

|  |  |
| --- | --- |
| **题 目：** | 最高位优先基数排序算法的应用分析与实现 |
| **姓 名：** | 陈润鹏 |
| **学 号：** | 19001013 |
| **所属学院：** | 计算机系 |
| **专业班级：** | 计科192 |
| **指导老师：** | 王梅 |

**完成日期 2022 年 6月**

**本科毕业设计（论文）任务书**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 题目 | 最高位优先基数排序算法的应用分析与实现 | | | | | |
| 姓名 | 陈润鹏 | 专业 | 计算机科学与技术 | | 学号 | 19001013 |
| 1.毕业设计（论文）课题的主要任务（说明：课题背景、意义和培养目标，工程设计类课题应有技术经济分析要求）：  排序是数据处理中最常用的运算之一，排序操作将一组无序的数据序列按某种次序重新排列，从而得到一组有序的数据序列，以方便用户查找，提高检索数据效率。比较关键字和移动记录是实现排序算法的两个基本操作，在经典排序算法中，基数排序是一种非比较型排序算法，利用“分配”和“收集”两种基本操作实现排序的算法。 | | | | | | |
| 2.课题的具体工作内容（原始数据、技术要求、案例分析资料、工作要求等。说明：工科类“技术要求”要有指标参数的具体要求。）：  　　本课题主要详细分析最高位优先基数排序算法的基本思想，算法复杂度，稳定性等特性，并用 C 或者 C++ 高级语言实现该算法，在此基础上，通过与其他排序算法的比较，研究分析该算法在实际应用中能解决的现实问题。   　　基数排序将关键字看作是以某个正整数为基的数，然后依次按关键字的各个字位分别对记录进行排序。本文主要从以下几个方面分析研究最高位优先基数排序算法：   　　1 ）熟悉数据结构，C或C++编程环境。   　　2 ）分析最高位优先基数排序算法的基本思想，算法结构、算法特性。   　　3 ）与其他常用排序算法的综合比较。   　　4 ）算法模块设计。   　　5 ）利用C或C++编程语言实现该算法。   　　6 ）分析研究该算法的具体应用。 | | | | | | |
| 3.课题完成后提交的书面材料要求（论文字数，图纸规格、数量，实物样品，外文翻译字数等）：  (1)毕业论文的字数15000字以上。 (2)毕业设计(论文)用A4纸打印，统一正规装订。 (3)完成课题相关的外文技术资料翻译，外文字符需达到20000字符以上。 (4)主要提交的材料有毕业设计任务书，毕业设计论文(电子版)。 (5)提交源代码。 | | | | | | |
| 4.主要参考文献（指导教师应向学生列出明确的参考文献（含外文文献）一般不少于5篇，不包括词典、手册。学生论文最终引用参考文献不限于任务书所列文献，须达到要求篇数）：   1. 谭浩强 . C语言程序设计（第四版） [M].北京：清华大学出版社，2010. 2. 严蔚敏.吴伟明.数据结构（C语言版）[M].北京：清华大学出版社，2007. 3. 何宇晨.严晶涛等.排序算法的性能比较及基数排序在数据分类中的应用 [J]. 4. 朱建莉.刘宏强.常用排序算法综述[J].胜利油田师范专科学校学报，2002年04期. 5. Colloquium on Computing，Communication， Control， and Management (CCCM2010) Volume 1[C];2010年. 6. 高涛陆丽娜C语言程序设计[M].西安：西安交通大学大学出版社，2007. | | | | | | |
| 要求完成日期： 2023年05月27日  指导教师（签名）： 王梅 | | | | 接受任务日期：2023年01月04日  学生（签名）： 陈润鹏 | | |

摘 要

在社会的高速发展中，计算机已经在人们的生活中息息相关，它的普及使得我们日常的生活更加方便快捷。计算机能够对大量的数据进行处理，最常见的是排序功能，排序算法有冒泡排序、选择排序、插入排序、快速排序，堆排序、基数排序等等。那么针对待排序数据长度过长、待排序的数据距离相差过大、数据类型是字符型等情况进行排序，最高位优先基数排序算法有很大的优势。

本文在阐述大量主流排序算法的基础上，对最高位优先基数排序算法进行了更加深度的分析与探讨。针对生活中常见的一些应用场景进行分析，并且对其中几个十分典型的场景应用算法进行了实现。同时，由于最高位优先基数排序算法和最低位优先基数排序算法都属于基数排序。所以，本文在相关地方对上述的两种算法进行了分析比较，结果显示最高位优先基数排序算法在一些特定的生活应用中更便捷、更高效。

关键词：排序；最高位优先算法；数据结构

ABSTRACT

In the rapid development of society, computers have become closely related to people's lives, and their popularity has made our daily lives more convenient and efficient. The computer can process a large amount of data, the most common is the sorting function. The most common sorting algorithm include bubble sort, selection sorting, insertion sort, quick sorting, heapsort, radix sort, and so on. The highest order first radix sort sorting algorithm has great advantages for sorting when the length of data to be sorted is too long, the distance between data to be sorted is too large, and the data type is character type.

On the basis of expounding a large number of mainstream sorting algorithm, this paper makes a more in-depth analysis and discussion of the highest order first radix sort sorting algorithm. Analyze some common application scenarios in daily life, and implement algorithms in several very typical scenarios. At the same time, because the highest order first radix sort algorithm and the lowest order first radix sort algorithm belong to radix sort. Therefore, this paper analyzes and compares the above two algorithms in relevant places, and the results show that the highest order first radix sort algorithm is more convenient and efficient in some specific life applications.

