第一部分、十道海量数据处理面试题

1、海量日志数据，提取出某日访问百度次数最多的那个IP。

首先是这一天，并且是访问百度的日志中的IP取出来，逐个写入到一个大文件中。

注意到IP是32位的，最多有个2^32个IP。

同样可以采用映射的方法， 比如模1000，把整个大文件映射为1000个小文件，

再找出每个小文中出现频率最大的IP（可以采用hash_map进行频率统计，然后再找出频率最大 的几个）及相应的频率。

然后再在这1000个最大的IP中，找出那个频率最大的IP，即为所求。

或者如下阐述（雪域之鹰）：

算法思想：分而治之+Hash

1.IP地址最多有2^32=4G种取值情况，所以不能完全加载到内存中处理；

2.可以考虑采用“分而治之”的思想 ,按照IP地址的Hash(IP)%1024值 ,把海量IP日志分别存储到1024个小文件中。

这样，每个小文件最多包含4MB个IP地址；

3.对于每一个小文件，可以构建一个IP为key，出现次数为value的Hash map，同时记录当前出现次数最多的那个IP地址；

4.可以得到1024个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP；

2、搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

典型的Top K算法，还是在这篇文章里头有所阐述，详情请参见：十一、从头到尾彻底解析Hash表算法。

文中，给出的最终算法是：

第一步、先对这批海量数据预处理，在O（N）的时间内用Hash表完成**统计**（之前写成了排序，特此订正。July、2011.04.27）；

第二步、借助堆这个数据结构，找出Top K，时间复杂度为N‘logK。

即，借助堆结构，我们可以在log量级的时间内查找和调整/移动。因此，维护一个K(该题目中是10)大小的小根堆 ,然后遍历300万的Query ,分别 和根元素进行对比所以 ,我们最终的时间复杂度是：O（N） + N’*O（logK），（N为1000万，N’为300万）。ok，更多，详情，请参考原文。

或者：采用trie树，关键字域存该查询串出现的次数，没有出现为0。最后用10个元素的最小推来对出现频率进行排序。

3、有一个1G大小的一个文件 ,里面每一行是一个词 ,词的大小不超过16字节 ,内存限制大小是1M。

返回频数最高的100个词。

方案：顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。

这样每个文件大概是200k左右。

如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。

对每个小文件，统计每个文件中出现的词以及相应的频率（**可以采用trie树/hash_map等**），并取出出现频率最大的100个词(可以用含100个结点的最小堆）,并把100个词及相应的频率存入文件，这样又得到了5000个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

//有可能每一行都不一样，所以要划分文件，否则内存不够用。

4、有10个文件，每个文件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。要求你按照query的频度排序。

还是典型的TOP K算法，解决方案如下：

方案1：

顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件（记为）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。

找一台内存在2G左右的机器，依次对用hash_map(query, query_count)来统计每个query出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的 query_cout输出到文件中。这样得到了10个排好序的文件（记为）。

对这10个文件进行归并排序（内排序与外排序相结合）。

方案2：

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

方案3：

与方案1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

5、 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？

方案1：可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件（记为a0,a1,...,a999）中。这样每个小文件的大约为300M。

遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为b0,b1,...,b999）。这样处理后，

所有可能相同的url都在对应的小文件（a0vsb0,a1vsb1,...,a999vsb999）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。
求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash_set中，如果是，那么就是共同的url，存到文件里面就可以了。

方案2：如果允许有一定的错误率，可以使用**Bloom filter**，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。
Bloom filter日后会在本BLOG内详细阐述。

6、在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。
方案1：采用**2-Bitmap**（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 * 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看 bitmap，把对应位是01的整数输出即可。

方案2：也可采用与第1题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

7、腾讯面试题：40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？
与上第6题类似，我的第一反应时快速排序+二分查找。以下是其它更好的方法：
方案1：申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。
（unsigned int 能表示的最大数是4294967295），全部读入内存是4294967295*4B=15G，但此时不要读入数据，在内存中每个数设置一个表示位来代表数存在不存在即可，共需要4294967295b=511M的内存，

方案2：这个问题在《编程珠玑》里有很好的描述，大家可以参考下面的思路，探讨一下：
又因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；
这里我们把40亿个数中的每一个用32位的二进制来表示
假设这40亿个数开始放在一个文件中。
然后将这40亿个数分成两类:
1.最高位为0
2.最高位为1

