。($1 \le n \le 5\times 10^5$，$0 \le K \le n$，$1 \le D \le 10^9$)

第二行包含 n 个整数 a_i，表示预测到的次元能量指数在未来 n 个时间点的波动。($-10^9 \le a_i \le 10^9$)

【输出格式】

输出一个非负整数，表示通过至多 K 次干预后的最小**次元波动指数**。

【样例 1 输入】

```
5 2 2
1 -2 3 1 1
```

【样例 1 输出】

```
1
```

【样例 1 解释】

选择第 3 跟第 4 个时间点干预，得到干预后的 A' 数组为 $1,-2,1,-1,1$，它的次元波动指数是 1。

【样例 2 输入】

```
12 4 56
1 9 8 -6 1 8 12 10 16 17 -4 -5
```

【样例 2 输出】

```
10
```

【样例 3】

见题目目录下的 ***3.in*** 与 ***3.ans***。

【子任务】

对于所有的数据，满足 $1 \le K \le n \le 5 \times 10^5$，$1 \le D \le 10^9$，$-10^9 \le a_i \le 10^9$。默认数据随机生成，部分子任务数据基于特殊构造。

<table>
<tr><th>子任务编号</th><th>分值</th><th>n</th><th>K</th><th>包含特殊构造</th></tr>
<tr><td>1</td><td>10</td><td>$\le 10^2$</td><td>≤ 3</td><td rowspan="4">否</td></tr>
<tr><td>2</td><td>10</td><td>$\le 10^3$</td><td>$\le 10^3$</td></tr>
<tr><td>3</td><td>20</td><td>$\le 10^4$</td><td>$\le 10^4$</td></tr>
<tr><td>4</td><td>20</td><td rowspan="2">$\le 10^5$</td><td rowspan="2">$\le 10^5$</td></tr>
<tr><td>5</td><td>20</td><td rowspan="2">是</td></tr>
<tr><td>6</td><td>20</td><td>$\le 5 \times 10^5$</td><td>$\le 5 \times 10^5$</td></tr>
</table>

本题中子任务**向前依赖**，即只有当前一个子任务全部通过时，下一个子任务才会开始评测。

简易类型系统（types）

【题目背景】

0x2023 年，CC 国与 SP 国展开了一场激烈的算力竞赛。为了存储计算过程中使用的海量数据，CC 国研发了一款容量为 1 QiB = 2^{100} B（B 为字节，1 字节 = 8 二进制位）的新型内存。已经成为 CC 国工程院院士的小 C 发现 64 位架构已经无法充分利用如此巨大的内存空间，于是他决定研发一款 128 位架构的新系统。

为了与绝大多数现有的 64 位系统兼容，小 C 决定沿用**小端序**作为新系统的字节顺序，即：低位字节存储在低地址上，高位字节存储在高地址上。例如一个占 32 位的无符号整型数 `0xDEADBEEF` 在内存中的存储方式为：

地址	`0x00`	`0x01`	`0x02`	`0x03`
Byte	`0xEF`	`0xBE`	`0xAD`	`0xDE`

作为小 C 的得意门生，你被指定来完成新系统中**类型系统**部分的验证工作。具体来说，你需要完成以下三个任务：

1. **静态类型检查**：计算类型的**对齐字节数**与**占用字节数**，并验证类型系统的完整性。
2. **分配类型实例**：在新型内存上为不同类型的变量分配空间，验证类型系统能够充分利用新型内存。
3. **读写类型实例**：读取、写入已分配的变量，验证类型系统能够在新型内存上正确工作。

由于还在验证阶段，用来测试的指令可能有些问题，你可能还需要处理一些非法或冲突的情况。对于第一个任务，如果出现了问题，你需要汇报第一个导致问题的类型并停止后续执行；对于后两个任务，如果遇到了有问题的指令，你需要汇报出现问题的指令，同时跳过这条指令继续执行后续指令。

由于系统中的类型和用来验证的指令数量都非常庞大，你决定写个程序来模拟这个过程。

【题目描述】

注记：如果你对以下描述中以**加粗下划线**格式标注的概念不熟悉，可以先阅读本题后附加的**名词解释**部分。

本题中，你要在一个最大位宽为 128 的**小端序**设备上实现一个**类型系统**，其中的类型包括：

- **整型数**`{u,i}{8,16,32,64,128}`

包含**无符号**和**有符号**两类，每类又包含 8、16、32、64 与 128 位五种位宽。

其中位宽为 $w = 2^k$ 的**无符号整型数**取值范围为 $[0, 2^k - 1]$，**类型名**为 `u{w}`（例如 `u8`）；位宽为 $w = 2^k$ 的**有符号整型数**使用二进制**补码**表示，取值范围为 $[-2^{k-1}, 2^{k-1} - 1]$，**类型名**为 `i{w}`（例如 `i64`）；位宽为 2^k 的**整型数**对齐到 2^{k-3} 字节。

- **浮点数**`f{16,32,64,128}`

包含 16、32、64 与 128 位四种位宽。

所有**浮点数**均采用 ***IEEE Std 754-2019* 标准**表示，其中位宽为 $w = 2^k$ 的 * 浮点数 ** 类型名 * 为 `f{w}`（例如 `f32`），对齐到 2^{k-3} 字节。

整型数与**浮点数**统称为**基本数据类型**。

- **指针**`T*`

其中 `T` 为任意类型。**指针**类型占用 16 字节，对齐到 16 字节。

- **定长数组**`T[N]`

其中 `T` 为任意类型，特别的，`T` 可以是另一个**定长数组**类型（例如 `i32[4][6]` 是一个合法的类型）；`N` 是一个小于 2^{127} 的正整数，`N`**不会**被存储到内存中，因此也**不会**占用任何空间。规定**定长数组**的下标从 0 开始。若类型 `T` 占用 s 字节，则 `T[N]` 占用 $s \times$ `N` 字节；若类型 `T` 对齐到 2^t 字节，则 `T[N]` 也对齐到 2^t 字节。

- **联合体**`union typename { T1 name1, T2 name2, ..., TN nameN }`
- **结构体**`struct typename { T1 name1, T2 name2, ..., TN nameN }`

其中 `typename` 为**类型名**。**联合体**或**结构体**包含至少一个子项，即**子项数** $N \geq 1$。`T{i}` 为第 i 个子项的**类型**，记其**占用字节数**为 s_i，**对齐字节数**为 a_i；`name{i}` 为第 i 个**子项名**。

联合体与**结构体**的**对齐字节数**均为所有子项**对齐字节数**的最大值，即 $a_{\texttt{union}} = a_{\texttt{struct}} = \max_{i=1}^{N} a_i$。**联合体**内所有子项被**靠前且重叠地**存储在同一段内存中，**占用字节数**为所有子项**占用字节数**的最大值并**向上对齐**到**对齐字节数**，即满足 $s_{\texttt{union}} \geq \max_{i=1}^{N} s_i$ 且 $a_{\texttt{union}} | s_{\texttt{union}}$ 的最小值；**结构体**内各个子项的存储方式及结构体的**占用字节数**按如下方式确定：

- 设第 i 个子项的**偏移字节数**为 o_i，规定首个子项的**偏移字节数** $o_1 = 0$。
- 其后第 i 个子项的**偏移字节数** o_i 为满足 $o_i \geq o_{i-1} + s_{i-1}$ 且 $a_i | o_i$ 的最小值。
- **结构体**的**占用字节数** $s_{\texttt{struct}}$ 为满足 $s_{\texttt{struct}} \geq o_N + s_N$ 且 $a_{\texttt{struct}} | s_{\texttt{struct}}$ 的最小值。

保证本题中出现的所有类型（含中间类型）**占用字节数不超过** 2^{120}。

【任务一：静态类型检查】

给定 n_1 条**联合体**或**结构体**的**声明指令**或**定义指令**。两种指令均为一整行、以 `union` 或 `struct` 开头、以 `';'` 结尾。

声明指令的格式为：

```
{union,struct} typename;
```

其中 `typename` 为任意**标识符**。**标识符**是满足以下条件的字符串：

- 由大小写英文字母、下划线与数字组成，不包含空格或其他字符。
- **不是**`struct`、`alloc` 等**关键字**，因此你无需考虑**标识符**与**关键字**冲突的问题。

在此基础上，规定**合法标识符**还需要满足以下条件：

- 以大小写字母或下划线开头，即不以数字开头。
- 不能与**基本数据类型**的**类型名**冲突。

声明指令的示例如下：

```
union myUnion;
struct myStruct;
```

定义指令的格式为：

```
{union,struct} typename { T1 name1, T2 name2, ..., TN nameN };
```

其中 `typename` 为任意**标识符**；`T{i}` 为任意**类型名**；**子项名**`name{i}` 为任意**标识符**，但不能与其他**子项名**冲突。保证输入数据中 `'{'` 前后、`','` 后、`'}'` 前均有且只有一个空格，最后一个**子项名**后没有 `','`。示例如下：

```
union myUnion { u64 dword, u8[8] bytes };
struct pair { int first, int second };
struct tuple { pair first, int third };
```

