海量数据场景高频面试题总结

目录

对于赶时间的同学，其中间过程可以选择跳过，直接看对应的最佳实践部分，这里进行一个目录综合

12. 40 亿个非负整数中找到出现两次的数
13. 大数据 Top K 问题常用套路

1. 统计不同号码的个数

题目来源：百度二面

题目描述

已知某个文件内包含大量的电话号码，每个号码的数字为 8 位，怎么统计不同号码的个数?

思路分析

这类题目其实是求解数据重复的问题，对于这类问题，我们可以采用**位图法**来进行处理。

8 位的电话号码可以表示的范围为 00000000 ~ 99999999。如果用 bit 表示一个号码，那么一共需要 1 亿个 bit，只需要大约 **10 MB** 的内存。

计算过程如下：

00000000 ~ 99999999 一共有 1 亿个数字， 1 亿 bit = 10 ^ 8 / (8 ^ 1000 ^ 1000) = 12.5 MB

这个时候，我们申请一个位图并且初始化为 0 ，然后遍历所有的电话号码，把遍历到的电话号码对应的位图中的 bit 设置为 1 。当遍历完成之后，如果 bit 值为 1，则表示这个电话号码在文件中存在，如果 bit 值为 0 则表示这个电话号码在文件中不存在。

最后，这个位图中 **bit 值为 1 的数量**就是不同电话号码的个数了。

那么，如何确定电话号码对应的是位图中的哪一位呢?

可以使用下面的方法来实现**电话号码和位图的映射**。

```
00000000 对应位图最后一位：0×0000…000001。
00000001 对应位图倒数第二位：0×0000…0000010（1 向左移 1 位）。
00000002 对应位图倒数第三位：0×0000…0000100（1 向左移 2 位）。
……
00000012 对应位图的倒数第十三位：0×0000…0001 0000 0000 0000（1 向左移 12 位）。
```

也就是说，电话号码就是这个数字 1 左移的次数。

具体实现

首先位图可以使用一个**int数组**来实现（在Java中int占用**4byte**）。

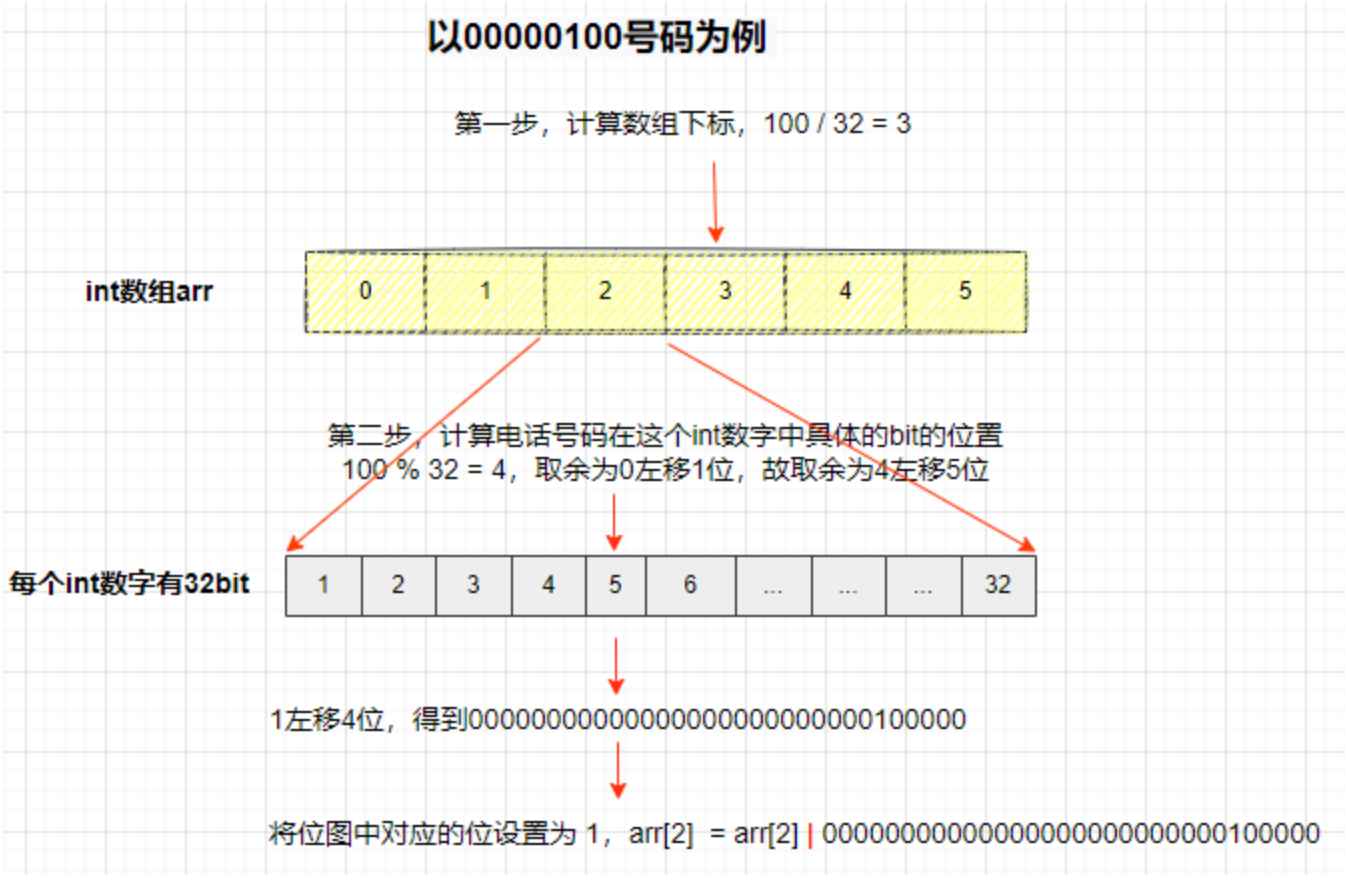
假设电话号码为 P，而通过电话号码获取位图中对应位置的方法为：

第一步，因为int整数占用4*8=32bit，通过 **P/32** 就可以计算出该电话号码在 bitmap 数组中的下标，从而可以确定它对应的 bit 在数组中的位置。

第二步，通过 **P%32** 就可以计算出这个电话号码在这个int数字中具体的bit的位置。只要把1向左移 **P%32** 位，然后把得到的值与这个数组中的值做或运算，就可以把这个电话号码在位图中对应的位设置为1。

以00000100号码为例。

1. 首先计算数组下标，100 / 32 = 3，得到数组下标位3。
2. 然后计算电话号码在这个int数字中具体的bit的位置，100 % 32 = 4。取余为0左移1位，故取余为4左移5位，得到000...000010000
3. 将位图中对应的位设置为 1，即arr[2] = arr[2] | 000..00010000。
4. 这就将电话号码映射到了位图的某一位了。