**KEY WORDS:** Sort High priority algorithm; Data structure;

目 录

[1 绪论 1](#_Toc136728278)

[1.1 课题背景及意义 1](#_Toc136728279)

[1.2 本课题研究主要内容 1](#_Toc136728280)

[1.3 本课题国内外研究现状 2](#_Toc136728281)

[2 排序的基本概念和算法 3](#_Toc136728282)

[2.1 排序的概念 3](#_Toc136728283)

[2.1.1 时间复杂度 3](#_Toc136728284)

[2.1.2 空间复杂度 3](#_Toc136728285)

[2.1.3 稳定性分析 3](#_Toc136728286)

[2.2 排序的两大主流算法思想 4](#_Toc136728287)

[2.2.1 比较类排序 4](#_Toc136728288)

[2.2.2 非比较类排序 5](#_Toc136728289)

[2.2.3 比较类排序和非比较类排序特点之间的比较 5](#_Toc136728290)

[2.3 最高位优先基数排序的思想及其算法 7](#_Toc136728291)

[2.3.1 最高位优先基数排序的具体思想 7](#_Toc136728292)

[2.3.2 最高位优先基数排序的算法分析 8](#_Toc136728293)

[3 最高位优先基数排序算法的应用分析 11](#_Toc136728294)

[3.1 数据库索引 11](#_Toc136728295)

[3.2 垃圾邮件过滤器 12](#_Toc136728296)

[3.3 编码和解码 13](#_Toc136728297)

[3.4 文本搜索 13](#_Toc136728298)

[3.5 电话簿排序 14](#_Toc136728299)

[4 最高位优先基数排序算法的应用实现 15](#_Toc136728300)

[4.1 解决扑克牌排序问题 15](#_Toc136728301)

[4.2 解决电话号码排序问题 19](#_Toc136728302)

[4.3 解决单词表排序问题 24](#_Toc136728303)

[结论与展望 30](#_Toc136728304)

[致 谢 31](#_Toc136728305)

[参考文献 32](#_Toc136728306)

[附录 33](#_Toc136728307)

[正文代码附录1 33](#_Toc136728308)

[正文代码附录2 40](#_Toc136728309)

[正文代码附录3 44](#_Toc136728310)

[**Engineering Radix Sort** 49](#_Toc136728311)

[1. Introduction 49](#_Toc136728312)

[4.4 1.1. Radix Exchange 50](#_Toc136728313)

[4.5 1.2.Quiclcsort to the Fore 52](#_Toc136728314)

[2. List-Based Sort 54](#_Toc136728315)

[3. Two-Array Sort 57](#_Toc136728316)

[4. American Flag Sort 59](#_Toc136728317)

[4.6 4.1. Stack Growth 61](#_Toc136728318)

[4.7 4.2 . Tricks for Tallying 62](#_Toc136728319)

[5. Performance 63](#_Toc136728320)

[6. Discussion 66](#_Toc136728321)

[7. Addendum 71](#_Toc136728322)

[References 71](#_Toc136728323)

[工程基数排序 73](#_Toc136728324)

[**4.8 1. 介绍** 73](#_Toc136728325)

[4.9 1.1. 基数交换 74](#_Toc136728326)

[4.10 1.2. 快速排序到前列 76](#_Toc136728327)

[2. 基于列表的排序 77](#_Toc136728328)

[3. 两个数组排序 80](#_Toc136728329)

[4. 美国国旗分类 81](#_Toc136728330)

[4.11 4.1. 堆栈增长 84](#_Toc136728331)

[4.12 4.2 理货技巧 84](#_Toc136728332)

[5. 展示 85](#_Toc136728333)

[6. 讨论 87](#_Toc136728334)

[7. 附录 91](#_Toc136728335)

[引用 91](#_Toc136728336)

1 绪论

## 课题背景及意义

自第一台计算机诞生以来，计算机产业的飞速发展已远远超出人们对它的预料。快到了几秒就能生产出一台微型计算机，产量猛增，价格低廉，这就使得它的应用范围迅速扩展。如今，计算机已深入到人类社会的各个领域。与此相应的数据也在不断的增加，为了查找方便，通常希望计算机中的表是按关键字有序的。因此，排序就显得尤为重要。

在大量排序算法中，有好多解决不同问题的重要算法，对于不同数据处理的方式存在不同的排序算法。冒泡排序算法经常被用作向学生介绍排序算法和算法复杂性的教学工具；快速排序在相互比较的排序算法中效率算是很优的，常用于金融中对金融数据进行排序，如股价、交易量和市场指数；堆排序用于游戏中对游戏对象的进行排序和管理游戏状态；基数排序，是通过比较各个元素关键字的大小进行排序，既然是数据有起始位置和结束位置，那么就分为最高位优先基数排序和最低位优先基数排序。

数据结构里面的算法可以运用到生活中，研究最高位优先基数算法可以更加了解这个算法。在这个计算机普遍的网络时代 ，最高位基数优先算法的运用可以解决生活中一些特定问题，比如：将大量无序的手机号码进行排序或者是将大量单词按照首字母顺序排序等等。

