’ 

**数据库系统实验报告2**

题 目 多路归并排序算法的实现

学 院 软件学院

专 业 软件工程

学 号 120L021011

学 生 石卓凡

哈尔滨工业大学计算学部软件学院

2022.春

**要求：**

1：自己设计记录格式，至少包括2个属性（A和B），其中A为数值型，B作为记录的内容类型不限。

2：随机生成足够数量的记录，并存储为外存文件（尽量选择2进制格式）。

3：基于数值型属性A，用高级语言实现多路归并排序算法，并分析性能（时间和空间）。

目录

[1：描述实验环境的构建（记录和文件的准备）； 2](#_Toc134297032)

[模拟环境设定 2](#_Toc134297033)

[Record类 3](#_Toc134297034)

[2：给出基本算法设计的伪代码或流程图； 3](#_Toc134297035)

[外部排序算法设计的流程： 3](#_Toc134297036)

[分析时空复杂度 5](#_Toc134297037)

[时间复杂度 5](#_Toc134297038)

[空间复杂度 6](#_Toc134297039)

[优化第二趟归并的时间复杂度 6](#_Toc134297040)

[朴素算法思路概述 7](#_Toc134297041)

[最小堆思路概述 7](#_Toc134297042)

[败者树思路概述 7](#_Toc134297043)

[3：描述主要函数的设计过程； 8](#_Toc134297044)

[lab2\_main.cpp 8](#_Toc134297045)

[record.h 8](#_Toc134297046)

[Record类 8](#_Toc134297047)

[败者树LoserTree的设计： 8](#_Toc134297048)

[最小堆的实现 10](#_Toc134297049)

[externalSorting.h 10](#_Toc134297050)

[生成随机记录 10](#_Toc134297051)

[第一趟排序firstRun() 11](#_Toc134297052)

[第二趟排序SecondRUN() 11](#_Toc134297053)

[4：程序运行结果； 12](#_Toc134297054)

# 1：描述实验环境的构建（记录和文件的准备）；

**说明：**该实验模拟外部排序，*考虑内存有限的条件，来在内存中多次生成随机记录，直到生成到指定数据记录数，*设定条件:内存<总计Record大小<外存

在generateRecords（）进行调用initialRecords生成随机数据，并且将生成的records写入文件input\_records

在Record \*initialRecords(int num\_records, int a\_max) 在内存中生成随机Records，数据的A最大值为a\_max，返回指向Records的指针

其中，利用for循环遍历给每一个records随机数值

## 模拟环境设定

*由于内存<<总计Record大小<<外存，并且采用两趟排序，因此需要提前通过宏定义，来设定模拟环境的条件*

#define RECORDS\_TOTAL\_NUM 20000 // 定义记录总数  
#define A\_MAX 30000 // 定义记录中A属性的最大值  
#define BUFFER\_TOTAL\_SIZE 1000 // 定义总缓冲区大小  
#define SUB\_BUF\_SIZE 20 // 有序子集合子缓冲区大小  
#define OUTPUT\_BUF\_SIZE 200 // 输出缓冲区大小