在本任务中，对于每条指令，你首先需要检查它是否合法且没有冲突，具体包括：

- **类型名**、**子项名**（保证它们总是**标识符**）是否为**合法标识符**。
- 同一**类型名**是否被**定义**了超过一次。
- 同一**类型名**是否既被**声明**或**定义**为**联合体**又被**声明**或**定义**为**结构体**。
- 在**联合体**或**结构体**内，同一**子项名**出现了超过一次。

保证**不会出现除以上列出情形外的错误**。若在处理第 i（**从 2 开始计数**）条指令时出现以上问题中的任何一种，输出 `syntax error on line {i}`。你无需再进行后续处理，**直接退出程序**即可。

接下来你需要检查是否存在**不完整类型**。**不完整类型**包括且仅包括：

- **声明**但未**定义**的类型。
- 直接或间接包含自身的类型。间接包含需要考虑**定长数组**、**联合体**、**结构体**；**允许**包含指向自身类型的**指针**。
- 包含其他**不完整类型**的类型。

若存在这样的类型 `typename`，输出 `incomplete type {typename}`；若有多个这样的类型，只需要输出**声明指令**或**定义指令最靠前**的那个。你无需再进行后续处理，**直接退出程序**即可。

若没有出现以上问题，你需要计算每种类型的**占用字节数**和**对齐字节数**。按首次**声明**或**定义**的顺序，每种类型输出一行一个字符串与两个整数，分别表示**类型名**、**占用字节数**与**对齐字节数**，以空格分隔。保证**类型的占用字节数不超过** 2^{124}。

【任务二：分配类型实例】

在一段首地址为 0、大小为 2^{100} **字节**的内存空间中，你需要依次处理 n_2 条**分配指令**。每条指令均为一整行、以 `alloc` 开头、以 `';'` 结尾。

分配指令的格式为：

```
alloc T name;
```

其中 `T` 为任意类型，**变量名**`name` 为任意**标识符**，但不能与**类型名**或**已经成功分配的变量名**冲突。示例如下：

```
alloc i32[2][3][4] a;
alloc pair[2] b;
alloc myUnion** c;
```

在本任务中，对于每条指令，你首先要检查它是否合法且没有冲突，具体包括：

- **变量名**（保证为**标识符**）是否为**合法标识符**且不与**类型名**或**已经成功分配的**的变量名冲突。
- **类型**中是否出现了未**定义**的**类型名**。

保证**不会出现除以上列出情形外的错误**。若在处理第 i（**从** $n_1 + 2$ **开始计数**）条指令时出现以上问题中的任何一种，输出 `syntax error on line {i}` 并**忽略这条指令，后续指令仍需处理**。

若没有出现以上问题，你需要为这个变量分配一段连续的内存空间，大小为其类型的**占用字节数**，首地址需要为其类型的**对齐字节数**的倍数。若存在多个满足条件的首地址，你需要选择**最小**的那个。若无法找到满足条件的首地址，输出 `memory allocation failed for {name}` 并**忽略这条指令，后续指令仍需处理**。

【任务三：读写类型数据】

假设**任务二**中的内存空间是零初始化的，你需要依次处理 n_3 条**读取指令**或**写入指令**。每条指令均为一整行、以 `read` 或 `write` 开头、以 `';'` 结尾。

读取指令的格式为：

```
read expr;
```

其中 `expr` 为任意**表达式**。**表达式**由以下规则生成：

- 任何**标识符**都是**表达式**。
- 若 `expr` 是**表达式**且类型**不为地址立即数**，则 `&expr` 也是**表达式**，表示取 `expr` 的地址，类型为**指向** ***expr*** **的类型的地址立即数**。（请注意区分**地址立即数**与**指针**类型：**指针**类型在指向一段内存的同时，自身也被存储在内存中，可以对其进行取地址操作；**地址立即数**是对任意变量取地址得到的地址值，没有存储在内存中，**不能**继续取地址）

- 若 expr 是**表达式**且类型为**地址立即数**或**指针**，则 *expr 也是**表达式**，表示访问其指向的内存，类型为其指向的类型。
- 若 expr 是**表达式**且类型为**定长数组**，则 expr[index] 也是**表达式**，表示访问定长数组中下标为 index 的元素，类型为定长数组的元素类型。保证 index 是一个小于 2^{127} 的非负整数。
- 若 expr 是**表达式**且类型为**结构体**或**联合体**，则 expr.name 也是**表达式**，表示访问结构体或联合体中名为 name 的子项，类型为子项的类型。保证 name 为由大小写英文字母、下划线与数字组成的字符串。
- 若 expr 是**表达式**，则 (expr) 也是**表达式**，用于改变操作优先级，类型不变。
- 所有**表达式**只能由以上规则生成。

运算优先级从高到低依次为：[index] $>$ & $=$ * $>$ .name。示例如下：

```
read &a;
read a[3][1][1];
read b[1].second;
read (**c).bytes[2];
```

写入指令的格式为：

```
write expr = value;
```

其中 expr 为**表达式**，value 为一个整型或浮点常量（格式规定见下文）。示例如下：

```
write a[1][1][2] = 233;
write b[1].second = -0666;
write (**c).bytes[2] = 0xDD;
write some_f32 = -1.2p3;
```

在本任务中，对于每条指令，你首先要检查它是否合法且没有冲突，具体包括：

- **表达式**中的**标识符**是否为**任务二**中成功分配的**变量名**。
- & 的操作对象**不能**是**地址立即数**。
- * 的操作对象**只能**是**地址立即数**或**指针**。
- [index] 的操作对象**只能**是**定长数组**，并且 index 需要小于**定长数组**的大小。
- .name 的操作对象**只能**是**联合体**或**结构体**，并且 name 需要为合法的**子项名**。
- **指针**指向的地址需要是其指向类型的**对齐字节数**的倍数，并且其指向对象**完整地落在内存空间内**。无论是否进行后续操作，任何**指针**类型都需要进行此检查。

保证**不会出现除以上列出情形外的错误**。若在处理第 i（**从** n_1+n_2+2 **开始计数**）条指令时出现以上问题中的任何一种，输出 syntax error on line {i} 并**忽略这条指令，后续指令仍需处理**。

若没有出现以上问题，你需要对**表达式expr** 对应的内存（或**地址立即数**）进行读取或写入操作。首先规定本题中**基本数据类型**使用如下的格式进行输入输出：

- **整型数**的一般格式：正数与 0 不带符号；除 0 外，数字部分无前导零。
- **整型数**的 10 **进制格式**：没有额外前缀。
- **整型数**的 8 **进制格式**：除负号外以单个 0 为前缀。例如 233 的 8 进制格式为 0351；−10 的 8 进制格式为 -012。
- **整型数**的 16 **进制格式**：除负号外以 0x 为前缀，字母**大写**。例如 233 的 16 进制格式为 0xE9；−10 的 16 进制格式为 -0xA。
- **浮点数**的 16 **进制科学记数法**：一般格式为 <S>0x<A>.<B>p<C>，表示

$$\mathsf{S}\left(\mathsf{A}+\sum_{i=1}\mathsf{B}_i\times 16^{-i}\right)\times 16^{\mathsf{C}}$$

  其中：

  - <S> 为符号位，正数为空，负数为 -；
  - <A> 为 $\{1,\cdots,9,A,\cdots,F\}$ 中的一个字符；
  - <B> 为 $\{0,\cdots,9,A,\cdots,F\}$ 组成的字符串（包含空串），但不以 0 结尾；特别的，当 <B> 为空串时，之前的 '.' 一起**省略**；
  - <C> 为 10 **进制格式**整型数；特别的，当 <C> 为 0 时，仍需要**保留**。
  - 作为特例，规定 0x0p0 和 -0x0p0 分别与浮点数 0 和 −0 一一对应。

- 在输出时，你还需要考虑 ±∞ 与 ±NaN，规定它们的输出格式分别为 inf、-inf 与 nan、-nan；保证输入浮点数时**不会**出现这两种特殊值。

若操作为**写入操作**，你还需要检查 expr 的类型是否为**基本数据类型**。若不是，输出 cannot write to nonprimitive type 并**忽略这条指令**，**后续指令仍需处理**；否则，你需要将 value 写入到 expr 对应的内存中。

- 当 expr 的类型为**整型数**时，value 为一个 8、10 或 16 **进制格式**的整型常量。保证**value 在对应整型的取值范围内**。
- 当 expr 的类型为**浮点数**时，value 为一个 16 **进制科学记数法**表示的浮点常量。保证 value 为对应浮点类型的一个**精确值**。

若操作为**读取操作**，你需要根据 expr 的类型，以不同的格式输出其值：

- **整型数**：以 10 **进制格式**输出。
- **浮点数**：以 16 **进制科学记数法**输出其**精确值**。
- **地址立即数**或**指针**：输出 pointer to {addr}。其中 addr 为其指向的内存地址，使用整型数的 16 **进制格式**输出。
- **定长数组**：输出 array[N] at {addr}。其中 N 为数组长度，采用十进制格式；addr 为数组首地址，格式同上。
- **联合体**或**结构体**：输出 typename at {addr}。其中 typename 为类型名；addr 为**联合体**或**结构体**首地址，格式同上。