并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；
与要查找的数的最高位比较并接着进入相应的文件再查找
再然后把这个文件为又分成两类:
1.次最高位为0
2.次最高位为1
并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）；
与要查找的数的次最高位比较并接着进入相应的文件再查找。
.....
以此类推，就可以找到了,而且时间复杂度为O(logn)，方案2完。

附：这里，再简单介绍下，**位图方法**：
使用位图法判断整形数组是否存在重复
判断集合中存在重复是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。
位图法比较适合于这种情况，它的做法是按照集合中最大元素max创建一个长度为max+1的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上 1，如遇到5就给新数组的第六个元素置1，这样下次再遇到5想置位时发现新数组的第六个元素已经是1了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时置零其后置一的做法类似于位图的处理方法故称位图法。它的运算次数最坏的情况为2N。如果已知数组的最大值即能事先给新数组定长的话效率还能提高一倍。欢迎，有更好的思路，或方法，共同交流。

8、怎么在海量数据中找出重复次数最多的一个？

方案1：先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

9、上千万或上亿数据（有重复），统计其中出现次数最多的钱N个数据。
方案1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用第2题提到的堆机制完成。

10、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。
方案1：这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是O(n*le)（le表示单词的平准长度）。然后是找出出现最频繁的前10 个词，可以用堆来实现，前面的题中已经讲到了，时

间复杂度是O(n*lg10)。所以总的时间复杂度，是O(n*le)与O(n*lg10)中较大的哪一 个。

附、100w个数中找出最大的100个数。

方案1：在前面的题中，我们已经提到了，用一个含100个元素的最小堆完成。复杂度为O(100w*lg100)。

方案2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比100多的时候，采用传统排序算法排序，取前100个。复杂度为O(100w*100)。

方案3：采用局部淘汰法。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的 要大，那么把这个最小的元素删除，并把x利用插入排序的思想 ,插入到序列L中。依次循环 ,知道扫描了所有的元素。复杂度为O(100w*100)。

致谢：http://www.cnblogs.com/youwang/。

第二部分、十个海量数据处理方法大总结

ok，看了上面这么多的面试题，是否有点头晕。是的，需要一个总结。接下来，本文将简单总结下一些处理海量数据问题的常见方法，**而日后，本BLOG内会具体阐述这些方法。**

下面的方法全部来自http://hi.baidu.com/yanxionglu/blog/博客，对海量数据的处理方法进行了一个一般性的总结，当然这些方法可能并不能完全覆盖所有的问题，但是这样的一些方法也基本可以处理绝大多数遇到的问题。下面的一些问题基本直接来源于公司的面试笔试题目，方法不一定最优，如果你有更好的处理方法，欢迎讨论。

一、Bloom filter

适用范围：可以用来实现数据字典，进行数据的判重，或者集合求交集

基本原理及要点：

对于原理来说很简单，位数组+k个独立hash函数。将 hash函数对应的值的位数组置1，查找时如果发现所有hash函数对应位都是1说明存在，很明显这个过程并不保证查找的结果是100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是 counting Bloom filter，用一个counter数组代替位数组，就可以支持删除了。

还有一个比较重要的问题，如何根据输入元素个数n，确定位数组m的大小及hash函数个数。当hash函数个数k=(ln2)*(m/n)时错误率最小。在错误率不大于E的情况下，m至少要等于n*lg(1/E)才能表示任意n个元素的集合。但m还应该更大些，因为还要保证bit数组里至少一半为0，则m应该>=nlg(1/E)*lge 大概就是nlg(1/E)1.44倍(lg表示以2为底的对数)。

举个例子我们假设错误率为0.01，则此时m应大概是n的13倍。这样k大概是8个。

注意这里m与n的单位不同，m是bit为单位，而n则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多bit的。所以使用bloom filter内存上通常都是节省的。

扩展：

Bloom filter将集合中的元素映射到位数组中，用k（k为哈希函数个数）个映射位是否全1表示元素在不在这个集合中。Counting bloom filter（CBF）将位数组中的每一位扩展为一个counter，从

而支持了元素的删除操作。Spectral Bloom Filter（SBF）将其与集合元素的出现次数关联。SBF采用counter中的最小值来近似表示元素的出现频率。
问题实例：给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢？
根据这个问题我们来计算下内存的占用，4G=2^32大概是40亿*8大概是340 亿，n=50亿，如果按出错率0.01算需要的大概是650亿个bit。现在可用的是340亿，相差并不多，这样可能会使出错率上升些。另外如果这些 urlip是一一对应的，就可以转换成ip，则大大简单了。