#define WAYS\_NUM 20 // 定义归并排序的路数

***要求：***

* *RECORDS\_TOTAL\_NUM <= BUFFER\_TOTAL\_SIZE \* WAYS\_NUM*

*总缓冲区大小\*归并路数 >= 总Records数量*

*如果不满足，则内存和归并路数还不足以将外存中所有的records排序，会报错*

* *OUTPUT\_BUF\_SIZE + WAYS\_NUM \* (SUB\_BUF\_SIZE + 1) <= BUFFER\_TOTAL\_SIZE*

输出缓冲区大小+子缓冲区大小 <= 总缓冲区大小

如果不满足，则实际使用的缓冲区大小比设定的模拟缓冲区大小还要大，不符合条件

## Record类

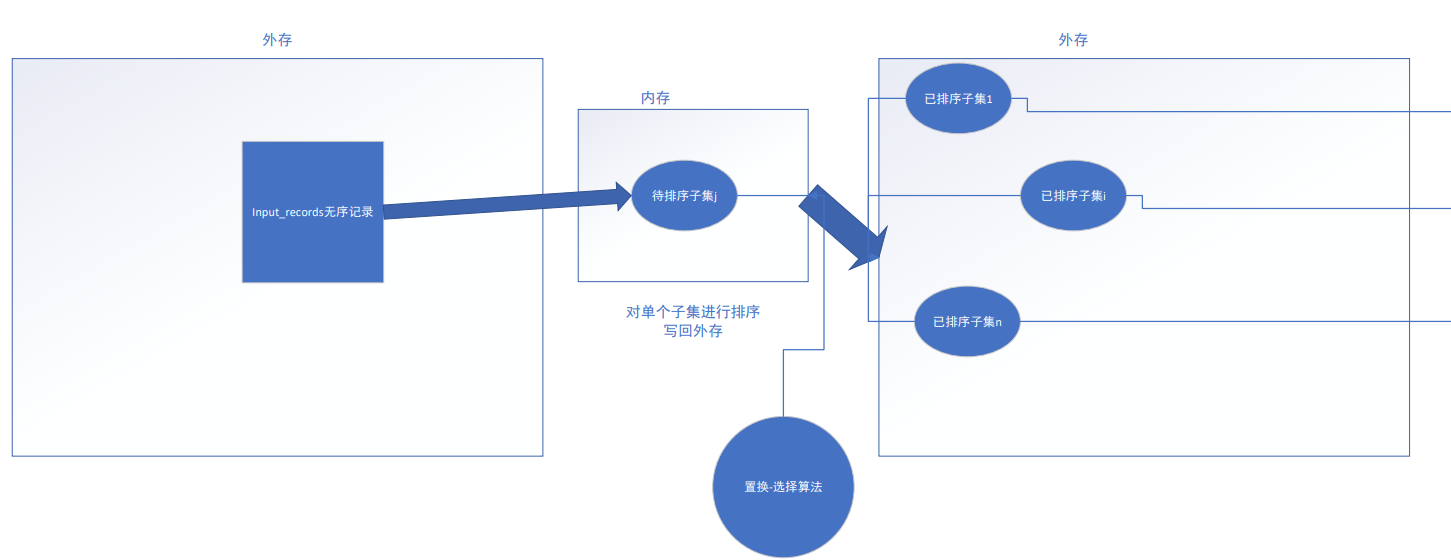
设计了Record记录格式的*记录类，包含一个整数型属性A和一个字符串型属性B*

int A;  
char B[20];