【输入格式】

从标准输入读入数据。

输入的第一行包含三个非负整数 n_1, n_2, n_3，分别表示每类任务的操作数量。$(0 \le n_1, n_2, n_3 \le 3 \times 10^4)$

接下来 n_1 行，每行一条**声明指令**或**定义指令**，格式见**任务一**描述。

接下来 n_2 行，每行一条**分配指令**，格式见**任务二**描述。

接下来 n_3 行，每行一条**读取指令**或**写入指令**，格式见**任务三**描述。

【输出格式】

输出到标准输出。

对于**任务一**，输出一行一个字符串表示**错误信息**。或者对于每种类型（按**声明指令**或**定义指令**首次出现的顺序）：

- 输出一行一个字符串与两个 10 **进制格式**的整数，分别表示**类型名**、**占用字节数**与**对齐字节数**，以空格分隔。

对于**任务二**的每条指令：

- 输出一行一个字符串表示**错误信息**。或者
- 输出一行一个 16 **进制格式**的非负整数，表示分配到的内存首地址。

对于**任务三**的每条指令：

- 输出一行一个字符串表示**错误信息**。或者
- 对于**读取指令**，输出一行表示读取到的数据（格式见**任务三**描述）。或者
- 对于**写入指令**，没有输出。

【样例 1 输入】

```
5 1 4
union data { u64 dword, u8[8] bytes };
struct node;
union pointer { node* ptr, u128 addr };
struct neighbors { pointer prev, pointer next };
struct node { data[2] dat, neighbors nbr };
alloc node[10] nodes;
write nodes[1].dat[1].dword = 0x123456789ABCDEF0;
write nodes[5].nbr.prev.addr = 0x30;
read nodes[1].dat[1].bytes[4];
read (*(nodes[5].nbr.prev.ptr)).dat[1].bytes[4];
```

【样例 1 输出】

```
data 8 8
node 48 16
pointer 16 16
neighbors 32 16
0x0
120
120
```

【样例 2 输入】

```
0 8 0
alloc u8 a;
alloc u128 b;
alloc u16 c;
alloc u32 d;
alloc u64 e;
alloc u128 f;
alloc u8 g;
alloc u32 h;
```

【样例 2 输出】

```
0x0
0x10
0x2
0x4
0x8
0x20
0x1
0x30
```

【样例 3 输入】

```
4 0 0
struct a;
```

```
3 struct b { a _a };
4 struct c;
5 struct a { b _b };
```

【样例 3 输出】

```
1 incomplete type a
```

【样例 4 输入】

```
1 0 5 5
2 alloc f16[10][10] a;
3 alloc u16[5] b;
4 alloc gg c;
5 alloc u64* d;
6 alloc f128* e;
7 write *d = 0x123456789ABCDEF0;
8 read &a[0];
9 read a[0][2];
10 write a[0] = 0x1.2p-1;
11 read *e;
```

【样例 4 输出】

```
1 0x0
2 0xC8
3 syntax error on line 4
4 0xE0
5 0xF0
6 pointer to 0x0
7 0x6.78p1
8 cannot write to nonprimitive type
9 0x4.8D159E26AF37BCp-4109
```

【样例 5】

见题目目录下的 *5.in* 与 *5.ans*。

【子任务】

对于所有的数据，满足 $0 \le n_1, n_2, n_3 \le 3 \times 10^4$。

保证输入文件大小、输出文件大小均不超过 2^{24} 字节。

保证本题中出现的所有类型（含中间类型）占用字节数不超过 2^{120}。

<table>
<tr><th>子任务编号</th><th>直接依赖</th><th>分值</th><th>$n_1 \le$</th><th>$n_2 \le$</th><th>$n_3 \le$</th><th>特殊性质</th></tr>
<tr><td>1sc</td><td>无</td><td>5</td><td rowspan="2">10</td><td rowspan="2">0</td><td rowspan="4">0</td><td>A</td></tr>
<tr><td>1se</td><td>1sc</td><td>3</td><td>无</td></tr>
<tr><td>2sc</td><td>无</td><td>5</td><td rowspan="6">0</td><td rowspan="10">10</td><td>A</td></tr>
<tr><td>2se</td><td rowspan="2">2sc</td><td>3</td><td>无</td></tr>
<tr><td>23sc</td><td>7</td><td rowspan="8">10</td><td>AB</td></tr>
<tr><td>23se</td><td>2se,23sc</td><td>2</td><td>B</td></tr>
<tr><td>23fsc</td><td>23sc</td><td>5</td><td>A</td></tr>
<tr><td>23fse</td><td>23se,23fsc</td><td>2</td><td>无</td></tr>
<tr><td>123sc</td><td>1sc,23sc</td><td>11</td><td rowspan="4">10</td><td>AB</td></tr>
<tr><td>123se</td><td>1se,23se,123sc</td><td>3</td><td>B</td></tr>
<tr><td>123fsc</td><td>123sc</td><td>5</td><td>A</td></tr>
<tr><td>123fse</td><td>23fse,123se,123fsc</td><td>5</td><td rowspan="2">无</td></tr>
<tr><td>12me</td><td>1se,2se</td><td>2</td><td>300</td><td rowspan="3">300</td><td>0</td></tr>
<tr><td>23me</td><td>23se</td><td>2</td><td rowspan="2">0</td><td rowspan="2">300</td><td>B</td></tr>
<tr><td>23fme</td><td>23fse,23me</td><td>5</td><td rowspan="3">无</td></tr>
<tr><td>1le</td><td>1se</td><td>5</td><td>3×10^4</td><td>0</td><td rowspan="3">0</td></tr>
<tr><td>2le</td><td>2se</td><td>5</td><td>0</td><td rowspan="4">3×10^4</td></tr>
<tr><td>12lc</td><td>1sc,2sc</td><td>5</td><td>3×10^4</td><td rowspan="2">A</td></tr>
<tr><td>23flc</td><td>23fsc</td><td>7</td><td>0</td><td rowspan="2">3×10^4</td></tr>
<tr><td>123fle</td><td>(ALL)</td><td>13</td><td>3×10^4</td><td>无</td></tr>
</table>

特殊性质：

- A：保证没有错误。
- B：任务三中不涉及浮点数输入输出。

【提示】

由于部分数据规模较大，你可能需要使用高精度整型数：

- 在 C++ 语言中，你可以使用 GCC 的 `__int128` 和 `unsigned __int128` 类型。
- 在 Java 语言中，你可以使用 `BigInteger` 类。
- 在 Python 语言中，默认的 `int` 类型即可满足精度要求。

【名词解释】

补码：正数与 0 直接使用二进制表示；负数使用其绝对值的二进制表示，然后按位取反再加一。例如 -10 的 8 位**补码**为 `11110110`。

***IEEE Std 754-2019* 标准**：规定了浮点数的表示方式，包括 16、32、64 与 128 位四种位宽。

其表示的数值形如 $(-1)^s \cdot M \cdot 2^E$，其中 $s \in \{0,1\}$ 控制**符号**；$M \in [1,2)$ 为**尾数**；E 为**阶码**。

其编码按二进制位从高到低分为 `s`、`exp` 与 `frac` 三部分，分别与 s、E 与 M 对应，其中 `s` 占 1 位，取值与 s 相同；`exp` 占 w 位，`frac` 占 t 位，w 和 t 的取值随位宽改变而变化，`exp` 和 `frac` 与 E 和 M 的对应关系也随 `exp` 不同而有所差别。

- 当 $\texttt{exp} \neq \texttt{000}\cdots\texttt{0}$ 且 $\texttt{exp} \neq \texttt{111}\cdots\texttt{1}$ 时，此表示称为**规格化数**，此时 $E = \texttt{exp} - 2^{w-1} + 1$，$M = \texttt{1.frac} = 1 + \sum_{i=1}^{t} \texttt{frac}_i \cdot 2^{-i}$。
- 当 $\texttt{exp} = \texttt{000}\cdots\texttt{0}$ 且 $\texttt{frac} \neq \texttt{000}\cdots\texttt{0}$ 时，此表示称为**非规格化数**，此时 $E = 2 - 2^{w-1}$，$M = \texttt{0.frac} = \sum_{i=1}^{t} \texttt{frac}_i \cdot 2^{-i}$。
- 当 $\texttt{exp} = \texttt{000}\cdots\texttt{0}$ 且 $\texttt{frac} = \texttt{000}\cdots\texttt{0}$ 时，表示的数为 0。需要特别注意的是，此时无论 s 如何取值，表示的数值都是 0，但 s 仍然影响符号，即 $s=1$ 时表示 -0，$s=0$ 时表示 $+0$。
- 当 $\texttt{exp} = \texttt{111}\cdots\texttt{1}$ 且 $\texttt{frac} = \texttt{000}\cdots\texttt{0}$ 时，表示的数为 $\pm\infty$，符号由 s 决定。
- 当 $\texttt{exp} = \texttt{111}\cdots\texttt{1}$ 且 $\texttt{frac} \neq \texttt{000}\cdots\texttt{0}$ 时，表示的数为 $\pm$NaN，符号由 s 决定。

不同位宽的浮点数的 w 和 t 取值如下：

位宽	w	t
16	5	10
32	8	23
64	11	52
128	15	112

更多有关 ***IEEE Std 754-2019* 标准**的细节可以阅读**题目目录**下的 `IEEE Std 754-2019.pdf`。

【参考资料】

- "IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, vol., no., pp.1-84, 22 July 2019.

