对于基数排序算法的优化

董震宇1

（1.同济大学 软件学院，上海 201800）

**摘要：**在常规编程环境中，基数排序的关键字索引统计程序内部循环中包含大量操作，导致时间开销很大，并不像线性增长的时间复杂度那样优良。对此问题，本文进行了三个方面的优化：避免复杂数学运算、采用内联函数和选取最适基数，成功地减少了时间的开销，同时证明了基数排序的线性复杂度优势，从而可以更高效地使用基数排序。

**关键词：**基数排序 性能优化 时间复杂度

**Optimization of radix sorting algorithm**

*DONG Zhenyu1*

(1. Tongji University, Shanghai 201800, China)

**Abstract:** In a regular programming environment, cardinality sort keyword index statistics contain a large number of operations in the inner loop of the program, resulting in high time overhead and not as good as the linear increase in time complexity. In this paper, three aspects of optimization: avoid complex mathematical operations, use inline functions and select the optimal cardinality, successfully reduce the cost of time, and prove the linear complexity advantage of cardinality sort, so that it can be used more efficiently.

**Key words:** Radix sort,Performance optimization,Time complexity

引言：在本学期数据结构课程设计时，我对各排序算法的时间开销进行了比较，发现基数排序所需要的时间开销远大于快速排序，这与理论是不相符的。基数排序的时间复杂度为O(d(n+k))(d是数组中的最大位数)，而快速排序的时间复杂度为O(n)。分析发现，在常规编程环境中，关键字索引统计程序中所包含的操作要比快速排序算法内部循环的操作多得多。针对此问题，本文对基数排序算法进行了优化，有明显的优化效果。

**1线性时间复杂度的排序算法——基数排序**

一般来说,排序算法可分为两种:比较排序算法和线性时间排序算法。

比较排序算法，是指通过元素间的大小比较来确定输入序列的元素间相对次序的排序算法。如冒泡排序、直接插入排序、快速排序、堆排序等。评价一个排序算法是否优秀的重要标准是这个排序算法的比较次数。直接插入排序的时间复杂度为O() ,改进后的希尔排序的时间复杂度约为O()，快速排序在平均状态下的时间复杂度为O() ,在特殊状况下(如输入序列本身就是一个有序序列)时间复杂度蜕化为O(n)。而线性时间排序算法的时间复杂度均为O()，所以才被称为线性时间排序算法。

基数排序是一种典型的线性时间排序算法，顾名思义，是基于数列中每个数的每一位而完成的排序，最后得到一个真正有序的序列。文献一已证明,通过比较确定两个元素之间相对位置的比较排序算法的时间复杂度下界为O()，要想改进这个下界，就必须对输入的数据作某些限制。基数排序属于“分配式排序”，它是通过键值的各个部位的值，将要排序的元素分配至某些“桶”中，从而达到排序的作用。

基数排序有两种实现方法，分别是最高位优先法和最低位优先法。最高位优先(Most Significant Digit first)法，简称MSD法：先按k1排序分组，同一组中记录，关键码k1相等，再对各组按k2排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码kd对各子组排序后。再将各组连接起来，便得到一个有序序列。

最低位优先(Least Significant Digit first)法，简称LSD法：先从kd开始排序，再对kd-1进行排序，依次重复，直到对k1排序后便得到一个有序序列。

**2基数排序的时间复杂度与时间运行函数**

在基数排序算法中，对于有n个元素的数组，每趟进行“分配”的while循环需要执行n次，把n个元素分配到radix（基数的个数）个序列中去。进行“收集”的for循环需要执行radix次，从各个队列中把元素收集起来按顺序存放。若每个排序码有d位，需要重复执行d趟“分配”与“收集”，所以总的时间复杂度为O(d(n+radix))。另外，基数排序是一种稳定的排序方法。