实现了方法compareTo，通过属性A来比较类的大小

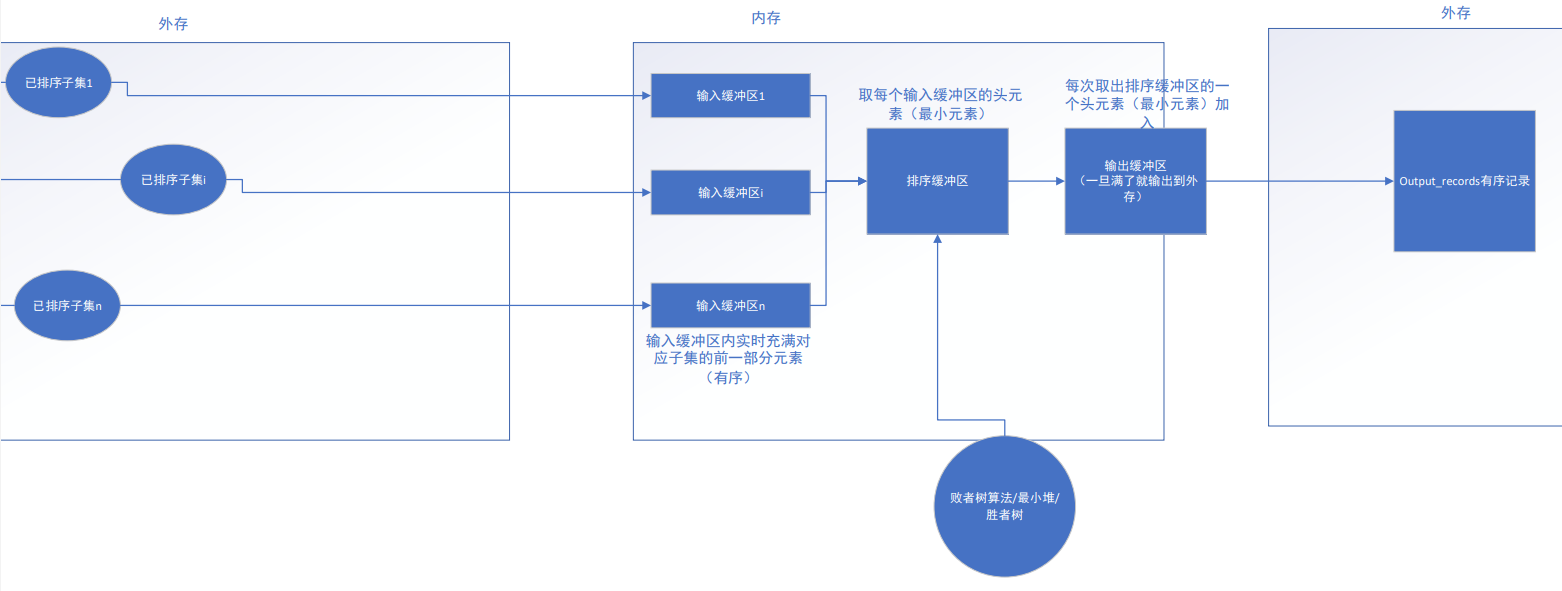
# 2：给出基本算法设计的伪代码或流程图；

## 外部排序算法设计的流程：



第一趟排序图

1. **第一步-生成初始归并段：**
   1. 将外存大文件input\_record作为流输入，利用置换-选择排序算法或者其他算法，将原文件划分为多个数据子集，输出为多个小的已排序好的records（初始归并段），保存至subSet
   2. 本实验中直接采用的是划分路数个数的子集，使得每个子集大小可以装入内存，对每个子集单独在内存中进行排序，排序后，输出为多个小的已排序好的records（初始归并段），保存至subSet



第二趟排序图

1. **第二步-归并：**
   1. 输入缓冲区初始化
      1. 把之前的每个已排序好的初始归并段（子集）的前面一部分元素，分配到到内存中的其对应的输入缓冲区
   2. 排序缓冲区初始化
      1. 将每个输入缓冲区的头元素（最小元素），加入到排序缓冲区
   3. 输入缓冲区工作：
      1. 每次输入缓冲区元素被排序缓冲区取出一个元素之后，实时从对应初始归并段中补充元素
   4. 排序缓冲区工作：
      1. 通过特定算法（算法1：顺序遍历所有元素，找出最小值 算法2：通过最维护最小堆，找出最小值 算法3：通过维护败者树，找出最小值），选出排序缓冲区中最小元素，然后加入到输出缓冲区
      2. 每次排序缓冲区的元素被输出缓冲区取出一个之后，实时从输入缓冲区补充一个元素进排序缓冲区
   5. 输出缓冲区工作：
      1. 如果输出缓冲区满了，则把缓冲区内容输出到外存中，清空输出缓冲区继续工作
      2. 直到所有输入缓冲区为空
2. 当归并完成时候，排序结束，此时output\_record是已经排序好的文件

## 分析时空复杂度

### 时间复杂度

假设需排序的数据集合大小为n，内存可容纳的数据大小为M，分为B（B>M）块，磁盘访问速度为D，磁盘读写操作的时间约为内存读写操作的时间的k倍，则有以下几种情况：

（1）内部排序的时间复杂度：对于快速排序和归并排序等排序算法，在输入数据量相同的情况下，时间复杂度误差不大，都为O(nlogn)。

（2）初始归并的时间复杂度：将B块数据全部读入内存，进行k路归并排序的最坏时间复杂度为O(B*M*logk)。

（3）归并路径的总长度：数据合并的过程可以看作一个二叉树，每个节点表示一次归并的过程，那么在B个块合并成1个有序数据集的过程中，归并路径的总长度为O(B\*logk)。

（4）磁盘I/O次数：假设磁盘读写操作时间为内存读写操作时间的k倍，那么在磁盘I/O次数上，外部排序的时间复杂度为O(nlog(B/M))。

综合（2）、（3）、（4）可知，外部排序的时间复杂度为O(nlog(B/M))

### 空间复杂度

外部排序的空间复杂度取决于内存可以容纳的数据大小M，以及对数据进行分块时每个块的大小。

如果将数据集合划分为B个块，则需要开辟至少B个缓冲区用于存储这些块，缓冲区的总大小为B\*M。

此外，还需要开辟一个输出缓冲区用于存储排序后的数据集合，输出缓冲区的大小也为M。因此，外部排序的空间复杂度为O(M)。

总结起来，外部排序的时间复杂度主要受到数据块数量、内存大小和磁盘I/O带宽等因素的影响，而空间复杂度则主要受到内存大小的影响。在实际应用中，需要根据具体情况选择合适的算法和参数以平衡效率和空间开销之间的关系。