二、Hashing

适用范围：快速查找，删除的基本数据结构，通常需要总数据量可以放入内存
基本原理及要点：
hash函数选择，针对字符串，整数，排列，具体相应的hash方法。
碰撞处理，一种是open hashing，也称为拉链法；另一种就是closed hashing，也称开地址法，opened addressing。
扩展：
d-left hashing中的d是多个的意思，我们先简化这个问题，看一看2-left hashing。2-left hashing指的是将一个哈希表分成长度相等的两半，分别叫做T1和T2，给T1和T2分别配备一个哈希函数，h1和h2。在存储一个新的key时 ,同 时用两个哈希函数进行计算 ,得出两个地址h1[key]和h2[key]。这时需要检查T1中的h1[key]位置和T2中的h2[key]位置，哪一个 位置已经存储的（有碰撞的）key比较多，然后将新key存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个key，就把新key 存储在左边的T1子表中，2-left也由此而来。在查找一个key时，必须进行两次hash，同时查找两个位置。
问题实例：
1).海量日志数据，提取出某日访问百度次数最多的那个IP。
IP的数目还是有限的，最多2^32个，所以可以考虑使用hash将ip直接存入内存，然后进行统计。

三、bit-map

适用范围：可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下
基本原理及要点：使用bit数组来表示某些元素是否存在，比如8位电话号码
扩展：bloom filter可以看做是对bit-map的扩展
问题实例：
1)已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。
8位最多99 999 999，大概需要99m个bit，大概10几m字节的内存即可。
2)2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。
将bit-map扩展一下，用2bit表示一个数即可，0表示未出现，1表示出现一次，2表示出现2次及以上。或者我们不用2bit来进行表示，我们用两个bit-map即可模拟实现这个2bit-map。

四、堆

适用范围：海量数据前n大，并且n比较小，堆可以放入内存

基本原理及要点：最大堆求前n小，最小堆求前n大。方法，比如求前n小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元素。这样最后得到的n个元素就是最小的n个。适合大数据量，求前n小，n的大小比较 小的情况，这样可以扫描一遍即可得到所有的前n元素，效率很高。

扩展：双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。

问题实例：

1)100w个数中找最大的前100个数。

用一个100个元素大小的最小堆即可。

五、双层桶划分—-其实本质上就是【分而治之】的思想，重在"分"的技巧上！

适用范围：第k大，中位数，不重复或重复的数字

基本原理及要点：因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子。

扩展：

问题实例：

1).2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

有点像鸽巢原理，整数个数为2^32,也就是，我们可以将这2^32个数，划分为2^8个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

2).5亿个int找它们的中位数。

这个例子比上面那个更明显。首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成2^24个区域，然后确定区域的第几大数，在将该区域分成2^20个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有2^20，就可以直接利用direct addr table进行统计了。

六、数据库索引

适用范围：大数据量的增删改查

基本原理及要点：利用数据的设计实现方法，对海量数据的增删改查进行处理。

七、倒排索引(Inverted index)

适用范围：搜索引擎，关键字查询

基本原理及要点：为何叫倒排索引？一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。
以英文为例，下面是要被索引的文本：
T0 = “it is what it is”
T1 = “what is it”
T2 = “it is a banana”
我们就能得到下面的反向文件索引：
“a”: {2}
“banana”: {2}
“is”: {0, 1, 2}
“it”: {0, 1, 2}
“what”: {0, 1}
检索的条件”what”,”is”和”it”将对应集合的交集。
正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。
在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档 指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

扩展：
问题实例：文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索。

八、外排序
适用范围：大数据的排序，去重
基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树
扩展：
问题实例：
1).有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16个字节，内存限制大小是1M。返回频数最高的100个词。
这个数据具有很明显的特点，词的大小为16个字节，但是内存只有1m做hash有些不够，所以可以用来排序。内存可以当输入缓冲区使用

九、trie树
适用范围：数据量大，重复多，但是数据种类小可以放入内存
基本原理及要点：实现方式，节点孩子的表示方式
扩展：压缩实现。

问题实例：
1).有10个文件，每个文件1G，每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序。
2).1000万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现？
3).寻找热门查询：查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个，每个不超过255字节。