最后，统计位图中 bit 值为 1 的数量，就可以得到不同电话号码的个数了。

2. 出现频率最高的 100 个词

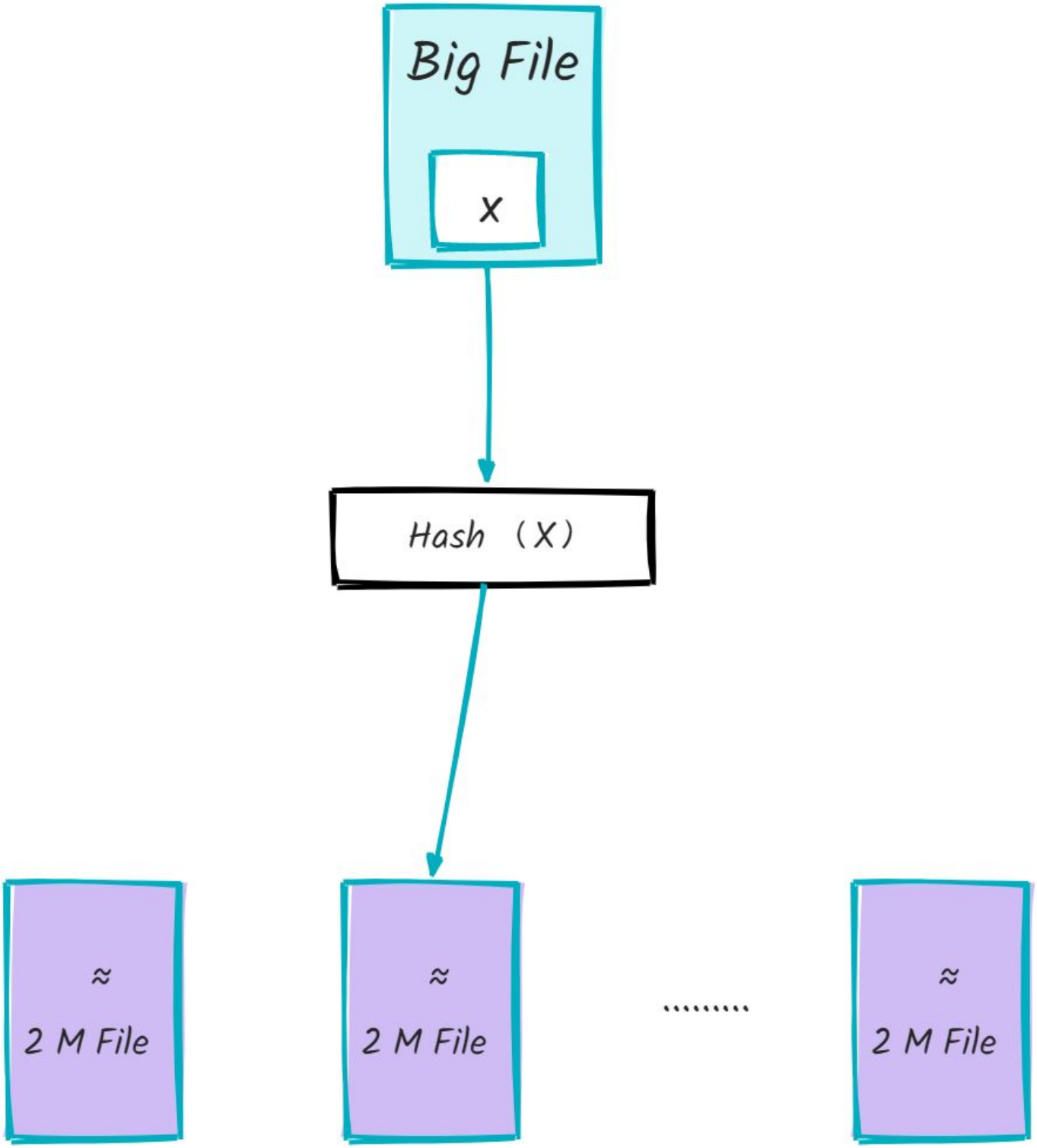
题目描述

假如有一个 1G 大小的文件，文件里每一行是一个词，每个词的大小不超过 16 bytes，要求返回出现频率最高的 100 个词。内存限制是 **10M**。

解法 1

由于内存限制，所以我们没有办法一次性把大文件里面的所有内容一次性读取到内存中去。

对此我们可以采用**分治的策略**来实现，把一个大文件分解成多个小文件，保证每个文件的大小小于 10 M，进而直接将单个小文件读取到内存中进行处理。



第一步，首先遍历一遍大文件，对遍历到的每个词 x ，执行 hash（x）% 500,将结果为 i 的词存放到文件 f（i）中，遍历结束之后，可以得到 500 个小文件，每个小文件的大小为 2 M 左右；

第二步，接着统计每个小文件中出现频率最高的 100 个词。可以用 HashMap 来实现，其中 key 为词，value 为该次出现的频率。（伪代码如下）

```java
BufferedReader br = new BufferedReader(new FileReader(fin));
String line = null;
while ((line = br.readLine()) != null) {
    if(map.containsKey(line)){
        map.put (line, map.get(x) + 1)
    }else{
        map.put(line,1);
    }
}

br.close();
```

对于遍历到的词 x，如果在 map 中不存在，则执行 map.put（x，1）。

若存在，则执行 map.put（x，map.get(x) + 1）,将该词出现的次数 + 1。

第三步，在第二步中找出了每个文件出现频率最高的 100 个词之后，通过维护一个**小顶堆**来找出所有小文件中出现频率最高的 100 个词。

具体方法是，遍历第一个文件，把第一个文件中出现频率最高的 100 个词构建成一个小顶堆。

如果第一个文件中词的个数小于 100，可以继续遍历第二个文件，直到构建好有 100 个结点的小顶堆为止。

继续遍历其他小文件，如果遍历到的词的出现次数大于堆顶词的出现次数，可以用新遍历道的词替换堆顶的词，然后重新调整这个堆位小顶堆。

当遍历完所有小文件后，这个小顶堆中的词就是出现频率最高的 100 个词。

总结

总结一下，这个解法的主要思路如下：

1. 采用**分治**的思想，进行哈希取余
2. 使用 HashMap 统计每个小文件单词出现的次数
3. 使用**小顶堆，**遍历步骤 2 中的小文件，找到词频 top 100 的单词。

很容易就会发现一个问题，如果第二步中，如果这个 1 G 的大文件中有某个词的频率太高，可能导致小文件大小超过 10 M，这种情况该怎么处理呢?

在此疑问上，我们提出了第二种解法。

解法 2

第一步：使用多路归并排序堆大文件进行排序，这样的话，相同的单词一定是紧挨着的

多路归并排序对大文件排序的步骤如下：

1. 将文件按照顺序切分成大小不超过 2 M 的小文件，总共 500 个小文件
2. 使用 10 MB 内存分别对 500 个小文件中的单词进行**排序**
3. 使用一个大小为 500 的堆，对 500 个小文件进行**多路排序，**然后将结果写到一个大文件中

其中第三步，对 500 个小文件进行多路排序的思路如下：

1. 初始化一个最小堆，大小就是有序小文件的个数 500。堆中的每个节点存放每个有序小文件对应的输入流。
2. 按照每个有序文件中的下一行数据对所有文件输入流进行排序，单词小的输入文件流放在堆顶。
3. 拿出堆顶的输入流，并且将下一行数据写入到最终排序的文件中，如果拿出来的输入流还有数据的话，那么就将这个输入流再次添加到栈中。否则说明该文件输入流中没有数据了，那么可以关闭这个流。
4. 循环这个过程，直到所有文件输入流中没有数据为止。

第二步：

1. 初始化一个 100 个节点的小顶堆，用于保存 100 个出现频率最高的单词。
2. 遍历整个文件，一个单词一个单词地从文件中读取出来，并且进行计数。

3. 等到遍历的单词和上一个单词不同的话，那么上一个单词及其频率如果大于堆顶的词的频率，那么放在堆中。否则不放

最终，小顶堆就是出现频率前 100 的单词了。

小结

解法 2 相对于解法 1，其更加严谨，如果某个词词频过高或者整个文件都是同一个单词的话，解法 1 不适用。

3. 5 亿个数的大文件怎么排序?