## 优化第二趟归并的时间复杂度

在归并子文件时，可以使用最小堆、胜者树或败者树等数据结构来选择下一个输入文件中的最小元素。这些数据结构都可以在O(log n)的时间复杂度内选出最小值，其中n是数据数量。(以取出k个最小元素为例分析时空复杂度)

**朴素思路**是遍历所有元素，然后比较所有元素，拿到最小元素，时间复杂度O(nk)，空间复杂度O(n)

**最小堆**的思路是将每个输入文件的第一个元素放入堆中，然后选择堆中的最小元素。当选择完一个元素之后，将来自同一输入文件的下一个元素放入堆中，并重新调整堆结构。如果有k个输入文件，则有k个初始堆，需要选择k个最小元素，时间复杂度为O(n log k)。

**最小堆**的空间复杂度是O(k)，其中k是输入文件的个数。这是因为最小堆需要存储每个输入文件的第一个元素，并根据需要动态调整堆大小以选择下一个最小元素。由于只有一个元素从堆中弹出时，新的元素才会被添加到堆中，因此，堆的大小始终小于等于k。

**胜者树**的思路是维护一棵完全二叉树，其中每个叶子节点代表一个输入文件，每个节点代表胜者，即相应叶子节点所代表输入文件中当前最小的元素。对于胜者树中的每个节点，只需要比较其两个子节点的最小值，并将胜者更新为其中的较小值。这样做的时间复杂度是O(n log k)，其中k是输入文件的个数。

**败者树**的思路与胜者树类似，只是在更新节点值时选择败者而非胜者。这样做可以减少比较次数和存储空间，并且能够避免数据倾斜带来的问题。败者树的时间复杂度也是O(n log k)。

**胜者树和败者树**的空间复杂度都是O(k)。这是因为在每个决赛节点上，只有两个胜者或败者中的一个被保留，因此胜者树和败者树不需要为每个输入文件存储一个节点。另外，胜者树和败者树中的节点数量与叶子节点数量相同，即与输入文件的数量k相同。

其中，败者树通常被认为是最高效的选择。

本实验实现了**朴素算法，最小堆，败者树**

### 朴素算法思路概述

1. 遍历buffer\_sorting排序缓冲区的所有元素，选出min\_index最小值的索引
2. 如果已经读取完全部元组, 返回-1

### 最小堆思路概述

1. 维护一个最小堆，利用最初的buffer\_sorting,初始化最小堆
2. 利用最小堆选择所有buffer\_sorting中头部最小的元素，并把它放进外存，即根节点对应的元素
3. 更换刚才被选出来的元素pop，利用buffer\_sorting补充的新元素adjust调整最小堆
4. 如果某个子集对应的输入缓冲区空了，则剔除该某个子集对应的输入缓冲区，然后调整败者树
5. 循环以上1~4直到全部输入缓冲区为空

### 败者树思路概述

1. 维护一个败者树，利用最初的buffer\_sorting,初始化败者树
2. 利用败者树选择所有buffer\_sorting中头部最小的元素，并把它放进外存，即ls[0]对应的bq
3. 更换刚才被选出来的元素pop，利用buffer\_sorting补充的新元素adjust调整败者树
4. 如果某个子集对应的输入缓冲区空了，则剔除该某个子集对应的输入缓冲区，然后调整败者树
5. 循环以上1~4直到全部输入缓冲区为空

# 3：描述主要函数的设计过程；

## lab2\_main.cpp

**主函数入口：**

通过调用records.h与externalSorting.h完成随机记录生成，外部排序，结果展示

**分析性能的实现：**

通过利用clock()实时记录当前程序运行时间，来纵观对比时间效率

## record.h

### Record类

设计了Record记录格式的*记录类，包含一个整数型属性A和一个字符串型属性B*

int A;  
char B[20];