## 本课题研究主要内容

本文通过对十大主流排序的时间复杂度，空间复杂度，稳定性的三大特点进行了简明的横向分析与对比，分析了比较类排序和非比较类排序，并且对于基数排序的背景下，对最高位优先(MSD)排序和最低位优先(LSD)排序在同一个案例中进行纵向比较，进而突出了最高位优先基数排序处理某些情况下的问题所展示的优越性。

本文对最高位优先算法在一些计算机领域和生活中的应用进行了分析，并且在一些十分具有代表性的问题下，对该算法进行了深入的探究，通过流程图的方式进行了直观的分析与讨论，并且通过代码实现与理论分析相结合的方式进行实现，对于已经实现的应用场景下，总结归纳最高位优先基数算法处理该类型问题相对于其他算法的优越性。

## 本课题国内外研究现状

国内外对MSD基数排序算法进行了研究。一些关键研究主题包括：

1）性能优化：研究人员探索了各种技术来优化MSD基数排序算法的性能，特别是对于大型数据集和具有大范围值的数据集。其中一些技术包括并行实现、缓存感知算法和硬件加速。

2）内存效率：MSD基数排序算法的空间复杂度为O（n+k），其中n是元素的数量，k是每个数字的值范围。研究人员已经研究了降低算法内存需求的方法，例如使用压缩数据结构或利用输入数据的稀疏性。

3）适应特定应用：MSD基数排序算法可以通过修改分区和排序策略来适应特定类型的数据或应用。例如，研究人员开发了用于排序字符串、浮点数和其他专用数据类型的算法变体。

4）理论性质分析：研究人员分析了MSD基数排序算法的理论性质，如其时间和空间复杂性、稳定性，以及与其他排序算法的比较。其中一些研究侧重于开发算法性能的下限，或在某些假设下证明其最优性。

总体而言，MSD基数排序算法是一种强大而通用的排序算法，一直是国内外广泛研究的主题。随着研究人员探索优化其性能、降低内存需求并使其适应新应用的新方法，其研究内容不断发展。

# 排序的基本概念和算法

## 排序的概念

排序是指将杂乱无章的数据元素，通过一定的方法按关键字顺序排列的过程；其目的是将一组“无序”的记录序列调整为“有序”的记录序列。排序分为内部排序和外部排序；若整个排序过程不需要访问外存便能完成，则称为内部排序；反之，则称为外部排序。

### 时间复杂度

时间复杂度可以认为是对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。常见的时间复杂度有：常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(nlog2n),平方阶O(n2)。

### 空间复杂度

空间复杂度是指算法在计算机内执行时所需存储空间的度量，它也是问题规模n的函数。

空间复杂度O(1)：当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)。

空间复杂度O(log2N)：当一个算法的空间复杂度与以2为底的n的对数成正比时，可表示为O(log2n)ax=N，则x=logaN。

空间复杂度O(n)：当一个算法的空间复杂度与n成线性比例关系时，可表示为0(n)。

### 稳定性分析

假设在待排序的记录序列中，存在多个具有相同关键字的记录，若经过排序这些记录相对次序保持不变，即原序列r[i]=r[j]且r[i]在r[j]前面，而排序之后的序列中r[i]仍然在r[j]前面则称这种排序是稳定的，否则称为不稳定的。

在生活中的一些场景下比如给学生考试成绩进行排序时候，如果两个学生的分数一样该怎么排名，如果取上次考试排名靠前的人，排在本次排名的前面，这样就保证了考试排名的公平性，这就是稳定性的优势所在，相反如果不关注这块内容，则排序是混乱的，是缺少公平性。由此可见，排序过程中稳定性是有必要的。

## 排序的两大主流算法思想

十种常见排序算法可以分为两大类：

1）比较类排序：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破O(nlog2n)，因此也称为非线性时间比较类排序。

2）非比较类排序：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。

### 比较类排序

1)冒泡排序（Bubble Sort）

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。（比较相邻两个元素，每两个都进行比较，然后交换。

2)选择排序（Selection Sort）

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

3）插入排序（Insertion Sort）

工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

4）希尔排序（Shell Sort）

把记录下标的一定增量分组，对每组使用插入排序算法，随着增量逐渐减少，每组包含的元素越来越多，当增量减至1时，所有元素被分为一组，算法终止。

5）快速排序（Quick Sort）

从冒泡排序算法演变而来的，实际上是在冒泡排序基础上的递归分治法。快速排序在每一轮挑选一个基准元素，并让其他比它大的元素移动到数列一边，比它小的元素移动到数列的另一边，从而把数列拆解成了两个部分。

6）归并排序（ Merge Sort）

建立在归并操作上的一种有效的排序算法，归并排序对序列的元素进行逐层折半分组，然后从最小分组开始比较排序，合并成一个大的分组，逐层进行，最终所有的元素都是有序的。

7）堆排序（Heap Sort）

利用二叉树的概念来排序的选择排序算法，分为两种：

（1）升序排序：利用二叉树建立大堆，进行排序。

（2）降序排序：利用二叉树建立小堆，进行排序。

### 非比较类排序

1)计数排序（Count Sort）

一种非基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中以达到排序的效果。

2)桶排序（Bucket Sort）

桶排序是计数排序算法的升级版，将数据分到有限数量的桶子里，然后对每个桶再分别进行排序。

3)基数排序（Radix Sort）

将待排序序列的每个元素统一为同样位数长度的元素，位数较短的通过补0达到长度一致，然后从最低位或从最高位开始，依次进行稳定的计数排序，最终形成有序的序列。

基数排序主要是针对整数的排序，由于整数也可以表示字符串或和特定格式的浮点数，因此能用整数表达的其他数据类型也能用基数排序。

基数排序，既可以从高位优先进行排序（简称MSD），也可以从低位优先进行排序（简称LSD）。

### 比较类排序和非比较类排序特点之间的比较

常见的快速排序、归并排序、堆排序、冒泡排序等属于比较排序，在排序的最终结果里，元素之间的次序依赖于他们之间的比较。每个数据都必须和其他数比较，才能确定自己的位置。