题目描述

给你一个文件，其大小为 4663 M，其中一共有 5 亿个数字，文件中的数据随机，一行一个整数，如何对这个大文件排序，这个怎么处理呢？（举例数据如下图所示）

```Java
23714719
13989184
23714893
37598491
28789147
4237184
......
137948
```

这里，针对这个问题我们来一步步推进，看看怎么实现：

内部排序

遇到排序问题，我们可以优先尝试一下内部排序，因为其排序的数字较多，所以我们可以采用分治的思想，这里优先选择我们的快速排序以及归并排序

Java

```java
private final int cutoff = 8;

public <T> void perform(Comparable<T>[] a) {
    perform(a,0,a.length - 1);
}

private <T> int median3(Comparable<T>[] a,int x,int y,int z) {
    if(lessThan(a[x],a[y])) {
        if(lessThan(a[y],a[z])) {
            return y;
        }
        else if(lessThan(a[x],a[z])) {
            return z;
        }else {
            return x;
        }
    }else {
        if(lessThan(a[z],a[y])){
            return y;
        }else if(lessThan(a[z],a[x])) {
            return z;
        }else {
            return x;
        }
    }
}

private <T> void perform(Comparable<T>[] a,int low,int high) {
    int n = high - low + 1;
    //当序列非常小，用插入排序
    if(n <= cutoff) {
        InsertionSort insertionSort = SortFactory.createInsertionSort();
        insertionSort.perform(a,low,high);
        //当序列中小时，使用median3
    }else if(n <= 100) {
        int m = median3(a,low,low + (n >>> 1),high);
        exchange(a,m,low);
        //当序列比较大时，使用ninther
    }else {
        int gap = n >>> 3;
        int m = low + (n >>> 1);
        int m1 = median3(a,low,low + gap,low + (gap << 1));
        int m2 = median3(a,m - gap,m,m + gap);
        int m3 = median3(a,high - (gap << 1),high - gap,high);
        int ninther = median3(a,m1,m2,m3);
```

```
            exchange(a,ninther,low);
        }

        if(high <= low)
            return;
        //lessThan
        int lt = low;
        //greaterThan
        int gt = high;
        //中心点
        Comparable<T> pivot =  a[low];
        int i = low + 1;

        /*
            * 不变式:
            *   a[low..lt-1] 小于pivot -> 前部(first)
            *   a[lt..i-1] 等于 pivot -> 中部(middle)
            *   a[gt+1..n-1] 大于 pivot -> 后部(final)
            *
            *   a[i..gt] 待考察区域
            */

        while (i <= gt) {
            if(lessThan(a[i],pivot)) {
                //i-> ,lt ->
                exchange(a,lt++,i++);
            }else if(lessThan(pivot,a[i])) {
                exchange(a,i,gt--);
            }else{
                i++;
            }
        }

        // a[low..lt-1] < v = a[lt..gt] < a[gt+1..high].
        perform(a,low,lt - 1);
        perform(a,gt + 1,high);
    }
```

归并排序

```java
/**
* 小于等于这个值的时候，交给插入排序
*/
private final int cutoff = 8;

/**
 * 对给定的元素序列进行排序
 *
 * @param a 给定元素序列
 */
@Override
public <T> void perform(Comparable<T>[] a) {
    Comparable<T>[] b = a.clone();
    perform(b, a, 0, a.length - 1);
}

private <T> void perform(Comparable<T>[] src,Comparable<T>[] dest,int low,
int high) {
    if(low >= high)
        return;

    //小于等于cutoff的时候,交给插入排序
    if(high - low <= cutoff) {
        SortFactory.createInsertionSort().perform(dest,low,high);
        return;
    }

    int mid = low + ((high - low) >>> 1);
    perform(dest,src,low,mid);
    perform(dest,src,mid + 1,high);

    //考虑局部有序 src[mid] <= src[mid+1]
    if(lessThanOrEqual(src[mid],src[mid+1])) {
        System.arraycopy(src,low,dest,low,high - low + 1);
    }

    //src[low .. mid] + src[mid+1 .. high] -> dest[low .. high]
    merge(src,dest,low,mid,high);
}

private <T> void merge(Comparable<T>[] src,Comparable<T>[] dest,int low,in
t mid,int high) {

    for(int i = low,v = low,w = mid + 1; i <= high; i++) {
        if(w > high || v <= mid && lessThanOrEqual(src[v],src[w])) {
```

```
            dest[i] = src[v++];
        }else {
            dest[i] = src[w++];
        }
    }
}
```

小结

这里我们对以上两种方法进行一个分析，以上两个方法是我们对于数据进行排序的方法常用方法，其适用于规模比较大的数据的排序，但是其对于本道题并不是非常使用，原因如下：

1. 因为其数据量太多，递归太深了，其很有可能产生栈溢出的问题，
2. 数据太多，数组的长度太长了，很大概率会导致 OOM 问题。

综上所示，以上两个方法对于这两道题并不是非常实用，所以，在此基础上演变出第三种解决方法。

4. 查找两个大文件共同的 URL：分治加统计

题目要求

给定 a、b 两个文件，各存放 50 亿个 URL，每个 URL 各占 64B，找出 a、b 两个文件共同的 URL。内存限制是 4G。

分析

每个 URL 占 64B，那么 50 亿个 URL占用的空间大小约为 320GB。

5,000,000,000 * 64B ≈ 320GB

由于内存大小只有 4G，因此，不可能一次性把所有 URL 加载到内存中处理。

可以采用分治策略，也就是把一个文件中的 URL 按照某个特征划分为多个小文件，使得每个小文件大小不超过 4G，这样就可以把这个小文件读到内存中进行处理了。

首先遍历文件a，对遍历到的 URL 进行哈希取余 hash(URL) % 1000，根据计算结果把遍历到的 URL 存储到 a0, a1,a2, ..., a999，这样每个大小约为 300MB。使用同样的方法遍历文件 b，把文件 b 中的

URL 分别存储到文件 b0, b1, b2, ..., b999 中。这样处理过后，所有可能相同的 URL 都在对应的小文件中，即 a0 对应 b0, ..., a999 对应 b999，不对应的小文件不可能有相同的 URL。那么接下来，我们只需要求出这 1000 对小文件中相同的 URL 就好了。

接着遍历 ai(i∈[0,999])，把 URL 存储到一个 HashSet 集合中。然后遍历 bi 中每个 URL，看在 HashSet 集合中是否存在，若存在，说明这就是共同的 URL，可以把这个 URL 保存到一个单独的文件中。

总结

1. 分而治之，进行哈希取余
2. 对每个子文件进行 HashSet 统计

5. 海量数据寻找中位数：分治法

题目要求

给定100亿个无符号的乱序的整数序列，如何求出这100亿个数的中位数（中位数指的是排序后最中间那个数），内存只有**512M**

分析

中位数问题可以看做一个统计问题，而不是排序问题，无符号整数大小为4B，则能表示的数的范围为为0 ~ 2^32 – 1（40亿），如果没有限制内存大小，则可以用一个2^32（4GB）大小的数组（也叫做桶）来保存100亿个数中每个无符号数出现的次数。遍历这100亿个数，当元素值等于桶元素索引时，桶元素的值加1。当统计完100亿个数以后，则从索引为0的值开始累加桶的元素值，当累加值等于50亿时，这个值对应的索引为中位数。时间复杂度为O(n)。

因为题目要求内存限制512M，所以上述解法不合适。

下面分享另一种解法（**分治法**的思想）

如果100亿个数字保存在一个大文件中，可以依次读一部分文件到内存(不超过内存限制)，将每个数字用二进制表示，比较二进制的最高位(第32位，符号位，0是正，1是负)。