基数排序的时间复杂度与三个变量有关，待排序元素的个数n、选取的基数radix和对于基数的最大位数d。而d = ，max为序列中的最大值。对于时间函数T(n,radix,max) = \*(n+radix)，当radix增大时，l减小而n+radix增大，由数学分析知，由于时间函数中有一项，当 max 已知确定时，radix在一定范围内取值的结果是相同的！这意味着，为了让时间尽可能的优，我们只有很少的radix值可以选择，因为当相同时，我们想让(n+radix)尽可能小，就是想让radix尽可能的小。于是，我们将radix可能的取值划分为若干个区间，在每个区间内 的取值相同，而radix可能取的最优值只可能在每个区间的最小值处。例如，当 max = 2 ^ 30 时，radix取1024一定比 1025 更优，因为两者lx的值都是 3，而(n+radix)的值显然前者小于后者。

radix 最小可以取 2，但是为了一项不至于过大，radix不能取过小的值。当基数radix取到n的时候，基数排序就退化成了计数排序。同样，由于基数排序的运行空间函数为 S(n,radix)=(n+radix)，我们也不能让radix取的值过大而超过空间限制。

**3基数排序的优化方法**

**3.1避免复杂的数学运算**

3.1.1 pow函数的替代

基数排序中有一个频繁的操作是获取整数的不同部分，该操作通过get\_part函数获得。上述函数中有一个复杂的pow系统调用函数，这会影响基数排序的速度。该操作的目的是计算基数的幂。在基数固定的前提下，可以提前将基数的幂计算出来，每次调用时通过查表来获得基数的幂函数值。为此，我定义一个常量数组power用来保存基数的次方：

1. **static** **int** power[]={1,radix,radix\*radix,radix\*radix\*radix,
2. radix\*radix\*radix\*radix};

当基数为1000时，上述数组可以应对的最大整数为10^12。

3.1.2将除法变为乘法

在实际运算过程中，除法运算的耗时大于乘法运算，所以一个优化技巧是将除法变成乘法。在此可以定义一个常量数组，用来保存基数次方的倒数，这样就可以将除法转变成乘法：

1. **static** **double** repower[]=
2. {1,1.0/p[1],1.0/p[2],1.0/p[3],1.0/p[4]}

**3.2采用内联函数**

内联函数和普通函数的区别在于：当编译器处理调用内联函数的语句时，不会将该语句编译成函数调用的指令，而是直接将整个函数体的代码插人调用语句处，就像整个函数体在调用处被重写。基数排序时获取整数的不同部分函数多次调用，同时其构造又很简单，可以考虑将其作为内联函数，是排序的性能有些许提升。

**3.3选取最适基数**

3.3.1 取大的基数

假设数组范围为(0, 1000 \* 1000)。以最基础的10为基，那么上述的循环就要进行6次；以100为基，那么上述循环只进行3次即可，而唯一的改变就是在每一次的计数排序过程中，用于计数的数组空间由O(10)变为了O(100)；以1000为基，那么上述循环只需进行2次就可以，这次在每一次的计数排序过程中，用于计数的数组空间由O(10)变为了O(1000);因此，通过增加基的大小可以显著改善效率。通过基数排序的复杂度也可以看出，影响复杂度的一个因素是基数的选择。很多人在使用基数排序时都会默认基数为10，但是这样会显著增大算法复杂度的常量，因而在数组长度较大时，选用较大的基数可能会使性能更好。本文进行的试验中将基数改为100和1000分别进行测试。

3.3.2取2的幂作为基数

在选择基数时我们总会习惯性的以10的幂作为基数，这与我们平时多用10进制运算相符合。但是，计算机是以二进制存储数据的，所以对于乘除取余运算，10进制数的操作较为繁琐。虽然已经对get\_part函数做了很多优化，但是还有一个取余操作尚未优化。我们完全可以采用2的幂作为基数，这样就可以完全避免复杂的乘除法运算。

我们将基数改为1024，此时我们需要修改常量数组p为：

**static** **int** p[]={0,10,20,30};

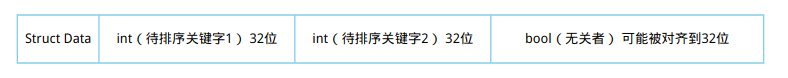
然后get\_part函数变为：

1. **inline** **int** get\_part(**int** n,**int** i)
2. {
3. **return** n>>power[i]&(radix-1);
4. }