1）在冒泡排序之类的排序中，问题规模为n，又因为需要比较n次，所以平均时间复杂度为O(n²)。

2）在归并排序、快速排序之类的排序中，问题规模通过分治法消减为log2N次，所以时间复杂度平均O(nlog2n)。

3）比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。

非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，唯一确定了arr[i]在排序后数组中的位置，非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。因此时间复杂度是O(n)常见的非比较排序有三种：桶排序、计数排序、计数排序。它们的时间复杂度都是O(n)，因为这些排序算法的时间复杂度都是线性的，所以也把这类排序算法称为线性排序。之所以能够做到线性的时间复杂度，主要原因是这几个算法是非基于比较的排序算法，不涉及元素之间的比较操作。这几种排序算法的时间复杂度虽然很低，但是对要排序的数据要求比较苛刻，所以关键是要知道这些排序算法的适用场景，非比较排序时间复杂度底，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

排序算法的时间复杂度，空间复杂度，稳定性的比较汇总如表2-1所示。

表2-1 排序算法时间复杂度，空间复杂度，稳定性的比较



## 最高位优先基数排序的思想及其算法

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。

### 最高位优先基数排序的具体思想

基数排序的思路是：是优先级高的先排序，然后收集；再按照次高优先级排序，然后再收集；依次类推，直到最低优先级的那一位。

基数排序可以说是扩展了的桶排序，比如当待排序列在一个很大的范围内0到999999，那么用桶排序是很浪费空间的。而基数排序把每个排序码拆成由d个排序码，比如任何一个6位数（不满六位前面补0）拆成6个排序码，分别是个位的，十位的，百位的…排序时，分6次完成，每次在同一位上的数字进行排序。

最高位优先基数排序的过程：

1)使用条件

基数排序对要排序的数据是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系，如果 a 数据的高位比 b 数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到 O(n\*k) 了。

2)性能分析

基数排序是一个稳定的排序算法。当输入的元素后，时间复杂度是O(n\*k)，空间复杂度是O(n+k)，其排序速度快于任何比较排序算法。

3)实际应用

假设有10万个手机号码，希望这10万个手机号码从小到大排序，那应该怎么排序。不能用桶排序，因为手机号码有11位，范围太大。这个时候可以使用基数排序。

针对这个问题，假设要比较两个手机号码a、b的大小，如果在前面几位中，a手机号码已经比b手机号码大了，那后面几位号码就不用比较了。这里按照每位来排序的排序算法必须是稳定的，否则这个实现思路就是不正确的。因为如果是非稳定的排序算法，那最好一次排序只会考虑最高位的大小顺序，完全不管其他位的大小关系。

按照每一位来排序，我们可以用基数排序，如果要排序的数据有k位，那就最多需要k次基数排序，例如手机号码排序，k 最大值就是11。

### 最高位优先基数排序的算法分析

最高位优先法通常是一个递归的过程：

先根据最高位关键码K1排序，得到若干对象组，对象组中每个对象都有相同关键码K1。再分别对每组中对象根据关键码K2进行排序，按K2值的不同，再分成若干个更小的子组，每个子组中的对象具有相同的K1和K2值。依此重复，直到对关键码Kd完成排序为止。最后，把所有子组中的对象依次连接起来，就得到一个有序的对象序列。分配排序的基本思想就是进行多次的桶式排序。基数排序过程无须比较关键字，而是通过“分配”和“收集”过程来实现排序。例如：扑克牌中52 张牌，可按花色和面值分成两个字段，其大小关系为：花色：梅花< 方块< 红心< 黑心,面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A,若对扑克牌按花色、面值进行升序排序，即可得到正确序列，即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这就是多关键码排序。为得到排序结果，我们讨论两种排序方法。  
方法一：先对花色排序，将其分为4个组，即梅花组、方块组、红心组、黑心组。再对每个组分别按面值进行排序，最后将4个组连接起来即可。流程如图2-1所示：

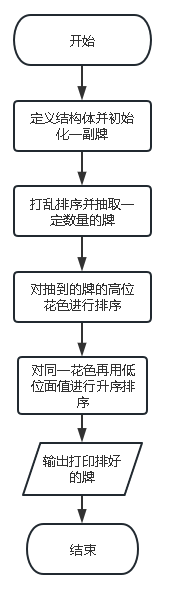


图2-1 按高位排乱序扑克牌的流程

方法二：先按13个面值给出13个编号组(2 号，3 号，...，A 号)，将牌按面值依次放入对应的编号组，分成13 堆。再按花色给出4个编号组(梅花、方块、红心、黑心)，将2号组中牌取出分别放入对应花色组，再将3 号组中牌取出分别放入对应花色组，……，这样，4个花色组中均按面值有序，然后，将4个花色组依次连接起来即可。流程如图2-2所示：