如果数字的最高位为0，则将这个数字写入 file_0文件中；如果最高位为 1，则将该数字写入file_1文件中。 从而将100亿个数字分成了两个文件。

假设 file_0文件中有 60亿 个数字，file_1文件中有 40亿 个数字。那么中位数就在 file_0 文件中，并且是 file_0 文件中所有数字排序之后的第 10亿 个数字。因为file_1中的数都是负数，file_0中的数都是正数，也即这里一共只有40亿个负数，那么排序之后的第50亿个数一定位于file_0中。

现在，我们只需要处理 file_0 文件了，不需要再考虑file_1文件。对于 file_0 文件，同样采取上面的措施处理：将file_0文件依次读一部分到内存(不超过内存限制)，将每个数字用二进制表示，比较二进制的 次高位（第31位），如果数字的次高位为0，写入file_0_0文件中；如果次高位为1，写入file_0_1文件 中。

现假设 file_0_0文件中有30亿个数字，file_0_1中也有30亿个数字，则中位数就是：file_0_0文件中的数字从小到大排序之后的第10亿个数字。

抛弃file_0_1文件，继续对 file_0_0文件 根据 次次高位(第30位) 划分，假设此次划分的两个文件为：file_0_0_0中有5亿个数字，file_0_0_1中有25亿个数字，那么中位数就是 file_0_0_1文件中的所有数字排序之后的 第 5亿 个数。

按照上述思路，直到划分的文件可直接加载进内存时，就可以直接对数字进行快速排序，找出中位数了。

6. 如何查询最热门的查询串：前缀树法

题目描述

搜索引擎会通过日志文件把用户每次检索使用的所有查询串都记录下来，每个查询床的长度不超过 255 字节。

假设目前有 1000w 个记录（这些查询串的重复度比较高，虽然总数是 1000w，但如果除去重复后，则不超过 300w 个）。请统计最热门的 10 个查询串，要求使用的内存不能超过 1G。（一个查询串的重复度越高，说明查询它的用户越多，也就越热门。）

方法一：分治法

分治法依然是一个非常实用的方法。

划分为多个小文件，保证单个小文件中的字符串能被直接加载到内存中处理，然后求出每个文件中出现次数最多的 10 个字符串；最后通过一个小顶堆统计出所有文件中出现最多的 10 个字符串。

方法可行，但不是最好，下面介绍其他方法。

方法二：HashMap 法

虽然字符串总数比较多，但去重后不超过 300w，因此，可以考虑把所有字符串及出现次数保存在一个 HashMap 中，所占用的空间为 300w*(255+4)≈777M（其中，4表示整数占用的4个字节）。由此可见，1G 的内存空间完全够用。

思路如下：

首先，遍历字符串，若不在 map 中，直接存入 map，value 记为 1；若在 map 中，则把对应的 value 加 1，这一步时间复杂度 O(N)。

接着遍历 map，构建一个 10 个元素的小顶堆，若遍历到的字符串的出现次数大于堆顶字符串的出现次数，则进行替换，并将堆调整为小顶堆。

遍历结束后，堆中 10 个字符串就是出现次数最多的字符串。这一步时间复杂度 O(Nlog10)。

方法三：前缀树法

方法二使用了 HashMap 来统计次数，当这些字符串有大量相同前缀时，可以考虑使用前缀树来统计字符串出现的次数，树的结点保存字符串出现次数，0 表示没有出现。

思路如下：

在遍历字符串时，在前缀树中查找，如果找到，则把结点中保存的字符串次数加 1，否则为这个字符串构建新结点，构建完成后把叶子结点中字符串的出现次数置为 1。

最后依然使用小顶堆来对字符串的出现次数进行排序。

方法总结

前缀树经常被用来统计字符串的出现次数。它的另外一个大的用途是字符串查找，判断是否有重复的字符串等。

7. 如何找出排名前 500 的数：堆排序

题目描述

有 1w 个数组，每个数组有 500 个元素，并且有序排列。如何在这 10000*500 个数中找出前 500 的数?

方法 1：归并排序

题目中每个数组是排好序的，可以使用**归并**的方法。

先将第1个和第2个归并，得到500个数据。然后再将结果和第3个数组归并，得到500个数据，以此类推，直到最后找出前500个的数。

方法2：堆排序

对于这种topk问题，更常用的方法是使用**堆排序**。

对本题而言，假设数组降序排列，可以采用以下方法：

首先建立大顶堆，堆的大小为数组的个数，即为10000 ，把每个数组最大的值存到堆中。

接着删除堆顶元素，保存到另一个大小为 500 的数组中，然后向大顶堆插入删除的元素所在数组的下一个元素。

重复上面的步骤，直到删除完第 500 个元素，也即找出了最大的前 500 个数。

示例代码如下：

```java
import lombok.Data;

import java.util.Arrays;
import java.util.PriorityQueue;

@Data
public class DataWithSource implements Comparable<DataWithSource> {
    /**
     * 数值
     */
    private int value;

    /**
     * 记录数值来源的数组
     */
    private int source;

    /**
     * 记录数值在数组中的索引
     */
    private int index;

    public DataWithSource(int value, int source, int index) {
        this.value = value;
        this.source = source;
        this.index = index;
    }

    /**
     *
     * 由于 PriorityQueue 使用小顶堆来实现，这里通过修改
     * 两个整数的比较逻辑来让 PriorityQueue 变成大顶堆
     */
    @Override
    public int compareTo(DataWithSource o) {
        return Integer.compare(o.getValue(), this.value);
    }
}


class Test {
    public static int[] getTop(int[][] data) {
        int rowSize = data.length;
        int columnSize = data[0].length;

```

```
        // 创建一个columnSize大小的数组，存放结果
        int[] result = new int[columnSize];

        PriorityQueue<DataWithSource> maxHeap = new PriorityQueue<>();
        for (int i = 0; i < rowSize; ++i) {
            // 将每个数组的最大一个元素放入堆中
            DataWithSource d = new DataWithSource(data[i][0], i, 0);
            maxHeap.add(d);
        }

        int num = 0;
        while (num < columnSize) {
            // 删除堆顶元素
            DataWithSource d = maxHeap.poll();
            result[num++] = d.getValue();
            if (num >= columnSize) {
                break;
            }

            d.setValue(data[d.getSource()][d.getIndex() + 1]);
            d.setIndex(d.getIndex() + 1);
            maxHeap.add(d);
        }
        return result;

    }

    public static void main(String[] args) {
        int[][] data = {
                {29, 17, 14, 2, 1},
                {19, 17, 16, 15, 6},
                {30, 25, 20, 14, 5},
        };

        int[] top = getTop(data);
        System.out.println(Arrays.toString(top)); // [30, 29, 25, 20, 19]
    }
}
```

8. 如何按照查询频率排序：分治法 / 哈希

题目描述

有 10 个文件，每个文件大小为 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求按照 query 的频度排序。

解答思路

如果 query 的重复度比较大，可以考虑一次性把所有 query 读入内存中处理；如果 query 的重复率不高，那么可用内存不足以容纳所有的 query，这时候就需要采用分治法或其他的方法来解决。

方法一：HashMap 法

如果 query 重复率高，说明不同 query 总数比较小，可以考虑把所有的 query 都加载到内存中的 HashMap 中。接着就可以按照 query 出现的次数进行排序。

方法二：分治法

分治法需要根据数据量大小以及可用内存的大小来确定问题划分的规模。对于这道题，可以顺序遍历 10 个文件中的 query，通过 Hash 函数 hash(query) % 10 把这些 query 划分到 10 个小文件中。之后对每个小文件使用 HashMap 统计 query 出现次数，根据次数排序并写入到零外一个单独文件中。