十、分布式处理 mapreduce
适用范围：数据量大，但是数据种类小可以放入内存
基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。
扩展：
问题实例：
1).The canonical example application of MapReduce is a process to count the appearances of
each different word in a set of documents:
2).海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。
3).一共有N个机器，每个机器上有N个数。每个机器最多存O(N)个数并对它们操作。如何找到N^2个数的中数(median)？

经典问题分析
上千万or亿数据（有重复），统计其中出现次数最多的前N个数据,分两种情况：可一次读入内存，不可一次读入。
可用思路：trie树+堆，数据库索引，划分子集分别统计，hash，分布式计算，近似统计，外排序

所谓的是否能一次读入内存，实际上应该指去除重复后的数据量。如果去重后数据可以放入内存，我们可以为数据建立字典，比如通过 map，hashmap，trie，然后直接进行统计即可。当然在更新每条数据的出现次数的时候，我们可以利用一个堆来维护出现次数最多的前N个数据，当 然这样导致维护次数增加，不如完全统计后在求前N大效率高。

如果数据无法放入内存。一方面我们可以考虑上面的字典方法能否被改进以适应这种情形，可以做的改变就是将字典存放到硬盘上，而不是内存，这可以参考数据库的存储方法。
当然还有更好的方法，就是可以采用分布式计算，基本上就是map-reduce过程， 首先可以根据数据值或者把数据hash(md5)后的值，将数据按照范围划分到不同的机子，最好可以让数据划分后可以一次读入内存，这样不同的机子负责处 理各种的数值范围，实际上就是map。得到结果后，各个机子只需拿出各自的出现次数最多的前N个数据，然后汇总，选出所有的数据中出现次数最多的前N个数 据，这实际上就是reduce过程。

实际上可能想直接将数据均分到不同的机子上进行处理，这样是无法得到正确的解的。因为 一个数据可能被均分到不同的机子上，而另一个则可能完全聚集到一个机子上，同时还可能存在具有相同数目的数据。比如我们要找出现次数最多的前100个，我 们将1000万的数据分布到10台机器上，找到每台出现次数最多的前 100个，归并之后这样不能保证找到真正的第100个，因为比如出现次数最多的第100个可能有1万个，但是它被分到了10台机子，这样在每台上只有1千 个，假设这些机子排名在1000个之前的那些都是单独分布在一台机子上的，比如有1001个，这样本来具有1万个的这个就会被淘汰，即使我们让每台机子选 出出现次数最多的1000个再归并，仍然会出错，因为可能存在大量个数为1001个的发生聚集。因此不能将数据随便均分到不同机子上 ,而是要根据hash 后的值将它们映射到不同的机子上处理，让不同的机器处理一个数值范围。
而外排序的方法会消耗大量的IO，效率不会很高。而上面的分布式方法，也可以用于单机版本，也就是将总的数据根据值的范围，划分成多个不同的子文件，然后逐个处理。处理完毕之后再对这些单词的及其出现频率进行一个归并。实际上就可以利用一个外排序的归并过程。
另外还可以考虑近似计算，也就是我们可以通过结合自然语言属性，只将那些真正实际中出现最多的那些词作为一个字典，使得这个规模可以放入内存。
ok，以上有任何问题，欢迎指正。谢谢大家。本文完。

问题
某海量用户网站，用户拥有积分，积分可能会在使用过程中随时更新。现在要为该网站设计一种算法 ,在每次用户登录时显示其当前积分排名。用户最大规模为2亿 ;积分为非负整数 ,且小于100万。
PS: 据说这是迅雷的一道面试题，不过问题本身具有很强的真实性，所以本文打算按照真实场景来考虑，而不局限于面试题的理想环境。

存储结构
首先，我们用一张用户积分表user_score来保存用户的积分信息。
下面的算法会基于这个基本的表结构来进行。

算法1：简单SQL查询
首先，我们很容易想到用一条简单的SQL语句查询出积分大于该用户积分的用户数量：
select 1 + count(t2.uid) as rankfrom user_score t1, user_score t2where t1.uid = @uid and t2.score > t1.score
对于4号用户我们可以得到下面的结果：
算法特点
优点：简单，利用了SQL的功能，不需要复杂的查询逻辑，也不引入额外的存储结构，对小规模或

性能要求不高的应用不失为一种良好的解决方案。
缺点 :需要对user_score表进行全表扫描 ,还需要考虑到查询的同时若有积分更新会对表造成锁定，在海量数据规模和高并发的应用中，性能是无法接受的。