实现了方法compareTo，通过属性A来比较类的大小

#### Record \*initialRecords(int num\_records, int a\_max)

在内存中生成随机Records，数据的A最大值为a\_max，返回指向Records的指针

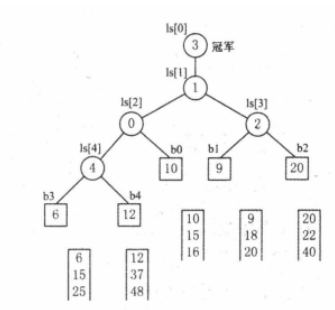
#### void print\_binFile\_for\_records(const char \*filename, int print\_num)

输出records的二进制文件数据，其中print\_num 需要输出的记录数,如果要读取整个文件中的所有记录，则可以设置 print\_num 为文件中总记录数目

#### Record \*\*createRecordsTwoDim(int m)

用于动态创建二维数组的函数，初始化一个二维记录数组,这个数组有m个指针, 每个指针分配n个空间，返回\* pRecords[m][n]

### 败者树LoserTree的设计：

​

**设计的败者树示意图**

* 按照这个图顺序定义b0,b1~b4的顺序，b0必须从第一个最小叶子结点作为b0，才符合b\_parnet = (b\_index+b\_size)/2
* 叶子节点对应着初始归并段的输入缓冲区
* 每次都是记录败者节点，拿着胜者去向上比较

#### 败者树的主要方法：

1. **void Adjust(LoserTree tree, int s, Record \*leaf)**

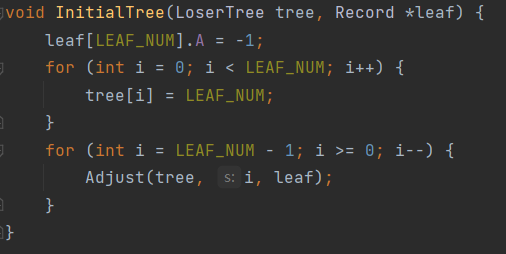
用于将败者树中的叶子结点与其双亲结点中记录的败者进行比较，调整败者的值

**应用场景**：每次叶子节点更新之后，去调整败者树

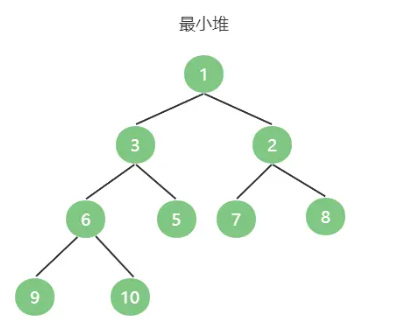


1. **InitialTree(LoserTree tree, Record \*leaf)**

初始化败者树



### 最小堆的实现



**设计的最小堆图**

利用的是c++的STL中的priority\_queue<int> min\_heap

## externalSorting.h

### 生成随机记录

利用generateRecords()函数生成随机Records，最大值为max\_value,内存中生成数据并写入外存



其中利用assert完成对设定条件是否合理的检查

### 第一趟排序firstRun()

#### divideSubSet()划分每个子集大小

将input\_records.bin原二进制数据记录进行划分，假设有n条记录，想将其均分到k个子集中，那么每个子集应该包含n/k条记录。当n不能被k整除时，余下的未分配的记录会放到前面的若干个子集中，每个子集比其他子集多容纳一条记录。

划分大小的结果存入subSetSize[]

#### sortSubSet()对每个子集排序并保存

依次遍历从input\_records.bin读取每个子集放入内存，此时内存足够大来对单个子集进行排序。使用快排对子集进行升序排序，然后将排序后的子集写入subSet\_i.bin的二进制文件