可以看出，修改基数为1024之后，除法操作就变为了右移操作，取模操作就变成了与操作。

**3.4一个扩展功能**

同时我们有时候还会有一个对于结构体进行多关键字排序的需求，我们注意到C++内定的数据类型至少为1个字节，一般用的整数为4个字节，然后我们发现对于一个32为整数k，我们对k进行32为的比较可以认为是先进行16位高位的比较然后再进行16位的低位的比较，在高位相等的时候比较低位，这不就是双关键字排序吗？于是结构体的双关键字排序可以这样做：先利用指针类型转换结构体成为32/64位整数，然后再适当调整基数直接排序这些整数，然后就可以得到这些排序后的结构体。

结构体的内存分布如下：

对于这个结构体，使用64位排序，基数。 实现方法：首先开四个数组作为4个关键字的桶，把元素用二进制运算拆开，放入桶内，桶子进行桶排序，利用上一次排序的结果交替利用数组进行下一次排序。

**4 实验结果**

**4.1 快速排序和原始基数排序的性能对比**

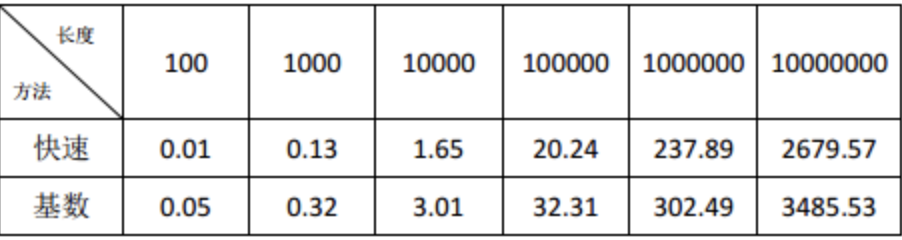