算法2：均匀分区设计

在许多应用中缓存是解决性能问题的重要途径，我们自然会想能不能把用户排名用Memcached缓存下来呢？不过再一想发现缓存似乎帮不上什么忙，因为用户排名是一个全局性的统计性指标，而并非用户的私有属性，其他用户的积分变化可能会马上影响到本用户的排名。然而，真实的应用中积分的变化其实也是有一定规律的，通常一个用户的积分不会突然暴增暴减，一般用户总是要在低分区混迹很长一段时间才会慢慢升入高分区，也就是说用户积分的分布总体说来是有区段的，我们进一步注意到高分区用户积分的细微变化其实对低分段用户的排名影响不大。于是，我们可以想到按积分区段进行统计的方法，引入一张分区积分表score_range：
表结构：
数据示例：
表示[from_score, to_score)区间有count个用户。若我们按每1000分划分一个区间则有[0, 1000), [1000, 2000), ..., [999000, 1000000)这1000个区间，以后对用户积分的更新要相应地更新score_range表的区间值。在分区积分表的辅助下查询积分为s的用户的排名，可以首先确定其所属区间，把高于s的积分区间的count值累加，然后再查询出该用户在本区间内的排名，二者相加即可获得用户的排名。
乍一看，这个方法貌似通过区间聚合减少了查询计算量，实则不然。最大的问题在于如何查询用户在本区间内的排名呢？如果是在算法1中的SQL中加上积分条件：

```
select 1 + count(t2.uid) as rankfrom user_score t1, user_score t2where t1.uid = @uid and t2.score > t1.score and t2.score < @to_score
```

在理想情况下，由于把t2.score的范围限制在了1000以内，如果对score字段建立索引，我们期望本条SQL语句将通过索引大大减少扫描的user_score表的行数。不过真实情况并非如此，t2.score的范围在1000以内并不意味着该区间内的用户数也是1000，因为这里有积分相同的情况存在！二八定律告诉我们，前20%的低分区往往集中了80%的用户，这就是说对于大量低分区用户进行区间内排名查询的性能远不及对少数的高分区用户，所以在一般情况下这种分区方法不会带来实质性的性能提升。
算法特点
优点：注意到了积分区间的存在，并通过预先聚合消除查询的全表扫描。
缺点：积分非均匀分布的特点使得性能提升并不理想。

算法3：树形分区设计

均匀分区查询算法的失败是由于积分分布的非均匀性，那么我们自然就会想，能不能按二八定律，把score_range表设计为非均匀区间呢？比如，把低分区划密集一点，10分一个区间，然后逐渐变成100分，1000分，10000分 ... 当然，这不失为一种方法，不过这种分法有一定的随意性，不容

易把握好，而且整个系统的积分分布会随着使用而逐渐发生变化，最初的较好的分区方法可能会变得不适应未来的情况了。我们希望找到一种分区方法，既可以适应积分非均匀性，又可以适应系统积分分布的变化，这就是树形分区。
我们可以把[0, 1,000,000)作为一级区间；再把一级区间分为两个2级区间[0, 500,000), [500,000, 1,000,000)，然后把二级区间二分为4个3级区间[0, 250,000), [250,000, 500,000), [500,000, 750,000), [750,000, 1,000,000)，依此类推，最终我们会得到1,000,000个21级区间[0,1), [1,2) … [999,999, 1,000,000)。这实际上是把区间组织成了一种平衡二叉树结构，根结点代表一级区间，每个非叶子结点有两个子结点，左子结点代表低分区间，右子结点代表高分区间。树形分区结构需要在更新时保持一种不变量(Invariant)：非叶子结点的count值总是等于其左右子结点的count值之和。