可容错实时流处理系统（stream）

【题目背景】

在互联网应用中，日志是非常重要的数据，因为互联网项目往往要求是 7×24 不间断运行的，所以能获取到监控系统运行的相关日志数据并进行分析就显得非常重要。网站流量统计是改进网站服务的重要手段之一，通过获取用户在网站的行为数据，进行分析，从而得到有价值的信息，并基于这些信息对网站进行改进。

现实世界中，需要处理各种类型的数据，数据集的性质往往决定了我们采取什么样的处理方式。常见的数据集包括**有界数据集**和**无界数据集**。

- **有界数据集**：有界数据集对开发者来说都很熟悉，例如我们从数据库中读取某个时刻的数据快照进行计算分析，就是针对有界数据集。其特点就是数据是静止不动的。所以有界数据集可以被称作为有时间边界的数据集，可以被统一处理。这种计算方式我们称之为**批处理**（Batch Process）。
- **无界数据集**：无界数据集是会发生持续变更的、连续追加的一种数据集。例如：网络传输流、实时日志信息等。对于此类持续变更、追加的数据的计算方式称之为**流处理**（Stream Process）。

由于我们的互联网项目是 7×24 不间断运行的，通过埋点日志获得的日志数据也是持续不间断产生的，因此日志事件流就是一种典型的无界数据，我们使用**流处理系统**来实时处理这些事件。

【题目场景】

假设你是一名负责统计网站点击事件流分析工程师。你的目标是通过分析埋点日志捕获的点击事件，实时统计不同时间段内，不同网址的点击率。本任务中，你需要处理 2023 年 10 月 25 日 9:00:00-21:00:00（Unix 时间戳：`1698195600-1698238800`）这个时间段内产生的网站点击事件，点击事件流中的每条数据如下所示：

Unix 时间戳 (UnixTimeStamp)	IP 地址	请求 URL	响应码
`1698195600`	`101.0.0.1`	`/a/...`	`200`
`1698195603`	`101.0.0.1`	`/a/...`	`404`
`1698196200`	`234.0.0.3`	`/b/...`	`200`
`1698196220`	`234.0.0.3`	`/b/...`	`200`
`1698196220`	`234.0.0.3`	`/a/...`	`200`

本次任务中，我们统计从 2023 年 10 月 25 日上午 9:00:00 开始，每个长度为 10 分钟（Unix 时间戳之差为 600，后续时间都用 Unix 时间戳来表示）的窗口内的点击事件的统计信息。上述例子最终应该得到的分析结果如下所示：

时间窗口	请求 URL	成功响应次数
[1698195600, 1698196200)	/a/...	1
[1698196200, 1698196800)	/a/...	1
[1698196200, 1698196800)	/b/...	2

为了简单起见，我们使用每个时间窗口的起始时间来代表该窗口，则上述统计结果可以简化为：

时间窗口	请求 URL	成功响应次数
1698195600	/a/...	1
1698196200	/a/...	1
1698196200	/b/...	2

我们实现了一个流处理程序click_count.cpp：

```
void createQuery() {
  // 定义输入数据的 schema
  TupleSchema inputSchema = TupleSchema::create()
          .addFixSizeField("timestamp", DataType::ULL)
          .addFixSizeField("ip", DataType::String)
          .addFixSizeField("url", DataType::String)
          .addFixSizeField("statusCode", DataType::Int);

  // Source - 根据输入文件路径和定义的输入 schema 来获得输入数据
  Query query = Query::source(inputSchema, paths)
        // Filter - 过滤掉失败请求，保留有效数据
        .filter(new Equal("statusCode", 200))
        // Map - 删除无效字段 ip、statusCode（已经使用过）
        //       增加新字段 clickNumber
        //            （本题中每次点击事件权重相同，都是1）
        .map(new Transform(...))
        // Window - 定义一个从1698195600开始，长度为600的滑动窗口
        //          按照窗口来分别统计各个 URL 的请求次数
        .window(new TumblingWindow(1698195600, 600))
        // Sink - 输出最终的统计结果
        .sink(OutputType::StandardOutput);
}
```

上述整个数据处理的流程抽象成一个如下图所示的**数据流**（Dataflow）。

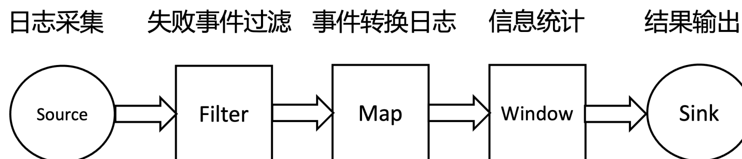


图 1: Dataflow-StreamGraph

该数据流是一个**有向无环图**（DAG），描述了数据如何在不同的操作之间流动。图中的顶点我们称之为算子（Operator），表示计算过程，边表示数据依赖关系。算子可以从日志采集器等外部数据源获得数据（如 Source 算子），或者从上游算子获取数据（如 Filter / Map / Window 等算子）。获得数据后每个算子根据自己的逻辑处理数据，然后产生新数据并且发送。

上图中，我们编写的程序直接映射成的数据流图是**逻辑流图**（Logical StreamGraph），表示的是计算逻辑的高级视图。而为了真正执行上面的流处理程序，我们需要把逻辑流图转换为物理数据流图，也被称为**执行图**（ExecutionGraph），如下图所示。

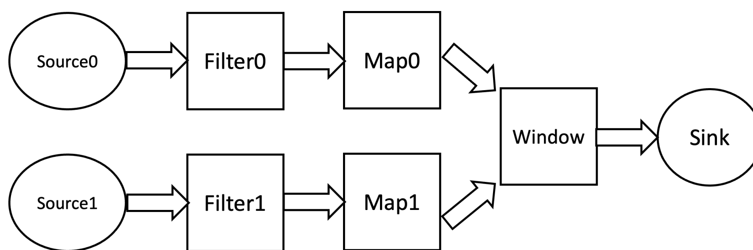


图 2: Dataflow-ExecutionGraph

- **逻辑流图**：根据用户编写的代码生成的最初的图，用来表示程序的拓扑结构。
- **执行流图**：流处理框架根据逻辑流图生成执行流图，它是逻辑流图的并行化版本，是调度层最核心的数据结构。

本题的实际场景中，我们有**两个**日志输入源 `Source0` 和 `Source1` 同时向流系统中输入数据。为了保证数据处理的高效性，我们并行处理这两个输入源，在最后的 Window Operator 中将两个输入源的信息进行汇总，并最终输出统计结果。

我们将实际执行图中的算子称为物理算子（Physical Operator），上图即为对应的实际执行图。其中 Source / Filter / Map Operator 的并行度都为 2，表示有两个算子子任务（Operator Subtask）。在现实场景中，每个子任务彼此独立，在不同的线程或在不同的计算机或容器中运行。

【题目描述】

现实世界中的流处理系统由于实际场景的不同也存在不同的问题。

- **延迟问题**：流系统内部能够保证算子间的数据流动是有序的，但现实场景中由于存在网络延迟等情况，点击事件的**发生时间**（被采集到的时间）和**处理时间**（到

达流系统的时间）是存在一定延迟的，即 12:00:00 发生的数据可能在 12:00:01 发生的数据之后才能到达流系统，这对我们的流系统的实时性和正确性提出挑战。

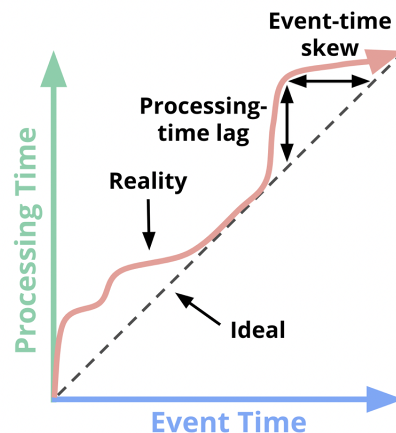
- **故障问题**：现实世界中，由于可能存在机器宕机的情况，流处理框架可能会面临任务执行到一半失败的情况，所以其需要具备一定的**容灾**（Fault Tolerance）能力。

你的任务就是帮助我们完善有**部分功能缺失**的流处理框架，使其能够**正确实时**地处理事件，同时具有一定的**容灾**能力。

【任务一：Watermark 的生成、对齐与传递】

我们定义日志采集器获取到点击事件的时间为**事件时间**（Event Time），事件流入系统被处理的时间为**处理时间**（Processing Time）。本题中点击事件的 UnixTimeStamp 字段代表的是**事件时间**。