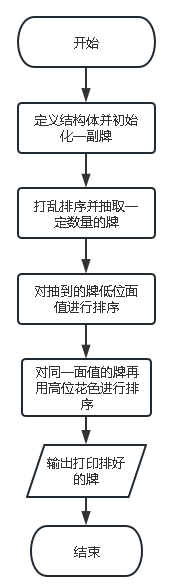


图2-2按低位排乱序扑克牌的流程图

多关键码排序按照从最主位关键码到最次位关键码或从最次位到最主位关键码的顺序逐次排序，分两种方法：

1）最高位优先(Most Significant Digit first)法在上述案例中的思想总结：

（1）先按花色排序分组，将序列分成若干子序列，同一组序列的记录中，关键码花色相等。

（2）再对各组按面值排序分成子组，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有序。

（3）再将各组连接起来，便得到一个有序序列。扑克牌按花色、面值排序中介绍的方法一即是MSD法。

2）最低位优先(Least Significant Digit first)法在上述案例中的思想总结：

（1） 先从扑克牌面值开始排序，再对相同面值不同花色的扑克牌进行花色排序。

（2）最后将各个子序列连接起来，便可得到一个有序的序列， 扑克牌按花色、面值排序中介绍的方法二即是LSD法。

LSD基数排序的最大问题是，它从差异最小的数字开始也就是数字的最低位开始。如果可以从最低位有效数字开始，则第一遍将对排序整个范围帮助不明显，此后的每一遍将对更高一位进行处理。

MSD基数排序的想法是将所有具有相等值的数字划分到各自的存储桶中，然后对所有存储桶执行相同的操作，直到对数组进行排序为止。自然建议使用递归算法，但这也意味着我们现在可以对可变长度的数据进行排序，而不必触摸所有数字即可获得排序数组。这使得MSD基数的分类大大加快并且更加有用。

LSD和MSD都是字符串数组排序算法，它们基于所谓的键索引计数而不是基于比较。因此，与传统的快速排序或归并排序相比，LSD和MSD具有不同的运行时间。

# 最高位优先基数排序算法的应用分析

## 数据库索引

当程序从数据库中快速检索数据时，需要使用索引。使用最高基数排序算法排有效数字可以高效准确地构建索引，实现快速的数据检索。通过高位数字对数据进行排序，该算法可以在索引中快速定位所需的数据。这对于具有许多记录的大型数据库尤其重要，因为传统的搜索方法可能太慢或不准确。因此，使用索引并考虑使用最高位优先基数算法是很重要的。

键索引计数法是MSD的基础。其允许通过一定的键值来对字符串进行排序，一个很好的例子就是分组——每个学生对应一个小组，将他们按照小组分开。在这个问题中，学生姓名就是字符串，而小组的组号就是对应的键值，需要将相同小组的同学分到一起，学生分组后的结果如图3-1所示。

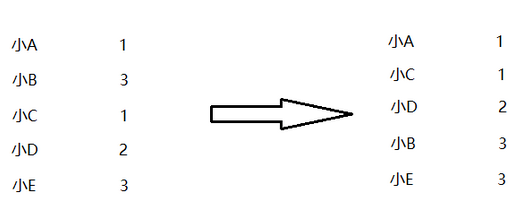


图3-1 将学生分组后的结果

为了完成上述的功能，先统计每个键值的个数并计入一个Count数组中， Count[n + 1]中统计键值为n的字符串的个数，在上图中Count[2] = 2，Count[3] = 1，Count[4] = 2；在统计完成后，需要将频数转换为索引，这是至关重要的一步，在这里用Count[n + 1]来统计键值n而不是Count[n]。使Count[n + 1] += Count[n]，这样Count[0] = 0，Count[1] = 0，Count[2] = 2，Count[3] = 3；可以发现这样变换了以后Count[n]正是下一个键值为n的字符串的索引下标。最后只需要根据索引将字符串存入预先准备的字符串数组及可以完成分组。

MSD的核心思想是分治算法，将大问题分为小问题来解决，其思想与快速排序类似，具体做法是先对最高位的字符进行排序，将排序后的字符串进行分组——最高位相同的在一组；在对同一组的进行MSD排序，不过此时以第二位字符进行排序，直到排完最低位，算法结束。MSD算法的排序过程如图3-2所示。

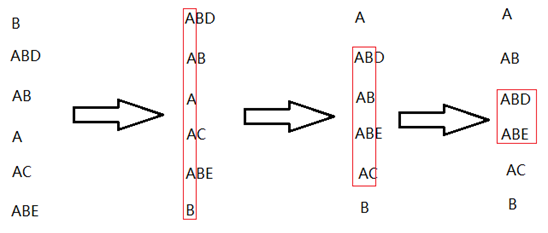


图3-2 MSD算法的排序过程

思想讲起来总是很简单，不过当中的一些细节确实我们需要注意的。 一个显而易见的问题是怎么处理结尾字符的问题，因为MSD运行字符的长度不同，那么总会有字符串先结束，这时候就需要对这些字符串进行处理。