以后，每次用户积分有变化所需要更新的区间数量和积分变化量有关系，积分变化越小更新的区间层次越低。总体上，每次所需要更新的区间数量是用户积分变量的log(n)级别的，也就是说如果用户积分一次变化在百万级，更新区间的数量在二十这个级别。在这种树形分区积分表的辅助下查询积分为s的用户排名，实际上是一个在区间树上由上至下、由粗到细一步步明确s所在位置的过程。比如，对于积分499,000，我们用一个初值为0的排名变量来做累加；首先，它属于1级区间的左子树[0, 500,000)，那么该用户排名应该在右子树[500,000, 1,000,000)的用户数count之后，我们把该count值累加到该用户排名变量，进入下一级区间；其次，它属于3级区间的[250,000, 500,000)，这是2级区间的右子树，所以不用累加count到排名变量，直接进入下一级区间；再次，它属于4级区间的…；直到最后我们把用户积分精确定位在21级区间[499,000, 499,001)，整个累加过程完成，得出排名！
虽然，本算法的更新和查询都涉及到若干个操作，但如果我们为区间的from_score和to_score建立索引，这些操作都是基于键的查询和更新，不会产生表扫描，因此效率更高。另外，本算法并不依赖于关系数据模型和SQL运算，可以轻易地改造为NoSQL等其他存储方式，而基于键的操作也很容易引入缓存机制进一步优化性能。进一步，我们可以估算一下树形区间的数目大约为200,000,000，考虑每个结点的大小，整个结构只占用几十M空间。所以，我们完全可以在内存建立区间树结构，并通过user_score表在O(n)的时间内初始化区间树，然后排名的查询和更新操作都可以在内存进行。一般来讲，同样的算法，从数据库到内存算法的性能提升常常可以达到10^5以上；因此，本算法可以达到非常高的性能。
算法特点
优点：结构稳定，不受积分分布影响；每次查询或更新的复杂度为积分最大值的O(log(n))级别，且与用户规模无关，可以应对海量规模；不依赖于SQL，容易改造为NoSQL或内存数据结构。
缺点：算法相对更复杂。

算法4：积分排名数组

算法3虽然性能较高，达到了积分变化的O(log(n))的复杂度，但是实现上比较复杂。另外，O(log(n))的复杂度只在n特别大的时候才显出它的优势，而实际应用中积分的变化情况往往不会太大，这时和O(n)的算法相比往往没有明显的优势，甚至可能更慢。

考虑到这一情况，仔细观察一下积分变化对排名的具体影响，可以发现某用户的积分从s变为s+n，积分小于s或者大于等于s+n的其他用户排名实际上并不会受到影响，只有积分在[s,s+n)区间内的用户排名会下降1位。我们可以用于一个大小为100,000,000的数组表示积分和排名的对应关系，其中rank[s]表示积分s所对应的排名。初始化时，rank数组可以由user_score表在O(n)的复杂度内计算而来。用户排名的查询和更新基于这个数组来进行。查询积分s所对应的排名直接返回rank[s]即可，复杂度为O(1) ;当用户积分从s变为s+n ,只需要把rank[s]到rank[s+n-1]这n个元素的值增加1即可，复杂度为O(n)。

算法特点

优点：积分排名数组比区间树更简单，易于实现；排名查询复杂度为O(1)；排名更新复杂度O(n)，在积分变化不大的情况下非常高效。

缺点：当n比较大时，需要更新大量元素，效率不如算法3。

总结

上面介绍了用户积分排名的几种算法，算法1简单易于理解和实现，适用于小规模和低并发应用；算法3引入了更复杂的树形分区结构，但是O(log(n))的复杂度性能优越，可以应用于海量规模和高并发；算法4采用简单的排名数组，易于实现，在积分变化不大的情况下性能不亚于算法3。本问题是一个开放性的问题，相信一定还有其他优秀的算法和解决方案，欢迎探讨！

原理：排序要加快速度，是让后一次的排序进行时，尽量利用前一次排序后的结果。

Shell排序即是基于此原则来改良直接插入排序的

按平均时间将排序分为四类：

01. 平方阶(O(n2))排序

一般称为简单排序，例如直接插入、直接选择和冒泡排序；

02. 线性对数阶(O(nlgn))排序

如快速、堆和归并排序；

03. O(n的1+£次方)阶排序

£是介于0和1之间的常数，即0<£<1，如希尔排序；

04. 线性阶(O(n))排序

如桶、箱和基数排序。

各种排序方法比较

简单排序中直接插入最好

快速排序最快

当文件为正序时，直接插入和冒泡均最佳

影响排序效果的因素

因为不同的排序方法适应不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素：

①待排序的记录数目n

②记录的大小(规模)

③关键字的结构及其初始状态

④对稳定性的要求

⑤语言工具的条件

⑥存储结构

⑦时间和辅助空间复杂度等

不同条件下，排序方法的选择

(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好。否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜。