接着对所有文件按照 query 的次数进行排序，这里可以使用归并排序（由于无法把所有 query 都读入内存，因此需要使用外排序）。

方法总结

- 内存若够，直接读入进行排序；
- 内存不够，先划分为小文件，小文件排好序后，整理使用外排序进行归并。

9. 用 4 KB 的内存寻找重复元素

本题是非常典型的海量数据处理的问题，使用的是位运算结果。

题目描述

给定一个数组，包含从1到N的整数，N最大为32000，数组可能还有重复值，且N的取值不定，若只有4KB的内存可用，该如何打印数组中所有重复元素。

解决方案

这道题本身是一道海量数据的热身题，如果去掉**“只有4KB”**的要求，我们可以先创建一个大小为N的数组，然后将这些数据放进来，但是整数最大为 32000 。如果直接采用数组存，则应该需要 **32000*4B=128KB** 的空间，而题目有 **4KB** 的内存限制，我们就必须先解决该如何存放的问题。

如果只有4KB的空间，那么只能寻址8*4*2^10个比特，这个值比32000要大的，因此我们可以创建32000比特的位向量(比特数组)，其中一个比特位置就代表一个整数。
利用这个位向量，就可以遍历访问整个数组。如果发现数组元素是v，那么就将位置为v的设置为1，碰到重复元素，就输出一下。

```java
public class FindDuplicatesIn32000 {
    public void checkDuplicates(int[] array) {
        BitSet bs = new BitSet(32000);
        for (int i = 0; i < array.length; i++) {
            int num = array[i];
            int num0 = num - 1;
            if (bs.get(num0)) {
                System.out.println(num);
            } else {
                bs.set(num0);
            }
        }
    }
    class BitSet {
        int[] bitset;

        public BitSet(int size) {
            this.bitset = new int[size >> 5];
        }

        boolean get(int pos) {
            int wordNumber = (pos >> 5);//除以32
            int bitNumber = (pos & 0x1F);//除以32
            return (bitset[wordNumber] & (1 << bitNumber)) != 0;
        }

        void set(int pos) {
            int wordNumber = (pos >> 5);//除以32
            int bitNumber = (pos & 0x1F);//除以32
            bitset[wordNumber] |= 1 << bitNumber;
        }
    }
}
```

使用 C 语言实现，要复杂一点，接下来来看一下 Demo 例子：

Java

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *bitset;
} BitSet;

BitSet *createBitSet(int size) {
    BitSet *bs = (BitSet *)malloc(sizeof(BitSet));
    bs->bitset = (int *)calloc(size / 32 + 1, sizeof(int));
    return bs;
}

int get(BitSet *bs, int pos) {
    int wordNumber = pos / 32;
    int bitNumber = pos % 32;
    return (bs->bitset[wordNumber] >> bitNumber) & 1;
}

void set(BitSet *bs, int pos) {
    int wordNumber = pos / 32;
    int bitNumber = pos % 32;
    bs->bitset[wordNumber] |= 1 << bitNumber;
}

void checkDuplicates(int *array, int length) {
    BitSet *bs = createBitSet(320000);
    for (int i = 0; i < length; i++) {
        int num = array[i];
        int num0 = num - 1;
        if (get(bs, num0)) {
            printf("%d\n", num);
        } else {
            set(bs, num0);
        }
    }
    free(bs->bitset);
    free(bs);
}

int main() {
    int array[] = {1, 2, 3, 4, 4, 5, 5};
    int length = sizeof(array) / sizeof(int);

    checkDuplicates(array, length);
```

```
46
47          return 0;
48      }
```

Python Java

```python
def check_duplicates(array):
    bitset = [0] * (32000 // 32)