在理想世界里，**事件时间** = **处理时间**，也就是事件一发生就会立即被处理。但是在现实世界中，这是不可能发生的，很多因素例如：共享资源有限，网络带宽 / CPU 等限制都会导致两者不一致。它们之间的关系可以用下图表示：

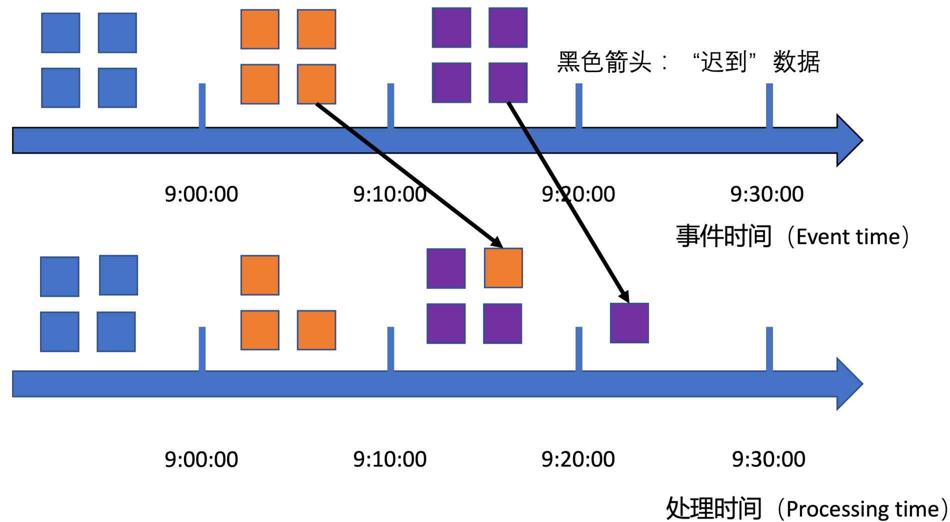


从上图可以看出**事件时间**和**处理时间**关系的两个特点：

- **处理时滞**：**处理时间**一定比**事件时间**晚。
- **事件时间偏差**：**处理时间**和**事件时间**的差并不固定。

本题中，我们针对**事件时间**而非**处理时间**进行统计。但是由于延迟问题的存在，会有何时结束统计的烦恼。例如，在 `1698196199` 时刻发生的事件可能在 `1698196199` 之后的任一时间到达流系统从而被处理，且该事件应该被归到 `[1698195600, 1698196200)` 窗口中，此时决定在何时结束 `[1698195600, 1698196200)` 窗口的信息统计是一个问题：结束过早则容易导致部分有价值的数据被遗漏；结束过晚则增加了处理延迟，不满足本次任务要求的**时效性**。

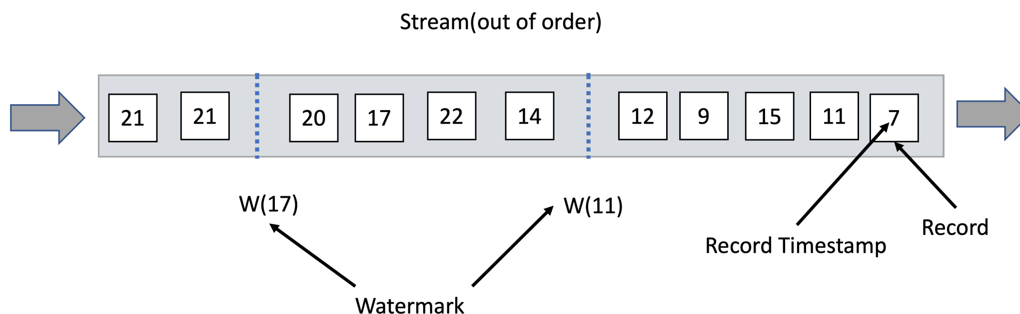
下图展示了一个基于事件时间的 10 分钟窗口算子（Window Operator）的例子，黑色箭头标注了其中的两个迟到数据：



所以，我们希望知道如何判断**事件时间**窗口的结束：窗口结束之后，不会再有这个窗口的数据到来。**Watermark** 就是一种常用处理机制。

Watermark：一个带有时间戳的特殊事件，代表了未到达事件的事件时间的下界。在本题的流处理系统中，Watermark 会被视作一种特殊的事件在执行图中传递，并且每个算子内部会维护其收到的最新的 Watermark 的值。**所有算子均需要保证其向下游传递的 Watermark 的时间戳严格单调递增**。

如下图所示，W(17) 代表包含了一个时间戳为 17 的 Watermark，其含义是断言在此之后不应该会出现时间戳小于 17 的数据，如果出现，判定为**迟到数据**（Lateness）。本题中我们对**迟到数据**采取**直接丢弃**的做法。



举例而言，若一个 `WindowOperator` 内部维护的最新的 Watermark 的值为 `1698196200`，这代表对于该算子而言，将来不会再出现事件时间小于 `1698196200` 的事件，`WindowOperator` 可以根据此信息结束 `[1698195600, 1698196200)` 窗口的信息统计，并输出这段时间的统计结果。因此，对于在 `1698196199` 时刻发生的事件，如果该事件到达 `WindowOperator` 时，其内部最新的 Watermark 的值大于 `1698196199`，则它会被视为**迟到数据**直接丢弃；反之按照正常事件更新 `WindowOperator` 的统计信息。

子任务一：实现周期性 Watermark 生成器

在不同的场景和不同的流处理系统中，Watermark 的生成策略也会有所不同。本题提供的流处理系统采用**周期性产生 Watermark** 的方式。

Watermark 生成器一般是绑定在 `SourceOperator` 的，Watermark 会在合适的时机由 `SourceOperator` 输入到系统中。本题中我们抽象出了一个 `WatermarkGenerator` 类，`SourceOperator` 会周期性调用 `WatermarkGenerator` 中的 `onPeriodicEmit` 函数来产生 Watermark。更具体的，在本题中 **`SourceOperator`** 每处理 10 个点击事件之后就会调用 `onPeriodicEmit` 函数来尝试生成一个 Watermark：若能够产生时间戳**严格单调递增**的 Watermark，则构造并返回一个 Watermark 类型的 `Event`；否则只需返回 `std::nullopt`。该接口的定义如下：

```
/**
 * WatermarkGenerator - 可以基于事件或者周期性的生成 watermark
 */
class WatermarkGenerator {
  public:
    /**
     * SourceOperator 处理每点击事件时都会调用，可以在其中检查或者记录事件的时间戳，
     * 也可以基于事件数据本身去生成 watermark
     */
    virtual void onEvent(Event &event) = 0;

    /**
     * SourceOperator 每处理 10 个点击事件之后就会调用该函数，
     * 若符合一定条件则产生 Watermark；反之则不会
     */
    virtual std::optional<Event> onPeriodicEmit() = 0;
};
```

为了帮助你理解，下面是一个我们已经实现了的周期性 Watermark 生成器的简单例子，该生成器生成的 Watermark 滞后于**处理时间**，且有一个固定的滞后时间 1 分钟，它假定元素最多会在 1 分钟延迟后到达处理程序中：

```
/**
 * 该生成器生成的 Watermark 滞后于处理时间固定量
 * 它假定元素会在有限延迟后到达处理程序中
 */
class ExampleWatermarkGenerator: public WatermarkGenerator {
```

```cpp
 private:
     size_t maxTimeLag = 60; // 1 min

 public:
   ExampleWatermarkGenerator() {}

   void onEvent(Event &event) { /* 这种情况下不需要实现该函数 */ }

   std::optional<Event> onPeriodicEmit() {
       // 获取当前时间点
       auto currentTime = std::chrono::system_clock::now();
       // 将时间点转换为时间戳
       std::time_t timestamp =
   std::chrono::system_clock::to_time_t(currentTime);
       UnixTimestamp unixTimeStamp =
   static_cast<UnixTimestamp>(timestamp);
       // 滞后于固定处理时间（系统时钟）的 WatermarkGenerator
       std::optional<Event> watermark(Event(EventType::WATERMARK,
   unixTimeStamp - maxTimeLag));
       return watermark;
   }
};
```

在本子任务中，你需要通过实现接口 WatermarkGenerator 来实现另一种周期性的 Watermark 生成器：其基于流数据来定期生成 Watermark，生成的 Watermark 滞后于数据流中**最新的事件时间**，且有一个固定的滞后时间 1 分钟。

你需要补全 stream.cpp 中的 **TimeLagWatermarkGenerator** 类，你可以在该类中添加任意你需要的字段来帮助你完成本任务。

为了解决该子任务，你可能需要了解以下信息：

代码中出现的 Event 有多种类型:RECORD(普通点击事件),WATERMARK,CHECKPOINTBARRIER 等。在本子任务中，你只需要关注 RECORD 和 WATERMARK 即可。此外，一些 Event 的相关用法如下（Event 类在 utils/Event.h 中定义和实现）：

```cpp
// 获取 Watermark 类型的 Event 的时间戳
// UnixTimeStamp 是我们预定义的，等价于 unsigned long long
Event watermark;
UnixTimeStamp eventTimestamp = watermark.getTimeStamp();