(3)若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。

快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短。

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

若要求排序稳定，则可选用**归并排序**。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。

先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

(将记录分块，块内直接插入，块间归并)

(4)在基于比较的排序方法中，每次比较两个关键字的大小之后，仅仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程。

当文件的n个关键字随机分布时，任何借助于"比较"的排序算法，至少需要O(nlgn)的时间。

箱排序和基数排序只需一步就会引起m种可能的转移，即把一个记录装入m个箱子之一。

因此在一般情况下，箱排序和基数排序可能在O(n)时间内完成对n个记录的排序。

但是，箱排序和基数排序只适用于像字符串和整数这类有明显结构特征的关键字，而当关键字的取值范围属于某个无穷集合(例如实数型关键字)时，无法使用箱排序和基数排序，这时只有借助于"比较"的方法

来排序。

若n很大，记录的关键字位数较少且可以分解时，采用**基数排序**较好。

虽然**桶排序**对关键字的结构无要求，但它也只有在关键字是随机分布时才能使平均时间达到线性阶，否则为平方阶。同时要注意，箱、桶、基数这三种分配排序均假定了关键字若为数字时，则其值均是非负的，否则将其映射到箱(桶)号时，又要增加相应的时间。

(5)有的语言(如Fortran，Cobol或Basic等)没有提供指针及递归，导致实现归并、快速(它们用递归实现较简单)和基数(使用了指针)等排序算法变得复杂。此时可考虑用其它排序。

(6)本章给出的排序算法，输人数据均是存储在一个向量中。当记录的规模较大时，为避免耗费大量的时间去移动记录，可以用链表作为存储结构。

譬如**插入排序**、**归并排序**、**基数排序**都易于在链表上实现，使之减少记录的移动次数。

但有的排序方法，如**快速排序和堆排序**，在链表上却难于实现，在这种情况下，可以提取**关键字建立索引表**，然后对索引表进行排序。

然而更为简单的方法是：引人一个整型向量t作为辅助表，排序前令t[i]=i(0≤i<n)，若排序算法中要求交换R[i]和R[j]，则只需交换t[i]和t[j]即可；

排序结束后，向量t就指示了记录之间的顺序关系：

R[t[0]].key≤R[t[1]].key≤…≤R[t[n-1]].key

若要求最终结果是：

R[0].key≤R[1].key≤…≤R[n-1].key

则可以在排序结束后，再按辅助表所规定的次序重排各记录，完成这种重排的时间是O(n)。

一、问题的提出

1、待排序的记录数量很大，不能一次装入内存，则无法利用前几节讨论的排序方法(否则将引起频繁访问内存)；

2、对外存中数据的读/写是以“数据块”为单位进行的;读/写外存中一个“数据块”的数据所需要的时间为：

TI/O = tseek + tla + n • twm

其中

tseek 为寻查时间(查找该数据块所在磁道)

tla 为等待(延迟)时间

n • twm 为传输数据块中n个记录的时间

二、外部排序的基本过程

由相对独立的两个步骤组成：

1、按可用内存大小，利用内部排序的方法，构造若干(记录的)有序子序列，通常称外存中这些记录有序子序列为“归并段”;

2、通过“归并”，逐步扩大(记录的)有序子序列的长度，直至外存中整个记录序列按关键字有序为止。

例如：假设有一个含10,000个记录的磁盘文件，而当前所用的计算机一次只能对1,000个记录进行内部排序，则首先利用内部排序的方法得到10个初始归并段，然后进行逐趟归并。

(每次读入1000个，采用内部排序的方式排好序，得到10组有序序列。再采用归并排序方式，进行归并。)

假设进行2路归并(即两两归并)，则

第一趟由10个归并段得到5个归并段;

第二趟由 5 个归并段得到3个归并段;

第三趟由 3 个归并段得到2个归并段;

最后一趟归并得到整个记录的有序序列。

分析上述外排过程中访问外存(对外存进行读/写)的次数：

假设“数据块”的大小为200，即每一次访问外存可以读/写200个记录。

则对于10,000个记录，处理一遍需访问外存100次(读和写各50次)。

由此，对上述例子而言：

1. 求得10个初始归并段需访问外存100（读写各50次）次;
2. 每进行一趟归并需访问外存100次;(这里要看内存的大小？)
3. 总计访问外存 100 + 4*100 = 500次。