### 第二趟排序SecondRUN()

#### 初始化缓冲区

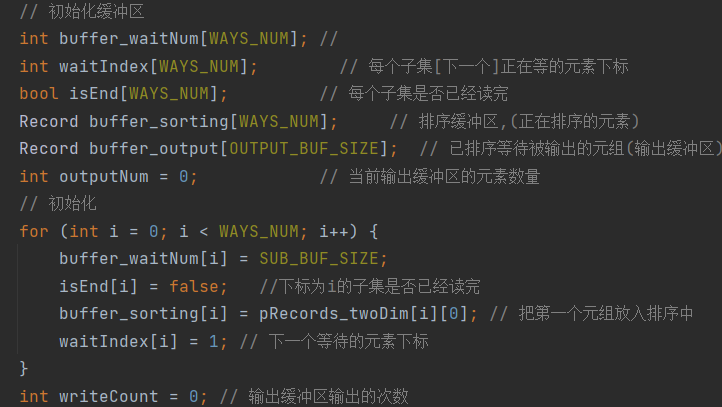
##### initBufferInput()初始化输入缓冲区

首先初始化输入缓冲区：利用initBufferInput()就是读取第一趟排序后的子集数据，每个子集都是取出当前头部一部分元素，放入其对应的输入缓冲区。

初始化后的输入缓冲区为pRecords\_twoDim

##### 初始化排序缓冲区及输出缓冲区

定义了一些数组变量和计数器变量，接着，通过循环初始化各个数组和变量，初始化外部排序所需要的各种变量和缓冲区，以便后续的外部排序操作



#### 利用算法进行归并排序mergeAndOutput（）

1. while循环，直到满足条件退出
   1. 取出当前buffer\_sorting最小元素下标，把当前排序缓冲区的最小值放入输出缓冲区
   2. 判断是否需要需要把输出缓冲区的内容输出到外存
      * 1. 如果输出缓冲区已满，则输出到外存
        2. 或者输出缓冲区未满，但全部子集已排序比较完，则输出到外存
   3. 如果某个子集缓冲区已经比较完，则从子集中补充元素到输入缓冲区

##### 三种取出最小值方法

顺序遍历（未优化版本）getMinNotOptimized(buffer\_sorting, WAYS\_NUM, A\_MAX)

败者树方法getMinByLoserTree(tree);

最小堆方法getMinByHeap(min\_heap)

# 4：程序运行结果；

* 1. 随机生成20000条乱序数据记录存入input\_record.bin二进制文件

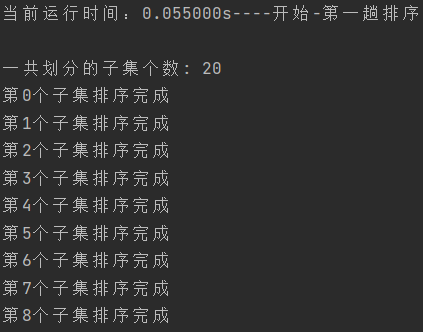
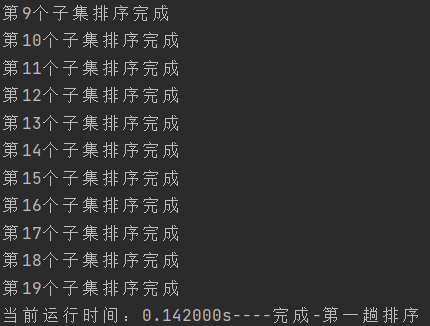


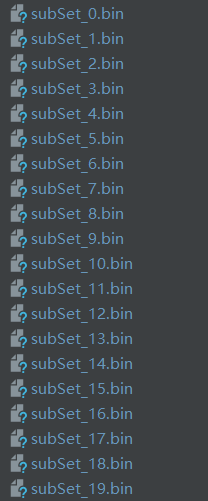
控制台信息



生成的记录

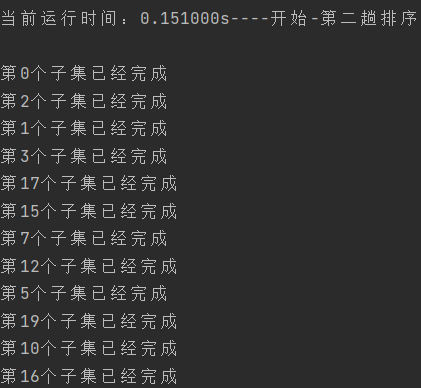
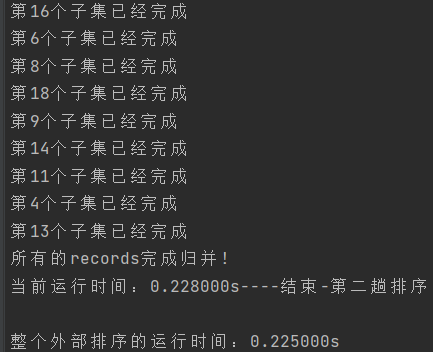
* 1. 进行第一趟排序，生成初始归并段

  控制台信息



生成的排序好的子集

* 1. 进行第二趟排序，归并，并生成最终文件output\_records.bin

控制台信息



排序好的记录

1. 外部排序结果展示