```

```
// 获取 RECORD 类型的 Event 的时间戳
Event record;
UnixTimestamp timestamp =
    std::stoull(record.getFieldValue("timestamp"));

// 定义一个带有时间戳 timestamp 的 Watermark 类型的 Event
UnixTimeStamp timestamp;
Event watermark = Event(EventType::Watermark, timestamp);
```

实现完成后，你可以通过我们提供的 MakeFile 得到一个可执行文件 `click_count`，针对该子任务，使用如下命令可以验证样例 1:

```
make clean
make
./click_count --paths ../data/1-0.in ../data/1-1.in --test-case 1
```

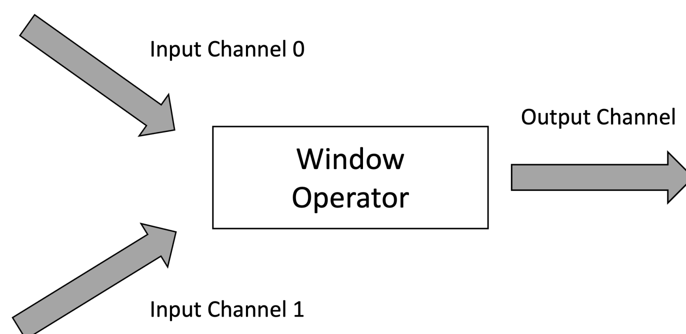
你也可以使用 `make test1` 命令快速测试样例 1 并与标准输出结果进行对比。

子任务二：Watermark 的对齐和传递

子任务一中你已经实现了周期性的 Watermark 生成器，而在流处理系统中，Watermark 会像常规事件一样在后续的算子中被处理。

从执行图中可以看出，`FilterOperator` / `MapOperator` 等属于单输入算子，只有一个输入管道，只需简单地将 Watermark 传递下去。而 `WindowOperator` 属于双输入-单输出算子，因此需要实现 `WindowOperator` 中 Watermark 的**对齐**以及**传递**操作。

一种常见策略是 `WindowOperator` 为每个输入管道维护一个 Watermark 值，分别记为 `(WatermarkChannel0, WatermarkChannel1)`，`WindowOperator` 在更新完收到的 Watermark 值向输出管道中转发时，可以取其中的最小值作为输出。即 `WatermarkOutput = min(WatermarkChannel0, WatermarkChannel1)`。



在本子任务中，你需要实现 `TwoInputWindowOperator` 中的 `process_event` 函数。在本子任务中输入的 `Event` 只有 `WATERMARK` 一种类型。你可以在 `TwoInputWindowOperator` 中添加字段来帮助你的实现。与子任务一类似，你也需要保证输出的 Watermark 的时

间戳**严格单调递增**，如果收到一个新的 Watermark 后 `WatermarkOutput` 保持不变，则不需要输出 Watermark。

```
/**
 * @brief 处理输入事件
 * @param watermark：到达该 Window 的一个 Watermark
 * @param channelId：该 Watermark 是哪个管道输入的，
 *                    值为 0 或 1，对应 Channel0/1
 */
void TwoInputWindowOperator::process_event(Event event, int
    channelId) {
    // TODO: 处理输入事件
}
```

在本子任务中，`TwoInputWindowOperator` 生成需要输出的 Watermark 之后需要调用我们提供的 `emit` 函数进行转发。在下发框架中，`emit` 函数的作用是把各种类型的 `Event` 以字符串的形式输出到标准输出，其使用方式如下：

```
// 将 Event 字符串化并输出到标准输出，该字符串的形式是我们在 Event
    中预定义好的
Event event;
emit(event);
```

实现完成后，你可以通过我们提供的 MakeFile 得到一个可执行文件 `click_count`，针对该子任务，使用如下命令可以验证样例 2:

```
make clean
make
./click_count --paths ../data/2-0.in ../data/2-1.in --test-case 2
```

你也可以使用 `make test2` 命令快速测试样例 2 并与标准输出结果进行对比。

至此，我们已经简单实现了 Watermark 的生成、对齐与传递的过程。

【任务二：算子对 Watermark 的使用】

本次任务中涉及到的算子可以被简单划分为两类：

- **无状态算子**（Stateless Operator）：`FilterOperator` / `MapOperator` 等与时间无关的算子，这类算子本身不需要存储额外的事件信息，只需要根据处理逻辑对到来的每条网站点击数据进行处理即可。数据的有界 / 无界、事件时间偏差都不会影响到这类算子的执行。
- **有状态算子**（Stateful Operator）：指 `WindowOperator` 等时间相关，需要存储多个事件信息的算子。

不同类型的算子对于到来的 Watermark 处理方式有所不同：对于**无状态算子**，接收到 Watermark 后可以直接转发；但是对于**有状态算子**，如果接收到的 Watermark 会对当前算子状态产生影响，根据规则修改相应状态后转发 Watermark 给下游算子，否则直接转发即可。

本题中我们使用是**滑动窗口**（Tumbling Window）：在时间维度上，按照固定长度将无界数据流切片，到来的事件会被分到唯一的数据切片中，最后针对每个数据切片进行处理。

本题中 `WindowOperator` 的主要功能是统计某些时间段内的网站点击量，算子的状态是一个时间窗口到网址和点击次数的映射 `map(timeWindow, map(url, click_count))`。为了简化处理，对于窗口 `[startUnixTimestamp, startUnixTimestamp + windowLength)`，我们使用起始时间戳 `startUnixTimestamp` 来代表该窗口。因此我们使用 `std::map<UnixTimestamp, std::map<std::string, int>>` 来表示窗口状态。

`WindowOperator` 需要依次处理输入管道中的事件，如果传入事件为网站点击事件，更新对应时间段的点击次数即可。若传入事件为 Watermark，则需要判断当前 Watermark 是否会触发统计信息的输出。如果会触发输出，`WindowOpeartor` 需要发送相应信息，同时内部的状态需要及时更新以减少内存占用。

我们规定：传递到 `WindowOperator` 的 Watermark 如果能够触发输出，则应**首先完成输出，再传递 Watermark**。

在本任务中，你需要完善 `TwoInputWindowOperator` 的 **`process_event`** 函数。在任务一中子任务二的基础上，你还需要正确处理从上游到来的 `RECORD`（点击事件）。具体要求如下：

- 正确维护内部状态。
- 正确实现 Watermark 的对齐和转发。
- 根据 Watermark 正确结束相应的窗口信息统计，并构造新的 `RECORD` 并输出。新的 `RECORD` 包含三个字段：`timeWindow - UnixTimestamp`, `url - string`, `totalNumber - int`。
- 本任务中所有生成的事件（`WATERMARK` 和 `RECORD`）需要转发时都通过 `emit` 函数输出到标准输出，若一个 Watermark 触发了多个 `RECORD` 的输出，则应**首先根据 `timestamp` 升序排序，其次根据 `url` 的字典序排序**。

```cpp
class TwoInputWindowOperator : public WindowOperator {
private:
    std::map<UnixTimestamp, std::map<std::string, int>> clickCount;

public:
    /**
     * @brief 处理输入事件
     * @param event 输入事件
```

```
     * @param channelId 输入事件的管道id
     */
    void process_event(Event event, int channelId) override;
};
```

为了实现上述功能，关于代码部分你可能需要了解到的信息：

- 输入的 RECORD 含有三个属性（Field），属性名称分别为：timestamp, url, clickNumber
- 可以通过调用 Event::getFieldValue(std::string fieldName) 函数来获得 Event 的属性值，该函数的返回值都是 std::string，你可以根据需要将其进行转化：

```
// 1. 获得 Event 的属性值
UnixTimestamp timestamp =
    std::stoull(event.getFieldValue("timestamp"));
std::string url = event.getFieldValue("url");
int click_count = std::stoi(event.getFieldValue("clickNumber"));

// 2. 构造 RECORD
std::vector<Field> fields = {Field("timeWindow", DataType::ULL),
    Field("url", DataType::String), Field("totalNumber",
    DataType::Int)};
std::vector<std::string> values = {std::to_string(1698195603),
    '/a/...', std::to_string(123)};
Event click_count_event = Event(fields, values);

// 3. 在一个 TwoInputWIndowOperator 中，
//    定位一个 event 的所属窗口
Event event;
UnixTimestamp timestamp =
    std::stoull(event.getFieldValue("timestamp"));
UnixTimestamp windowStartTimestamp =
    get_window_start_unix_timestamp(timestamp);
```

实现完成后，你可以通过我们提供的 MakeFile 得到一个可执行文件 click_count，针对该任务，使用如下命令可以验证样例 3：

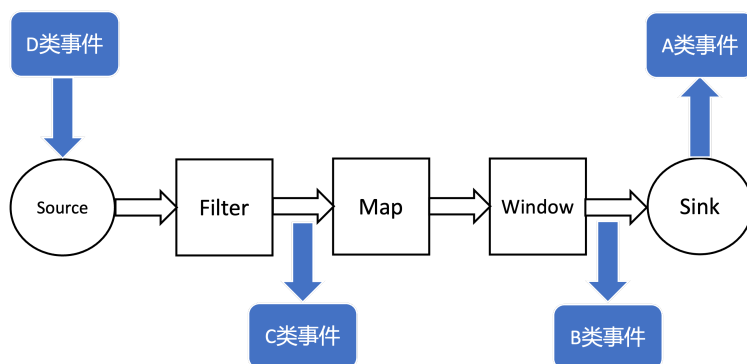
```
make clean
make
./click_count --paths ../data/3-0.in ../data/3-1.in --test-case 3
```