****

表1快速排序与原始基数排序性能对比

**4.2 快速排序和修改基数的基数排序的性能对比**

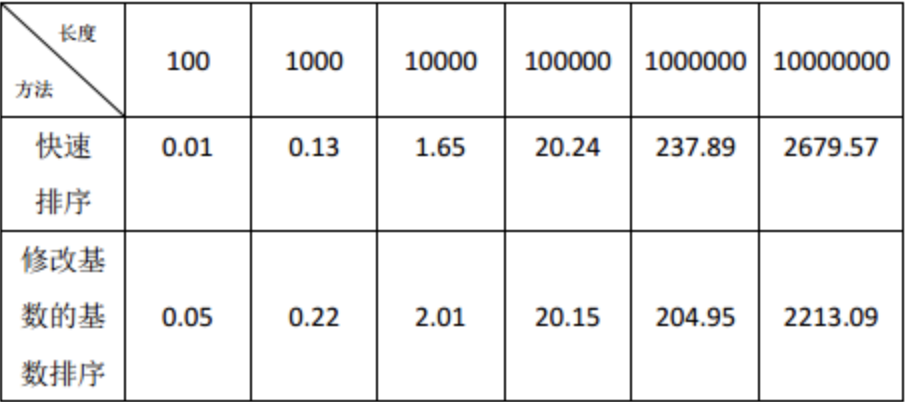


表2快速排序与修改基数的基数排序性能对比



图1 快速排序与基数为10的基数排序对比

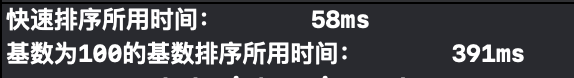


图2 快速排序与基数为100的基数排序对比

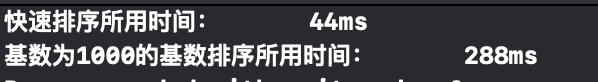


图3 快速排序与基数为1000的基数排序对比

**4.3 快速排序和避免乘方运算的基数排序的性能对比**

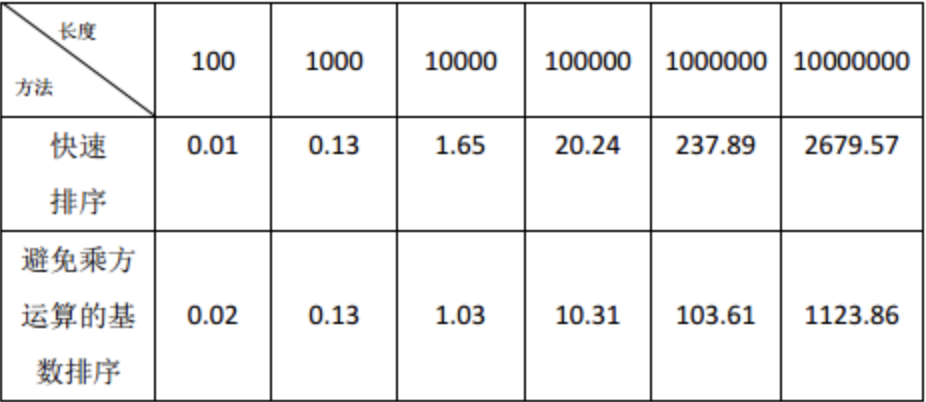


表3快速排序与避免乘方运算的基数排序性能对比

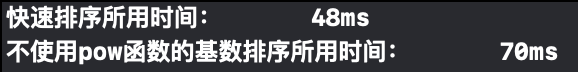