## 垃圾邮件过滤器

垃圾邮件过滤器常常需要分析电子邮件的内容并确定其是否为垃圾邮件。使用最高位优先基数算法可以有效地减少误报率和漏报率。

信头有很多的特征可以用来作为判断是否垃圾邮件。比如群发中的垃圾邮件往往在“To：”域中包含大量的收件人，或者“From：”域中的域名与“Received：”域中的域名不相同。该方法在避免分析邮件正文的前提之下，多数时候能取得比较好的效果。

该算法自动识别表头的“To”并且从该位置从前往后进行检索遇到一个‘@’符号就进行自动给他设置的变量从零开始加加，直到遇到字符‘，’。如果变量大于一定值就自动识别为垃圾邮件进而过滤掉如图3-3所示：

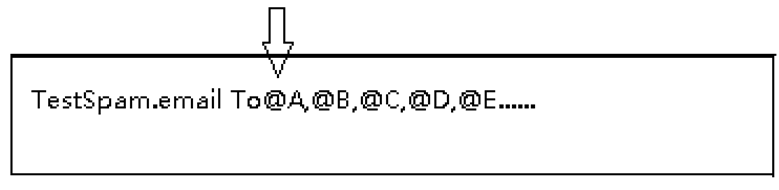


图3-3 模拟电脑通过MSD算法统计邮件的@过程

在垃圾邮件过滤器的上下文中，该算法可以用于有效且准确地分析电子邮件的内容，并确定它是否是垃圾邮件。通过根据数字对电子邮件内容进行排序，该算法可以快速识别垃圾邮件中常见的模式和特征。

总的来说，最高位优先基数排序算法对于各种应用程序是一个有价值的工具，它能够根据数字有效地对数据进行排序，这使它成为分析和处理大型数据集的强大算法。

## 编码和解码

将字符按一定的变法格式转化为字节的方式称为编码，将字节按一定编码格式沾化为字符的方式称为解码。编码和解码通常需要对数据进行排序，以便在传输和存储期间更快地访问数据。最高位优先基数算法可以将数据按照字母或数字的顺序排序，并且这种排序方法易于编码和解码，如图3-4所示：

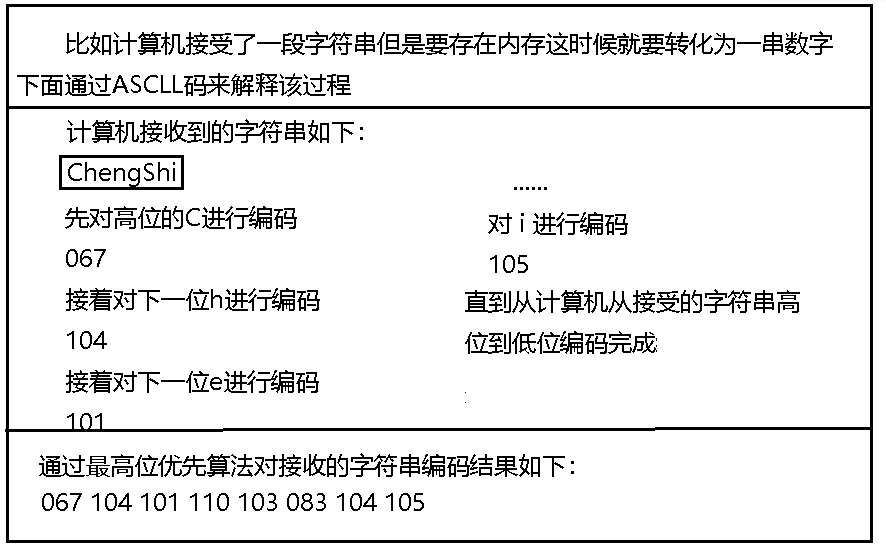


图3-4 模拟计算机内部对字符信息转化编码的过程

在对数据进行编码和解码的情况下，根据数据的数字对数据进行排序对于在传输和存储期间更快、更高效的访问是有用的。通过以这种方式对数据进行排序，该算法可以快速定位所需的数据，并减少编码和解码所需的时间和资源。