    for num in array:
        num0 = num - 1
        if (bitset[num0 // 32] >> (num0 % 32)) & 1:
            print(num)
        else:
            bitset[num0 // 32] |= 1 << (num0 % 32)
```

10. 从 40 亿中产生一个不存在的整数

题目描述

给定一个输入文件，包含 40 亿 个非负整数，请设计一个算法，产生一个不存在该文件中的整数，假设你有 1 GB 的内存来完成这项任务，你会怎么实现，然后再难一点，只用 10 MB 的内存呢?

位图法

如果用哈希表来保存出现过的数字，如果 40 亿个数字都不同，那么哈希表的记录数为 40 亿条，存一个 32 位整数需要 4B，所以最差情况下需要 40 亿*4B=160 亿字节，大约需要16GB 的空间，这是不符合要求的。
40 亿*4B=160 亿字节，大约需要 16 GB

40 亿/8 字节=5亿字节，大约0.5GB的数组就可以存下40亿个。

如果数据量很大，采用位方式（俗称位图）存储数据是常用的思路，那位图如果存储元素呢？我们可以使用 Bit Map 的方式来表示数出现的情况。具体来说，是申请一个长度为 4 294 967 295 的 bit 类型的数组 bitArr（存储boolean 类型），bitArr 上面的每个位置只可以表示 0 或1 状态。8 个bit 为 1B，所以长度为 4 294 967 295 的 bit 类型的数组占用 500MB 空间，这就满足题目给定的要求了。

那怎么使用这个 bitArr 数组呢？就是遍历这 40 亿个无符号数，遇到所有的数时，就把 bitArr 相应位置的值设置为 1。例如，遇到 1000，就把bitArr[1000]设置为 1。
遍历完成后，再依次遍历 bitArr，看看哪个位置上的值没被设置为 1，这个数就不在 40 亿个数中。例如，发现 bitArr[8001]==0，那么 8001 就是没出现过的数，遍历完 bitArr 之后，所有没出现的数就都找出来了。

位存储的核心是：我们存储的并不是这40亿个数据本身，而是其对应的位置。这一点明白的话，整个问题就迎刃而解了。

使用 10 MB 数据来存储：分块法 + 位图法

如果现在只有 10 MB 的内存，这个时候位图也没办法搞定了，我们只能另寻他发，这里使用我们最常用的分块实现，通过时间换空间的方式来，使用两次遍历来搞定。

40亿个数 需要500MB的空间，那如果只有10MB的空间，至少需要50个块才可以。

一般来说，我们划分都是使用2的整数倍，因此划分成64个块是合理的。
首先，将0~4 294 967 295(2^32) 这个范围是可以平均分成 64 个区间的，每个区间是 67 108 864 个数，例如：

- 第0 区间（0~67 108 863）
- 第 1 区间（67 108 864~134 217 728）
- 第 i 区间（67 108 864´i~67 108 864´(i+1)–1），
- ……，
- 第 63 区间（4 227 858 432~4 294 967 295）。

因为一共只有 40 亿个数，所以，如果统计落在每一个区间上的数有多少，肯定有至少一个区间上的计数少于67 108 864。利用这一点可以找出其中一个没出现过的数。具体过程是通过两次遍历来搞定：

第一次遍历，先申请一个长度为 64 的整型数组 countArr[0···63]，countArr[i] 用来统计区间 i 上的数有多少。遍历 40 亿个数，根据当前数是多少来决定哪一个区间上的计数增加。

例如，如果当前数是 3422 552090，其先对 67 108 864 取模，为 51，所以第 51 区间上的计数增加

countArr[51]++，遍历网 40 亿个数之后，遍历 countArr，必然会有某一个位置上的值（countArr[i]） 小于 67 108 864，表示第 i 区间上至少有一个数没出现过。我们肯定会找到至少一个这样的区间。

此时使用的内存就是countArr 的大小（64*4B），是非常小的。
假设找到第 37 区间上的计数小于 67 108 864，那么我们对这40亿个数据进行第二次遍历：

1. 申请长度为 67 108 864 的 bit map，这占用大约 8MB 的空间，记为 bitArr[0..67108863]。
2. 遍历这 40 亿个数，此时的遍历只关注落在第 37 区间上的数，记为 num（num满足num/67 108 864==37），其他区间的数全部忽略。
3. 如果步骤 2 的 num 在第 37 区间上，将 bitArr[num – 67108864*37]的值设置为 1，也就是只做第 37 区间上的数的 bitArr 映射。
4. 遍历完 40 亿个数之后，在 bitArr 上必然存在没被设置成 1 的位置，假设第 *i* 个位置上的值没设置成 1，那么 67 108 864´37+*i* 这个数就是一个没出现过的数。

总结：

这里总结一下第二种解决方案的过程：

1. 根据 10MB 的内存限制，确定统计区间的大小，就是第二次遍历时的 bitArr 大小。
2. 利用区间计数的方式，找到那个计数不足的区间，这个区间上肯定有没出现的数。
3. 对这个区间上的数做 bit map 映射，再遍历bit map，找到一个没出现的数即可。

疑问点：分块为什么是 64 块?

在上面的例子中，我们看到采用两次遍历，第一次将数据分成64块刚好解决问题。那我们为什么不是128块、32块、16块或者其他类型呢?
这里主要是要保证第二次遍历时每个块都能放进这10MB的空间中。2^23<10MB<2^24,而2^23=8388608大约为8MB，也就说我们一次的分块大小只能为8MB左右。在上面我们也看到了，第二次遍历时如果分为64块，刚好满足要求。

所以在这里我们需要最少分成 64 块，当然如果分成 128块、256块等也是可以的。

11. 使用 2 GB 内存在 20 亿个整数中找出出现次数最多的数

题目要求

有一个包含 20 亿个全是 32 位整数的大文件，限制内存为 2 GB，在其中找到出现次数最多的数。

想要在很多整数中找到出现次数最多的数，通常的做法是使用哈希表对出现的每一个数做词频统计，哈希表的 key 是某一个整数，value 是这个数出现的次数。就本题来说，一共有 20 亿个数，哪怕只是一个数出现了 20 亿次，用 32 位的整数也可以表示其出现的次数而不会产生溢出，所以哈希表的 key 需要占用 4B，value 也是 4B。那么哈希表的一条记录（key,value）需要占用 8B，当哈希表记录数为 2 亿个时，需要至少 1.6GB 的内存。

方法一：直接哈希

如果 20 亿个数中不同的数超过 2 亿种，最极端的情况是 20 亿个数都不同，那么在哈希表中可能需要产生 20 亿条记录，这样内存会不够用，所以一次性用哈希表统计 20 亿个数的办法是有很大风险的。

方法二：分而治之

把包含 20 亿个数的大文件用哈希函数分成 16 个小文件，根据哈希函数的性质，同一种数不可能被散列到不同的小文件上，同时每个小文件中不同的数一定不会大于 2 亿种， 假设哈希函数足够优秀。然后对每一个小文件用哈希表来统计其中每种数出现的次数，这样我们就得到了 16 个小文件中各自出现次数最多的数，还有各自的次数统计。接下来只要选出这16 个小文件各自的第一名中谁出现的次数最多即可。

把一个大的集合通过哈希函数分配到多台机器中，或者分配到多个文件里面，这个技巧是处理大数据面试题时最常用的技巧之一。但是到底分配到多少台机器、分配到多少个文件，在解题时一定要确定下来。可能是在与面试官沟通的过程中由面试官指定，也可能是根据具体的限制来确定，比如本题确定分成 16 个文件，就是根据内存限制 2GB 的条件来确定的。

12. 从 100 亿个 URL 中查找的问题

题目要求

有一个包含 100 亿个 URL 的大文件，假设每个 URL 占用 64B，请找出其中所有重复的 URL。

解决方法

本题的解法是解决大数据问题的一种常规方法：把大文件通过哈希函数分配到机器， 或者通过哈希函数把大文件拆成小文件，一直进行这种划分，直到划分的结果满足资源限制的要求。首先，要向面试官询问在资源上的限制有哪些，包括内存、计算时间等要求。在明确了限制要求之后，可以将每条 URL 通过哈希函数分配到若干台机器或者拆分成若干个小文件， 这里的“若干”由具体的资源限制来计算出精确的数量。

例如，将 100 亿字节的大文件通过哈希函数分配到 100 台机器上，然后每一台机器分别统计分给自己的 URL 中是否有重复的 URL，同时哈希函数的性质决定了同一条 URL 不可能分给不同的机器；或者在单机上将大文件通过哈希函数拆成 1000 个小文件，对每一个小文件再利用哈希表遍历，找出重复的 URL；还可以在分给机器或拆完文件之后进行排序，排序过后再看是否有重复的 URL 出现。总之，牢记一点，很多大数据问题都离不开分流，要么是用哈希函数把大文件的内容分配给不同的机器，要么是用哈希函数把大文件拆成小文件，然后处理每一个小数量的集合。

13. 求出每天热门 100 词、

题目要求

某搜索公司一天的用户搜索词汇是海量的（百亿数据量），请设计一种求出每天热门 Top 100 词汇的可行办法。

解题方法：分流 + 哈希

这道题和上面那道题一样，都是采用分流的思路来处理，把包含百亿数据量的词汇文件分流道不同的机器上，具体多少台机器由面试官规定或者由更多的限制来决定。对每一台机器来说，如果分到的数据量依然很大，比如，内存不够或存在其他问题，可以再用哈希函数把每台机器的分流文件拆成更小的文件处理。处理每一个小文件的时候，通过哈希表统计每种词及其词频，哈希表记录建立完成后，再遍历哈希表，遍历哈希表的过程中使用大小为 100 的小根堆来选出每一个小文件的 Top 100（整体未排序的 Top 100）。每一个小文件都有自己词频的小根堆（整体未排序的 Top 100），将小根堆里的词按照词频排序，就得到了每个小文件的排序后 Top 100。然后把各个小文件排序后的 Top 100 进行外排序或者继续利用小根堆，就可以选出每台机器上的 Top100。不同机器之间的 Top 100 再进行外排序或者继续利用小根堆，最终求出整个百亿数据量中的 Top 100。对于 Top K 的问题，除用哈希函数分流和用哈希表做词频统计之外，还经常用堆结构和外排序的手段进行处理。

14. 40 亿个非负整数中找到出现两次的数

题目要求

32 位无符号整数的范围是 0～4 294 967 295，现在有 40 亿个无符号整数，可以使用最多 1GB 的内存，找出所有出现了两次的数。

解题方法

本题和之前 40 亿个数字寻找重复的元素其实差不多，可以说是他的进阶，这道题目把出现次数限制在两次。

首先，可以用 bit map 的方式来表示数出现的情况。具体地说，是申请一个长度为4 294 967 295x2 的bit 类型的数组bitArr，用 2 个位置表示一个数出现的词频，1B 占用 8 个bit， 所以长度为 4 294 967 295x2 的 bit 类型的数组占用 1GB 空间。怎么使用这个 bitArr 数组呢？遍历这 40 亿个无符号数，如果初次遇到 num，就把bitArr[num*2 + 1]和 bitArr[num*2]设置为 01， 如果第二次遇到 num，就把bitArr[num*2+1]和bitArr[num*2]设置为 10，如果第三次遇到 num， 就把bitArr[num*2+1]和bitArr[num*2]设置为 11。以后再遇到 num，发现此时 bitArr[num*2+1]和 bitArr[num*2]已经被设置为 11，就不再做任何设置。遍历完成后，再依次遍历 bitArr，如果发现 bitArr[i*2+1]和bitArr[i*2]设置为 10，那么 *i* 就是出现了两次的数。

15. 对 20 GB 文件进行排序

题目要求

假设你有一个20GB的文件，每行一个字符串，请说明如何对这个文件进行排序?

解决方法：外部排序

这里给出大小是20GB，其实面试官就在暗示你不要将所有的文件都装入到内存里，因此我们只能将文件划分成一些块，每块大小是xMB，x就是可用内存的大小，例如1GB一块，那我们就可以将文件分为20块。我们先对每块进行排序，然后再逐步合并。这时候我们可以使用两两归并，也可以使用堆排序策略将其逐步合并成一个。

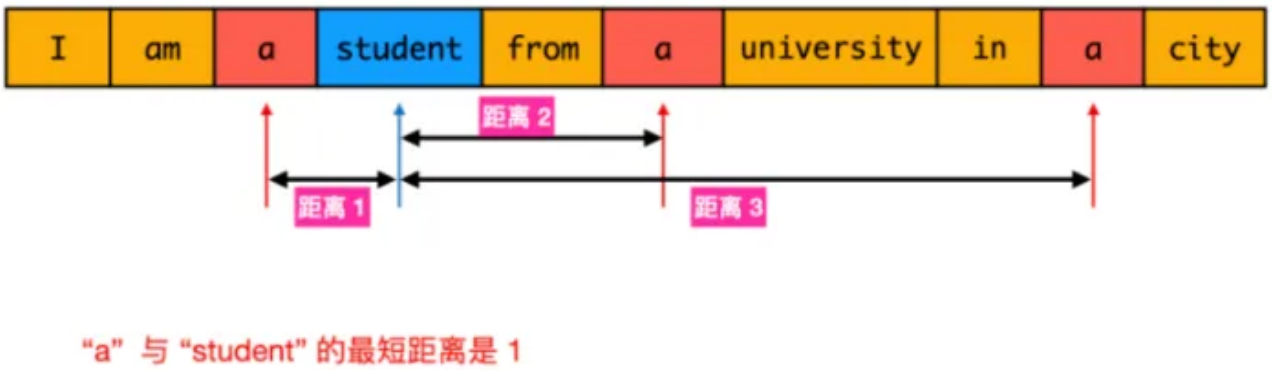
16. 超大文本中搜索两个单词的最短距离

题目要求

有个超大文本文件，内部是很多单词组成的，现在给定两个单词，请你找出这两个单词在这个文件中的最小距离，也就是像个几个单词。你有办法在O(n)时间里完成搜索操作吗？方法的空间复杂度如何。

解题方法

这个题咋看很简单，遍历一下，找到这两个单词w1和w2的位置然后比较一下就可以了，然而这里的w1可能在很多位置出现，而w2也会在很多位置出现，如下图：



“a” 与 “student” 的最短距离是 1

这时候如何比较寻找哪两个是最小距离呢?

最直观的做法是遍历数组 words，对于数组中的每个word1，遍历数组words 找到每个word2并计算距离。该做法在最坏情况下的时间复杂度是 O(n^2)，需要优化。

本题我们少不了遍历一次数组，找到所有word1 和word2出现的位置，但是为了方便比较，我们可以将其放到一个数组里，例如：