图4 快速排序与避免乘方运算的基数排序对比

**4.4 快速排序和避免乘方运算的基数排序的性能对比**

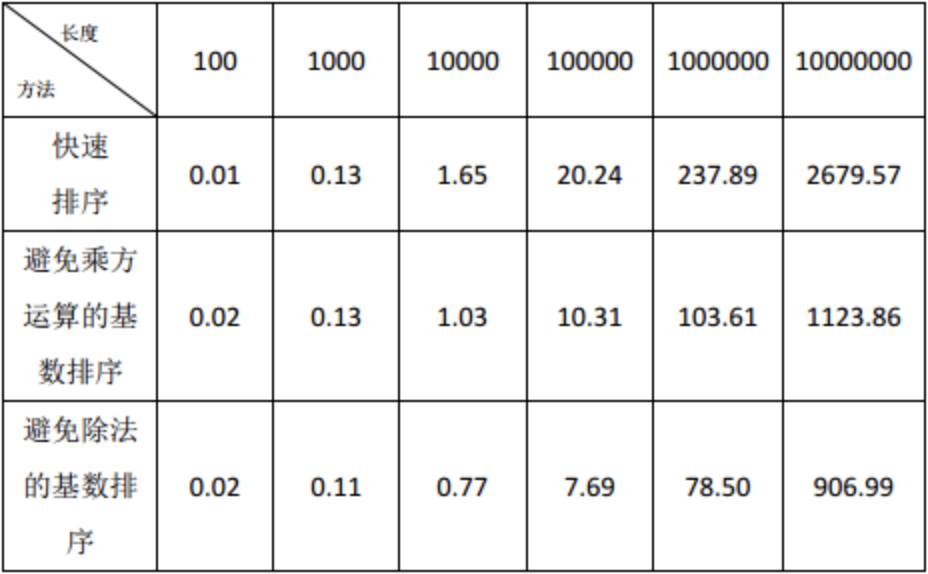


表4 快速排序、避免乘方运算、避免除法运算基数排序性能对比

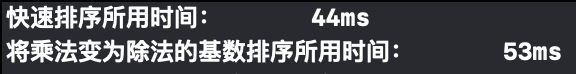


图5 快速排序与避免除法运算的基数排序对比

**4.5 快速排序和采用内联函数的基数排序的性能对比**

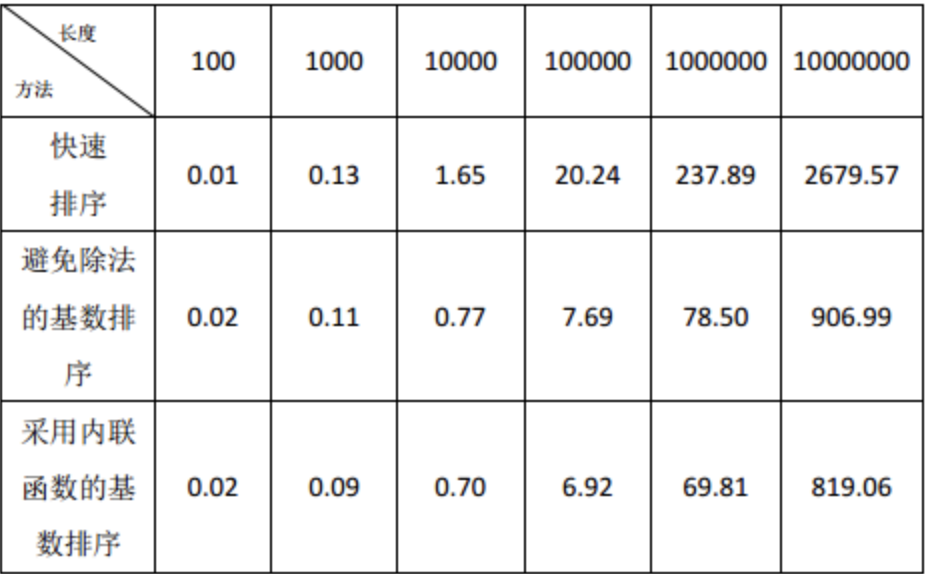


表5 快速排序、避免除法运算基数排序与采用内联函数基数排序性能对比



图6 快速排序与采用内联函数的基数排序对比

**4.6 快速排序和采用内联函数的基数排序的性能对比**

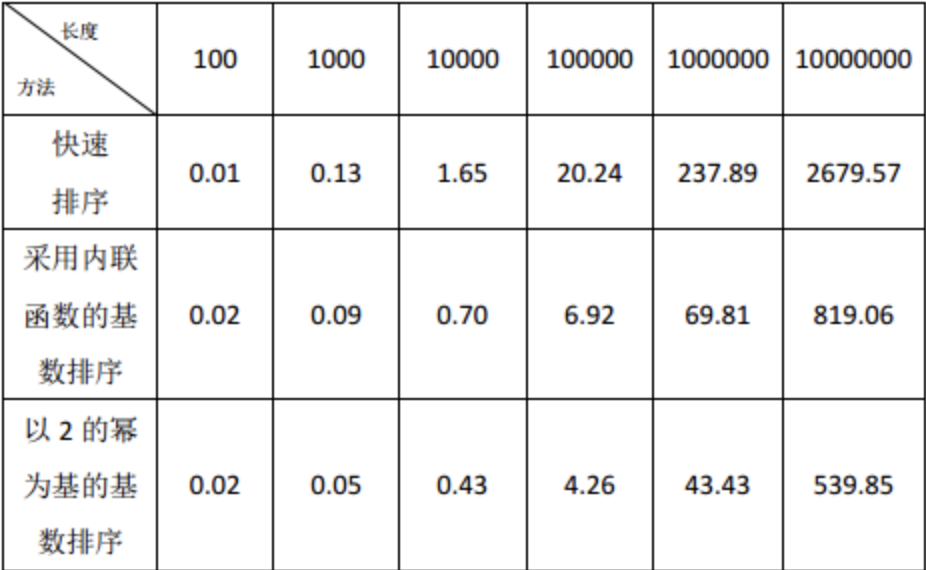
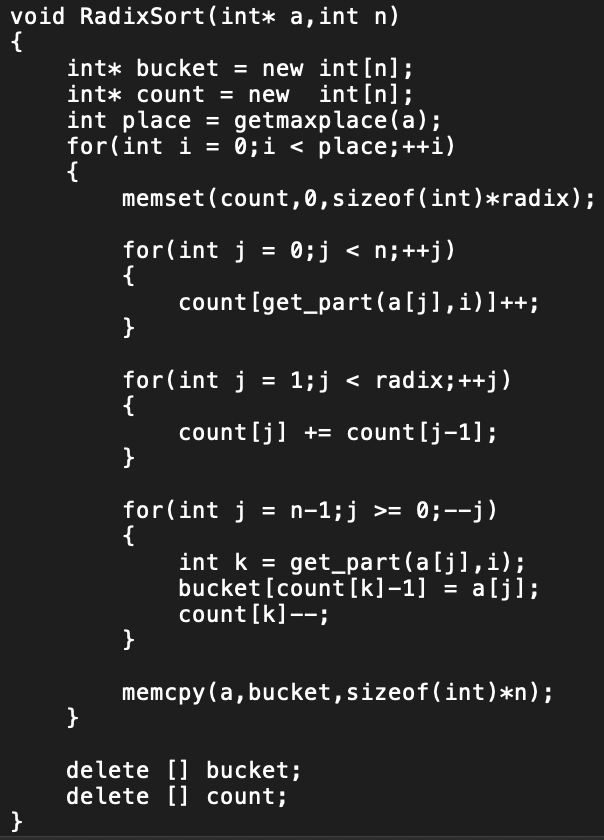