## 文本搜索

在文本编辑器中搜索单词或者在搜索引擎中搜索关键字时，需要使用排序算法。最高位优先基数算法可以帮助搜索引擎更快地找到相关的网页。如图3-5所示：

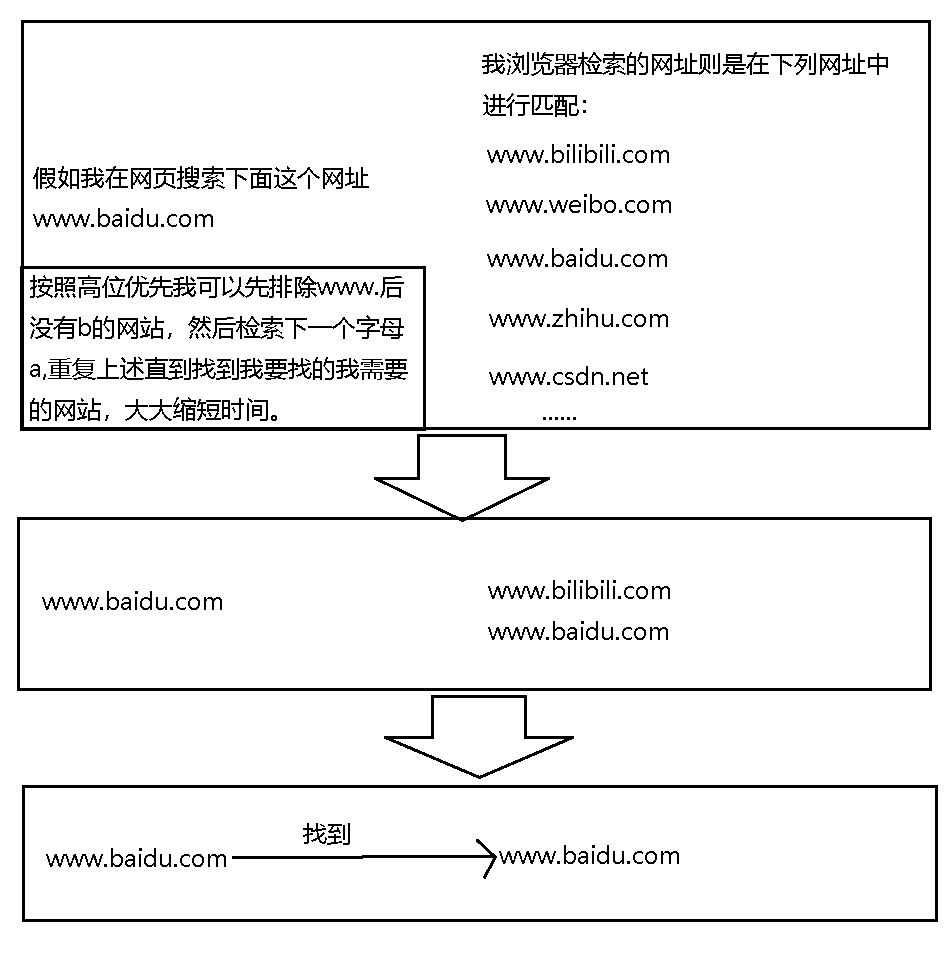


图3-5 模拟高位优先算法找网页的过程

在文本搜索的上下文中，根据文本的数字对文本进行排序对于在搜索过程中更快、更高效地访问非常有用。最高优先基数排序算法可用于根据字母或数字对文本进行排序，从而更容易搜索特定的单词或短语。通过这种方式对文本进行排序，该算法可以快速定位所需的文本，并减少搜索所需的时间和资源。

## 电话簿排序

电话号码经常按照区域和前缀进行排列，以便更快地查找人员的电话号码。最高位优先基数算法可以帮助电话簿更有效地排序。

在电话簿排序的背景下，根据电话号码的数字对其进行排序有助于在搜索过程中更快、更高效地访问。最高优先基数排序算法可用于根据电话号码的区号和前缀对其进行排序，从而更容易定位所需的电话号码。通过以这种方式对电话号码进行排序，该算法可以快速识别电话号码中常见的模式和特征，并减少搜索所需的时间和资源。

在电话簿排序的背景下，该算法可以根据电话号码的区号和前缀快速识别相关电话号码，从而有助于提高电话号码检索的速度和准确性。

总的来说，最高优先基数排序算法是一种有价值的工具，适用于各种应用，包括数据库索引、垃圾邮件过滤、数据编码和解码以及文本搜索等领域。它能够根据数字有效地对数据进行排序，这使它成为分析和处理大型数据集的强大算法。

# 最高位优先基数排序算法的应用实现

## 解决扑克牌排序问题

在网络快速发展的今天，人们的休闲娱乐方式随着科技的发展也变得丰富起来了，与此同时网络游戏在年轻人之间十分流行，人们接触过在线斗地主挖坑等等的网络纸牌游戏，在畅玩游戏的同时扑克牌是怎样精确且有序的分发到若干玩家手中？接下来通过最高位优先基数算法对玩家拿到的牌进行分析，并模拟出游戏内的功能。

扑克牌的顺序由两个内容来决定牌序，花色和面值花色分为，梅花，方块，红心，黑心。面值包括从2 -> 3 ->4 ->5 ->6 ->7 ->8 ->9 ->10 ->J ->Q -> K ->A。那么不算大小王的情况下如何将一副杂乱的扑克牌洗成由花色和面值有规律的序列，如何从牌堆中抽到的十五张顺序混乱的扑克牌排成玩家满意的顺序，这一切都离不开排序最高位优先基数排序。实现了一个扑克牌的排序和洗牌功能，按花色排序流程图如图4-1所示：