外排总的时间还应包括内部排序所需时间和逐趟归并时进行内部归并的时间，显然，除去内部排序的因素外，外部排序的时间取决于逐趟归并所需进行的“趟数”。

例如，若对上述例子采用5路归并，则只需进行2趟归并，总的访问外存的次数将

压缩到 100+ 2*100 = 300 次

一般情况下，假设待排记录序列含m个初始归并段，外排时采用k路归并，则归并趟数为 $\lceil \log_k m \rceil$ ，显然，随之k的增大归并的趟数将减少，因此对外排而言，通常采用多路归并。k的大小可选，但需综合考虑各种因素。

基本概念

动态规划过程是：

每次决策依赖于当前状态，又随即引起状态的转移。

一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程

就称为动态规划。

基本思想与策略

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，

前一子问题的解，为后一子问题的求解提供了有用的信息。

在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。

依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，

将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：

适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的

（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

适用的情况

能采用动态规划求解的问题的一般要具有3个性质：

(1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

(2) 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。

也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3)有重叠子问题：

即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。

（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

求解的基本步骤

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。

这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。

动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

初始状态→|决策１|→|决策２|→…→|决策ｎ|→结束状态

图1 动态规划决策过程示意图

(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。

在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3)确定决策并写出状态转移方程：

因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。

所以如果确定了决策，状态转移方程也就可写出。

但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。

实际应用中可以按以下几个简化的步骤进行设计：

（１）分析最优解的性质，并刻画其结构特征。

（２）递归的定义最优解。

（３）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（４）根据计算最优值时得到的信息，构造问题的最优解

算法实现的说明

动态规划的主要难点在于理论上的设计，也就是上面4个步骤的确定，一旦设计完成，实现部分就会非常简单。

使用动态规划求解问题，最重要的就是确定动态规划三要素：

（１）问题的阶段

（２）每个阶段的状态

（３）从前一个阶段转化到后一个阶段之间的递推关系。

递推关系必须是从次小的问题开始到较大的问题之间的转化，

从这个角度来说，动态规划往往可以用递归程序来实现，

不过因为递推可以充分利用前面保存的子问题的解来减少重复计算，

所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。

确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，
其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值
（如最短路径，最长公共子序列，最大价值等），
填表的过程就是根据递推关系，从1行1列开始，以行或者列优先的顺序，依次填写表格，
最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。
f(n,m)=max{f(n-1,m), f(n-1,m-w[n])+P(n,m)}

动态规划算法基本框架

代码

```
 1 for(j=1; j<=m; j=j+1) //  第一个阶段
 2       xn[j] =  初始值;
 3
 4   for(i=n-1; i>=1; i=i-1)//  其他n-1个阶段
 5       for(j=1; j>=f(i); j=j+1)//f(i)与i有关的表达式
 6           xi[j]=j=max（或min）{g(xi-1[j1:j2]), ....., g(xi-1[jk:jk+1])};
 8
 9 t = g(x1[j1:j2]); //  由子问题的最优解求解整个问题的最优解的方案
10
11 print(x1[j1]);
12
13 for(i=2; i<=n-1; i=i+1 ）
15 {
17          t = t-xi-1[ji];
18
19          for(j=1; j>=f(i); j=j+1)
21              if(t=xi[ji])
23                      break;
25 }
```

基本思想：

将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

二分法：

利用分治策略求解时，所需时间取决于分解后子问题的个数、子问题的规模大小等因素，
而二分法，由于其划分的简单和均匀的特点，是经常采用的一种有效的方法，例如二分法检索。

分治法解题的一般步骤：

（1）分解，将要解决的问题划分成若干规模较小的同类问题；
（2）求解，当子问题划分得足够小时，用较简单的方法解决；
（3）合并，按原问题的要求，将子问题的解逐层合并构成原问题的解。

运用分治策略解决的问题一般来说具有以下特点：

1、原问题可以分解为多个子问题
这些子问题与原问题相比，只是问题的规模有所降低，其结构和求解方法与原问题相同或相似。

2、原问题在分解过程中，递归地求解子问题
由于递归都必须有一个终止条件，因此，当分解后的子问题规模足够小时，应能够直接求解。

3、在求解并得到各个子问题的解后
应能够采用某种方式、方法合并或构造出原问题的解。

不难发现，在分治策略中，由于子问题与原问题在结构和解法上的相似性，用分治方法解决的问题，大都采用了递归的形式。

在各种排序方法中，如归并排序、堆排序、快速排序等，都存在有分治的思想。