你也可以使用 `make test3` 命令快速测试样例 3 并与标准输出结果进行对比。

【任务三：实现 Exactly-Once 语义】

任务一和任务二中，我们在一定程度上保证了流系统的**实时性**，任务三则需要我们提供一定的**容错支持**。为了使容错机制生效，数据源需要能重放数据流。我们假定输入源有记录所有数据的能力，并且在故障重启后能够重放从任意指定位置开始的所有数据。

我们将机器宕机发生时，输入流中的日志点击事件分为四类：



- **A 类网站点击事件**：已经完成处理的网站点击事件。
- **B 类网站点击事件**：正在处理，已经在 `WindowOperator` 中更新过其状态信息，但是没有完全处理的事件。
- **C 类网站点击事件**：正在处理，即事件已经被输入到流系统中，但是其并未走完全部的处理流程，未能在 `WindowOperator` 中更新对其状态信息的事件。
- **D 类网站点击事件**：尚未处理，即尚未被输入到流系统中的事件。不管是已经产生但是尚未到达流系统的事件还是尚未发生的点击事件都属于此类事件。

经典的流处理系统中在发生故障之后的恢复有三种处理语义：**最多一次**（At-Most-Once）、**最少一次**（At-Least-Once）和**精确一次**（Exactly-Once）。

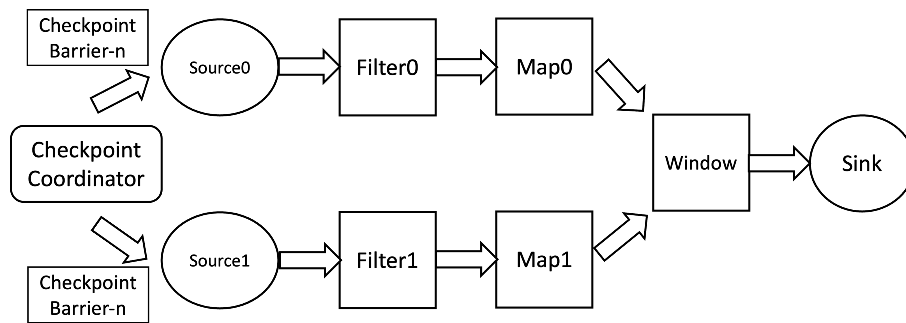
- **最多一次**：保证事件**最多**由应用程序中的所有算子处理一次。如果数据在被流应用程序完全处理之前发生丢失，则不会进行其他重试或者重新发送。具体到本题中故障重启后只会处理 D 类事件。这类语义的缺点在于**信息可能会发生丢失**（C 类点击事件本质上未被处理）。
- **最少一次**：应用程序中的所有算子都保证数据或事件**至少**被处理一次。这通常意味着如果事件在流应用程序完全处理之前丢失，则将从头重放或重新传输事件。这种处理语义下，B 类，C 类，D 类事件都会被重新输入并且处理，其缺点在于**某些信息会重复使用多次**（B 类事件被处理两次）。
- **精确一次**：即使是在各种故障的情况下，流应用程序中的所有算子都保证事件只会被**精确**一次的处理。这种语义下，只有 C 类，D 类事件被重新处理，所有的信

息刚好被处理一次，这类语义的缺点在于**实现机制相对复杂**。

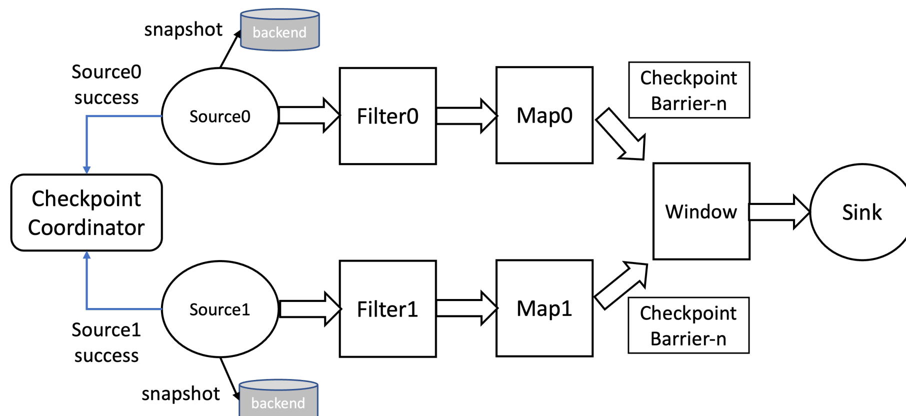
本任务中，我们的流系统希望能够实现**精确一次**语义，从而保证信息的准确性。我们使用分布式快照检查点（Checkpointing）机制来支持**精确一次**语义。该机制基于 **Chandy – Lamport** 算法，首先让每个节点独立建立检查点保存自身快照，并最终达到整个作业全局快照的状态。有了全局快照，当我们遇到故障或者重启的时候就可以直接从快照中恢复。

我们创建了一个检查点协调器（Checkpoint Coordinator），全权负责本应用的快照制作。在一个需要执行分布式快照算法的应用启动时，流程如下所示：

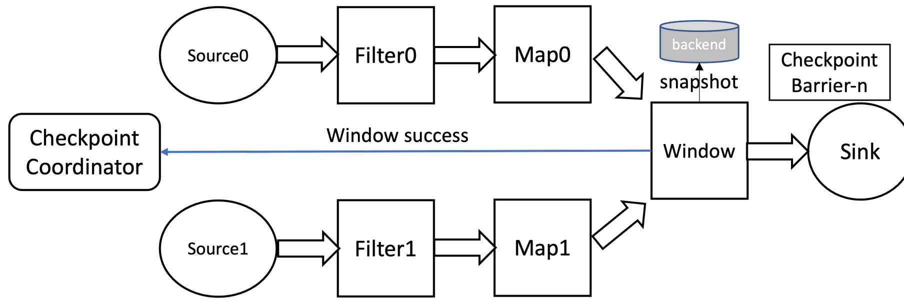
1. **开始阶段**：检查点协调器周期性的向该流应用的**所有** `SourceOperator` 发送 CheckpointBarrier-n 事件。



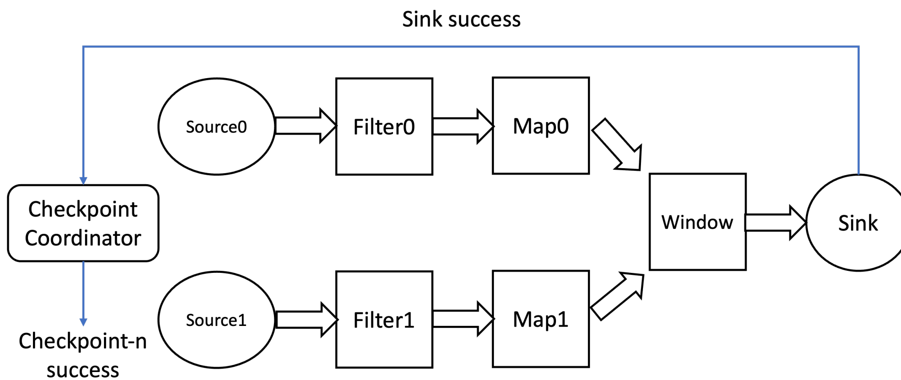
2. **算子处理阶段**：当某个 `SourceOperator` 收到一个 CheckpointBarrier-n 时，便暂停数据处理过程，将自己的状态（新输入到流处理系统中的日志点击事件的位置）制作成快照，并保存到指定的持久化存储中，最后向检查点协调器报告自己快照制作情况，同时转发该 CheckpointBarrier-n，恢复数据处理。



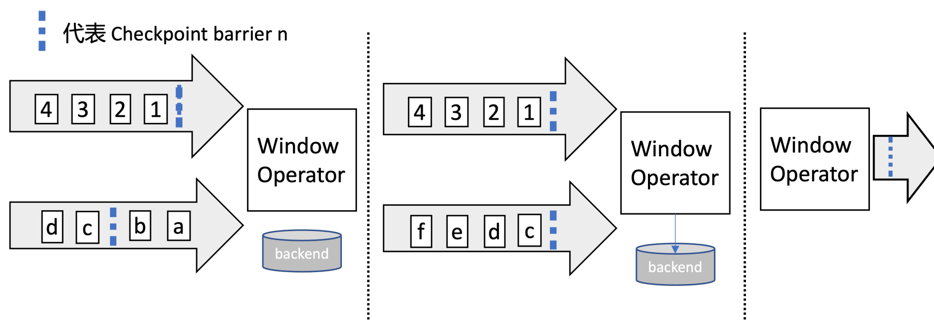
3. **对齐和转发阶段**：下游**无状态算子**收到 CheckpointBarrier-n 之后，直接转发即可；而**有状态算子**（如 `WindowOperator`）会暂停自己的数据处理过程，然后将自身的相关状态制作成快照，并保存到指定的持久化存储中，最后向报告自身快照情况，同时转发该 CheckpointBarrier-n，恢复数据处理。
4. **提交阶段**：最后 CheckpointBarrier-n 传递到 `SinkOpeartor`，快照制作完成。当检查点协调器收到所有算子的报告之后，认为该周期的快照制作成功；否则，如