```Java
listA:{1,2,9,15,25}
listB:{4,10,19}
合并成
list:{1a,2a,4b,9a,10b,15a,19b,25a}
```

合并成一个之后更方便查找，数字表示出现的位置，后面一个元素表示元素是什么。然后一边遍历一边比较就可以了。

但是对于超大文本，如果文本太大那这个list可能溢出。如果继续观察，我们会发现其实不用单独构造list，从左到右遍历数组words，当遍历到 word1时，如果已经遍历的单词中存在word2 ，为了计算最短距离，应该取最后一个已经遍历到的 word2所在的下标，计算和当前下标的距离。同理，当遍历到word2时，应该取最后一个已经遍历到的word1所在的下标，计算和当前下标的距离。
基于上述分析，可以遍历数组一次得到最短距离，将时间复杂度降低到O(n)。用index1和index2分别表示数组words 已经遍历的单词中的最后一个word1的下标和最后一个word2的下标，初始时index1 =index2=一1。遍历数组words，当遇到word2时，执行如下操作：

- 如果遇到word1 ，则将index1更新为当前下标；如果遇到word2，则将index2更新为当前下标。
- 如果index1和index2都非负，则计算两个下标的距离 |index1一index2 |，并用该距离更新最短距离。

遍历结束之后即可得到word1和word2的最短距离。
进阶问题如果寻找过程在这个文件中会重复多次，而每次寻找的单词不同，则可以维护一个哈希表记录每个单词的下标列表。遍历一次文件，按照下标递增顺序得到每个单词在文件中出现的所有下标。在寻找单词时，只要得到两个单词的下标列表，使用双指针遍历两个下标链表，即可得到两个单词的最短距离。

17. 从 10 亿个数字中寻找最小的 100 万个数字

题目要求

设计一个算法，给定 10 亿个数字，从中找出最小的 100 万个数字，假设计算机内存足够容纳全部 10 亿个数字。

本题有三种常用的方法,接下来来一一进行讲解：

方法一：先排序所有元素

先排序所有元素，然后取出前 100 完个数，该方法的时间复杂度为 O（nlogn）。很明显这种方式的时间代价是 O(nlogn)。很明显对于10亿级别的数据，这么做时间和空间代价太高。

方法二：选择排序

第二种方式是采用选择排序的方式，首先遍历10亿个数字找最小，然后再遍历一次找第二小 ，然后再一次找第三小，直到找到第100万个。很明显这种方式的时间代价是O(nm)也就是要执行10亿*100万次，这个效率一般的服务器都达不到。

方法三：大顶堆

这里分享一个口诀，**查小用大堆，查大用小堆。**

首先，为前100万个数字创建一个大顶堆，最大元素位于堆顶。

然后，遍历整个序列，只有比堆顶元素小的才允许插入堆中，并删除原堆的最大元素。

之后继续遍历剩下的数字，最后剩下的就是最小的100万个。

采用这种方式，只需要遍历一次10亿个数字，还可以接受。更新堆的代价是O(nlogn),也勉强能够接受。堆占用的空间是100万*4，大约为4MB左右的空间就够了，因此也能接收。

如果数据量没有这么大，也是可以直接使用这三种方式的。

如果将10亿数字换成流数据，也可以使用堆来找，而且对于流数据，几乎只能用堆来做。

18. 大数据 Top K 问题常用套路

最后一道题，来对于 Top K 问题进行一个总结，对于海量数据的处理，经常会涉及到 topK 问题。在设计数据结构和算法的时候，主要需要考虑的应该是当前算法（包括数据结构）跟给定情境（比如数据量级、数据类型）的适配程度，和当前问题最核心的瓶颈（如降低时间复杂度，还是降低空间复杂度）是什么。

首先，我们来举几个常见的 topK 问题的例子：

1. 给定 100 个 int 数字，在其中找出最大的 10 个；
2. 给定 10 亿个 int 数字，在其中找出最大的 10 个（这 10 个数字可以无序）；
3. 给定 10 亿个 int 数字，在其中找出最大的 10 个（这 10 个数字依次排序）；
4. 给定 10 亿个不重复的 int 数字，在其中找出最大的 10 个；
5. 给定 10 个数组，每个数组中有 1 亿个 int 数字，在其中找出最大的 10 个；
6. 给定 10 亿个 string 类型的数字，在其中找出最大的 10 个（仅需要查 1 次）；
7. 给定 10 亿个 string 类型的数字，在其中找出最大的 k 个（需要反复多次查询，其中 k 是一个随机数字）。

上面这些问题看起来很相似，但是解决的方式却千差万别。稍有不慎，就可能使得 topK 问题成为系统的瓶颈。不过也不用太担心，接下来我会总结几种常见的解决思路，遇到问题的时候，大家把这些基础思路融会贯通并且杂糅组合，即可做到见招拆招。

堆排序法

这里说的是堆排序法，而不是快排或者希尔排序。虽然理论时间复杂度都是 O(nlogn)，但是堆排在做 topK 的时候有一个优势，就是可以维护一个仅包含 k 个数字的小顶堆（想清楚，为啥是小顶堆哦），当新加入的数字大于堆顶数字的时候，将堆顶元素剔除，并加入新的数字。

用 C++ 来说明，堆在 stl 中是 priority_queue（不是 set）。

> Java 中同样提供了 PriorityQueue 的数据结构。

Java