表6 快速排序、采用内联函数、以2的幂为基数的基数排序性能对比

**4.7 实验代码（部分）**



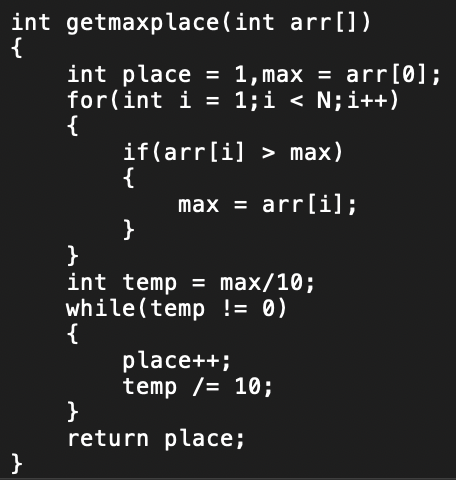
图7 基数排序部分代码1

图8 基数排序部分代码2

**5 结论**

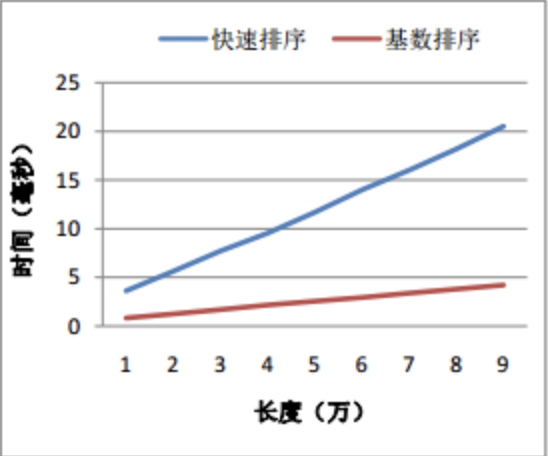
****

图9 快速排序与优化过后基数排序性能对比

通过上面的分析我们可以看出，基数排序确实符合它线性复杂度的优势。如果我们知晓整数数组元素的范围，基数排序确实是一个很好的选择。其实学习基数排序会比较实用，至少理根据新的功能对后缀数组的倍增求法的理解有大大的长进。但是要获得好的性能并不是特别容易，需要很多优化技巧。最好的优化方法就是选择2的幂作为基数。在具体应用时，我们要根据实际的数据范围去合理的选择基数，在确定基数之后再去考虑需要循环的次数。在上面的对比中，数据范围在100万以内，因而循环只有两次，所以快速排序和基数排序的性能差异接近5倍；如果在10亿以内，则需要三次循环，性能差异可能就会降为3倍左右。其实，10亿以内的数几乎快覆盖了int型整数的范围；如果基数选择为2048，则三次循环就完全覆盖了整个int型整数范围。所以，如果要排序的数据范围很大，但是数据量又不足以使用计数排序时，可以考虑采用基数为2048的基数排序。

**参考文献：**

[1] 殷人昆. 数据结构（用面向对象方法与C++语言描述）. 北京:清华大学出版社, 2007.

YIN Renkun. Data Structures (Described in object-oriented and C++ language). Beijing: Tsinghua university press. 2007.

[2] 唐健,周小跃.基数排序及其基于位操作的优化[J].南京工业大学学报(自然科学版).2006,28(6):99-101.

TANG Jian,ZHOU Xiaoyue. Radix sort and its optimization based on bit operation[J]. Journal of Nanjing university of technology (Natural Science edition).2006,28(6):99-101