果在规定的时间内没有收到所有算子的报告，则认为本周期快照制作失败。



本题场景中，`TwoInputWindowOperator` 有两个输入管道，接下来描述两个输入管道分别收到 CheckpointBarrier-n 时的处理步骤：我们会先暂时阻塞先收到 CheckpointBarrier-n 的输入管道，等到另一个输入管道中相同编号的 CheckpointBarrier 到来时，再制作自身快照并向下游广播该 CheckpointBarrier-n。一个例子如下图所示：



1. **预设条件**：`WindowOperator` 有两个输入管道：`Channel0`（输入网站点击数据 $1, 2, 3, \cdots$）和 `Channel1`（输入网站点击数据 $a, b, c, \cdots$）
2. **对齐阶段**：在 CheckpointBarrier-n 的制作过程中，由于某些原因，`Channel0` 上的 CheckpointBarrier 先到来，这时算子暂时将 `Channel0` 的输入通道阻塞，仅接收 `Channel1` 的数据。
3. **转发阶段**：当 `Channel1` 上的 CheckpointBarrier 到来时，`WindowOperator` 会制作快照，将自身状态保存到磁盘中，然后创建一个新的 CheckpointBarrier-n，通过 `emit` 函数来将其转发（本题中你不需要考虑向检查点协调器发送确认信息的步骤）。

上述流程即为所有的执行流程，当由于机器原因出现故障时，检查点协调器会将 SourceOperator 和 WindowOperator 统一恢复到最新的成功周期的快照，然后恢复数据流处理。分布式快照机制保证了数据仅被处理一次的语义。

在本任务中，你需要继续完善 TwoInputWindowOperator 类的 **process_event** 函数，并实现 **do_checkpoint** 函数。在任务二的基础上，你还需要正确处理从上游到来的 CHECKPOINTBARRIER，在合适的时机调用你实现的 do_checkpoint 完成快照创建。具体要求如下：

- 正确处理 WATERMARK，RECORD 和 CHECKPOINTBARRIER 三类事件。
- 正确实现 do_checkpoint 函数，该函数能够将 TwoInputWindowOperator 的状态持久化。处理 CHECKPOINTBARRIER 时需要调用该函数。
- **持久化快照的格式说明：**
  - 文件名称：函数内已经定义好。
  - 文件内容：不需要文件头，且每一行都有三个值，分别代表：timeWindow，url，totalNumber，值之间使用空格分割。对于 map<UnixTimestamp, map<url, int>> state，首先按照时间戳的升序排序，其次按照 url 字典序排序。

```
class TwoInputWindowOperator : public
    TwoInputWindowOperatorInterface {
private:
    std::map<UnixTimestamp, std::map<std::string, int>> clickCount;

public:
    /**
     * @brief 创建快照文件
     * @param checkpointId 快照编号
     */
    void do_checkpoint(int checkpoint) override;
    /**
     * @brief 处理输入事件
     * @param event 输入事件
     * @param channelId 输入事件的管道id
     */
    void process_event(Event event, int channelId) override;
};
```

实现本任务时你可能需要使用到的辅助函数说明（我们的框架在调度时会使用 isAvailable 函数检查输入管道是否阻塞，因此如果你不想使用 prohibitChannel

与 resumeChannel 函数，你也需要设置 probihitChannelId 使得 isAvailable 函数能够正确返回管道的阻塞状态）：

```
// 构造 CHECKPOINTBARRIER
int checkpointId = 0;
Event barrier = Event(EventType::CHECKPOINTBARRIER, checkpointId);

class TwoInputWindowOperatorInterface : public WindowOperator {
protected:
    /*
    * 处于禁用状态的输入管道id
    * 禁用状态：若该双输入算子的某输入管道处于禁用状态，
    * 则该 TwoInputWindowOperator 将停止接受该管道的数据，
    * 直到该管道解禁。
    * 在本题情况下，管道 id 的值为 0 或 1
    */
    std::optional<int> prohibitChannelId;

public:
    /**
     * @brief 将 channelId 对应的管道禁用
     * @param channelId
     */
    void prohibitChannel(int channelId) {
        if ((channelId != 0) && (channelId != 1)) { return; }
        if (!prohibitChannelId.has_value()) { prohibitChannelId =
    channelId; }
    }

    /**
     * @brief 恢复接受从 channelId 对应的管道来的数据
     * @param channelId
     */
    void resumeChannel(int channelId) {
        if ((channelId != 0) && (channelId != 1)) { return; }
        if (prohibitChannelId.has_value()) {
            if (prohibitChannelId.value() == channelId) {
                prohibitChannelId = std::nullopt;
```

```
35                 }
36             }
37     }
38 
39     /**
40      * @brief 查看 channelId 对应的管道是否处于禁用状态
41      * @param channelId
42      */
43     bool isAvailable(int channelId) {
44         if (prohibitChannelId.has_value()) {
45             if (prohibitChannelId.value() == channelId) {
46                 return false;
47             }
48         }
49         return true;
50     }
51 };
```

实现完成后，你可以通过我们提供的 MakeFile 得到一个可执行文件 `click_count`，针对该子任务，使用如下命令可以验证样例 4:

```
1 make clean
2 make
3 ./click_count --paths ../data/4-0.in ../data/4-1.in --test-case 4
```

你也可以使用 `make test4` 命令快速测试样例 4 并与标准输出结果进行对比。

【样例 1 输入】

`Source0` 的输入：

```
1 1698195600  10.0.0.1  /a/...  200
2 1698195600  10.0.0.1  /a/...  200
3 1698195603  215.0.0.1  /y/...  200
4 1698195605  125.0.0.1  /p/...  200
5 1698195609  17.0.0.1  /m/...  200
6 1698195610  78.0.0.1  /x/...  200
7 1698195620  240.0.0.1  /k/...  200
8 1698195637  35.0.0.1  /f/...  200
9 1698195651  95.0.0.1  /x/...  503
```

```
1698195642  118.0.0.1  /q/...  200
1698195655  160.0.0.1  /p/...  200
```

Source1 的输入（空文件）：

```
```

【样例 1 输出】

```
WATERMARK-1698195591
```

【样例 2 输入】

Source0 的输入：

```
1698195603  215.0.0.1  /y/...  200
1698195605  125.0.0.1  /p/...  200
1698195609  17.0.0.1  /m/...  200
1698195610  78.0.0.1  /x/...  200
1698195620  240.0.0.1  /k/...  200
1698195621  12.0.0.1  /m/...  200
1698195628  145.0.0.1  /h/...  200
1698195636  98.0.0.1  /h/...  200
1698195628  247.0.0.1  /a/...  200
1698195637  35.0.0.1  /f/...  200
1698195642  95.0.0.1  /x/...  503
```

Source1 的输入：

```
1698195600  102.0.0.1  /f/...  200
1698195610  74.0.0.1  /t/...  200
1698195610  50.0.0.1  /n/...  200
1698195618  149.0.0.1  /o/...  200
1698195618  103.0.0.1  /o/...  404
1698195624  236.0.0.1  /q/...  200
1698195626  16.0.0.1  /f/...  200
1698195628  102.0.0.1  /q/...  200
1698195626  124.0.0.1  /z/...  200
1698195634  205.0.0.1  /l/...  200
1698195639  229.0.0.1  /o/...  200
```

【样例 2 输出】

```
WATERMARK-1698195574
```

【样例 3】

见题目目录下的 `3-0.in`，`3-1.in` 与 `3.ans`。

【样例 4】

见题目目录下的 `4-0.in`，`4-1.in`，`4.ans` 与 `4-CLICK_COUNT_CHECKPOINT_0.ans`。

【子任务】

子任务编号	分值	说明
1	20	任务一/子任务一
2	20	任务一/子任务二
3	30	任务二
4	30	任务三

【评分方式】

本题的评测主要关注实现的正确性，时间限制与空间限制均非常宽松。

你提交的代码中无需包含 `main` 函数，只需提交 `stream.cpp` 中的内容，评测环境会将代码框架的其他文件与你提交的代码一起进行编译。

评测环境中使用的代码框架与下发框架**不完全相同**，但你需要实现的接口一致。**禁止尝试以任何方式攻击代码框架或利用其漏洞，无论是否攻击成功，一经发现，取消竞赛成绩**。

【参考资料】

- Chandy, K. M., & Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS), 3(1), 63-75.
- Akidau, T., Begoli, E., Chernyak, S., Hueske, F., Knight, K., Knowles, K., ... & Sotolongo, D. (2021). Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow (Vol. 14, No. 12). Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).