```java
int main() {
    const int topK = 3;
    vector<int> vec = {4,1,5,8,7,2,3,0,6,9};
    priority_queue<int, vector<int>, greater<>> pq;    // 小顶堆
    for (const auto& x : vec) {
        pq.push(x);
        if (pq.size() > topK) {
            // 如果超出个数，则弹出堆顶（最小的）数据
            pq.pop();
        }
    }

    while (!pq.empty()) {
        cout << pq.top() << endl;    // 输出依次为7,8,9
        pq.pop();
    }
    return 0;
}
```

类似快排法

快排大家都知道，针对 topK 问题，可以对快排进行改进。仅对部分数据进行递归计算。比如，在 100 个数字中，找最大的 10 个，第一次循环的时候，povit 被移动到了 80 的位置，则接下来仅需要在后面的 20 个数字中找最大的 10 个即可。

这样做的优势是，理论最优时间复杂度可以达到 O(n)，不过平均时间复杂度还是 O(nlogn)。需要说明的是，通过这种方式，找出来的最大的 k 个数字之间，是无序的。

```Java
int partition(vector<int>& arr, int begin, int end) {
    int left = begin;
    int right = end;
    int povit = arr[begin];

    while (left < right) {
        while (left < right && arr[right] >= povit) {right--;}
        while (left < right && arr[left] <= povit) {left++;}
        if (left < right) {swap(arr[left], arr[right]);}
    }

    swap(arr[begin], arr[left]);
    return left;
}

void partSort(vector<int>& arr, int begin, int end, int target) {
    if (begin >= end) {
        return;
    }

    int povit = partition(arr, begin, end);
    if (target < povit) {
        partSort(arr, begin, povit - 1, target);
    } else if (target > povit) {
        partSort(arr, povit + 1, end, target);
    }
}

vector<int> getMaxNumbers(vector<int>& arr, int k) {
    int size = (int)arr.size();
    // 把求最大的k个数，转换成求最小的size-k个数字
    int target = size - k;
    partSort(arr, 0, size - 1, target);
    vector<int> ret(arr.end() - k, arr.end());
    return ret;
}

int main() {
    vector<int> vec = {4,1,5,8,7,2,3,0,6,9};
    auto ret = getMaxNumbers(vec, 3);

    for (auto x : ret) {
        cout << x << endl;    // 输出7，8，9（理论上无序）
    }

```

```
        return 0;
    }
```

使用 Bitmap

有时候 topK 问题会遇到数据量过大，内存无法全部加载。这个时候，可以考虑将数据存放至 bitmap 中，方便查询。

比如，给出 10 个 int 类型的数据，分别是【13，12，11，1，2，3，4，5，6，7】，int 类型的数据每个占据 4 个字节，那这个数组就占据了 40 个字节。现在，把它们放到一个 16 个长度 bool 的 bitmap 中，结果就是【0，1，1，1，1，1，1，1，0，0，0，1，1，1，0，0】，在将空间占用降低至 4 字节的同时，也可以很方便的看出，最大的 3 个数字，分别是 11，12 和 13。

需要说明的是，bitmap 结合跳表一起使用往往有奇效。比如以上数据还可以记录成：从第 1 位开始，有连续 7 个 1；从第 11 位开始，有连续 3 个 1。这样做，空间复杂度又得到了进一步的降低。

这种做法的优势，当然是降低了空间复杂度。不过需要注意一点，bitmap 比较适合不重复且有范围（比如，数据均在 0 ～ 10 亿之间）的数据的查询。至于有重复数据的情况，可以考虑与 hash 等结构的混用。

使用 Hash

如果遇到了查询 string 类型数据的大小，可以考虑 hash 方法。

举个例子，10 个 string 数字
【"1001"，"23"，"1002"，"3003"，"2001"，"1111"，"65"，"834"，"5"，"987"】找最大的 3 个。我们先通过长度进行 hash，得到长度最大为 4，且有 5 个长度为 4 的 string。接下来再通过最高位值做 hash，发现有 1 个最高位为"3"的，1 个为"2"的，3 个为"1"的。接下来，可以通过再设计 hash 函数，或者是循环的方式，在 3 个最高位为"1"的 string 中找到最大的一个，即可找到 3 个最值大的数据。

这种方法比较适合网址或者电话号码的查询。缺点就是如果需要多次查询的话，需要多次计算 hash，并且需要根据实际情况设计多个 hash 函数。

字典树

字典树（trie）的具体结构和查询方式，不在这里赘述了，自行百度一下就有很多。这里主要说一下优缺点。

字典树的思想，还是通过前期建立索引信息，后期可以反复多次查询，并且后期增删数据也很方便。比较适合于需要反复多次查询的情况。

比如，反复多次查询字符序（例如：z>y>...>b>a）最大的 k 个 url 这种，使用字典树把数据存储一遍，就非常适合。既减少了空间复杂度，也加速了查询效率。

混合查询

以上几种方法，都是比较独立的方法。其实，在实际工作中，遇到更多的问题还是混合问题，这就需要我们对相关的内容，融会贯通并且做到活学活用。

我举个例子：我们的分布式服务跑在 10 台不同机器上，每台机器上部署的服务均被请求 10000 次，并且记录了个这 10000 次请求的耗时（耗时值为 int 数据），找出这 10*10000 次请求中，从高到低的找出耗时最大的 50 个。看看这个问题，很现实吧。我们试着用上面介绍的方法，组合一下来求解。

方法一：分流 + 快排

首先，对每台机器上的 10000 个做类似快排，找出每台机器上 top50 的耗时信息。此时，单机上的这 50 条数据是无序的。

然后，再将 10 台机器上的 50 条数据（共 500 条）放到一起，再做一次类似快排，找到最大的 50 个（此时应该这 50 个应该是无序的）。

最后，对这 50 个数据做快排，从而得到最终结果。

方法二：分流 + 堆排

首先通过堆排，分别找出 10 台机器上耗时最高的 50 个数据，此时的这 50 个数据，已经是从大到小有序的了。

然后，我们依次取出 10 台机器中，耗时最高的 5 条放入小顶堆中。

最后，遍历 10 台机器上的数据，每台机器从第 6 个数据开始往下循环，如果这个值比堆顶的数据大，

则抛掉堆顶数据并且把它加入，继续用下一个值进行同样比较。如果这个值比堆顶的值小，则结束当前循环，并且在下一台机器上做同样操作。

以上我介绍了两种方法，并不是为了说明哪种方法更好，或者时间复杂度更低。而是想说同样的事情有多种不同的解决方法，而且随着数据量的增加，可能会需要更多组合形式。在这个领域，数据决定了数据结构，数据结构决定了算法。