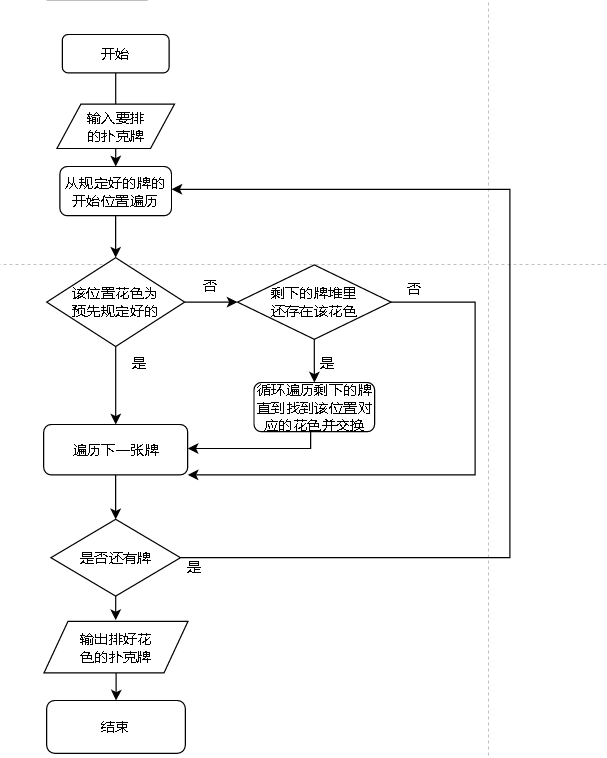


图4-1 对扑克牌花色进行排序流程图

按照花色排完后被分成了四组内容分别对应，梅花，方块，红心，黑心，接着对每种花色内部进行面值排序。按面值排序流程图如图4-2所示：

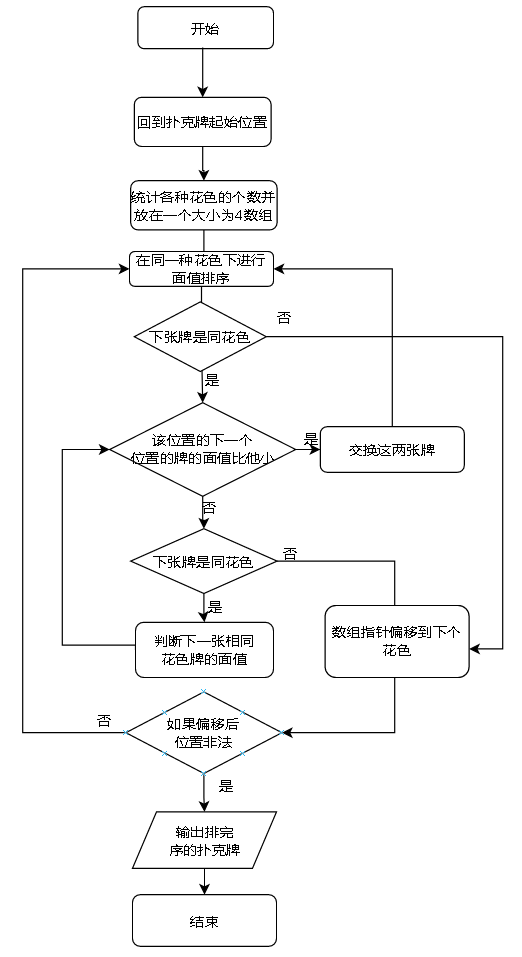


图4-2 按面值排扑克牌流程图

以下是每个函数的作用和结构：  
void swap(pai\_t\*a，pai\_t\*b)：交换两个牌的函数，参数为指向牌的指针。函数内部定义了一个临时变量t，用于交换两个牌的值。  
void init(pai\_tpai[])：初始化牌的函数，参数为指向牌数组的指针。函数内部使用两个循环将所有的牌按花色和面值排列。  
void upset(pai\_tpai[])：洗牌函数，参数为指向牌数组的指针。函数内部使用随机数生成器打乱牌的顺序。  
void print(pai\_tpai[]，int n)：输出牌的函数，参数为指向牌数组的指针和牌的数量。函数内部使用循环将牌按顺序输出。  
void sort(pai\_tpai[]，int n)：排序函数，参数为指向牌数组的指针和牌的数量。函数内部使用两个循环将牌按花色和面值排序。首先按花色排序，然后按面值下面排序。通过C语言来实现上面的思路

Swap()函数对下面要用到的地方进行传地址排序。

void swap(pai\_t\* a, pai\_t\* b) {

pai\_t t = \*a;

\*a = \*b;

\*b = t;

}

既然要排序，又因为扑克牌是由花色和面值来确定的那么创建一个pai的结构体， 其中包含两个整型变量huase和manzhi，分别表示牌的花色和面值。

typedef struct pai {

int huase;

int manzhi;

} pai\_t;

接下来设置两个字符串数组，目的就是针对以上数据进行排序。

char HS[4][8] = { "梅花", "方块", "红心", "黑心" };

char MZ[13][3] = { "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };

然后就是对扑克牌数组进行初始化，从而形成我需要排序的除了大小王的一组生活中常见扑克牌。

void init(pai\_t pai[]) {

int i, j;

for (i = 0; i < 4; i++) {

for (j = 0; j < 13; j++) {

pai[i \* 13 + j].huase = i;

pai[i \* 13 + j].manzhi = j;

}

upset()函数是一个洗牌函数，用于打乱牌的顺序。函数的参数为指向牌数组的指针。函数内部使用srand()和rand()函数生成随机数，然后使用swap函数交换两个牌的位置，从而打乱牌的顺序。具体来说，函数使用一个循环遍历牌数组，每次循环生成一个随机数r，然后将当前牌和第r张牌交换位置。循环次数为牌的数量减一，因为最后一张牌不需要交换

void upset(pai\_t pai[]) {

int r;

srand(time(0));

for (int i = 0; i < 51; i++) {

int ii = i + 1;

r = rand() % (51 - ii + 1) + ii;

swap(&pai[i], &pai[r]);

}

sort()函数是一个排序函数，用于将牌按花色和面值排序。函数的参数为指向牌数组的指针和牌的数量。函数内部使用两个循环将牌按花色和面值排序。首先按花色排序，然后按面值排序。

具体来说，函数使用两个循环遍历牌数组，外层循环变量为i，内层循环变量为j。外层循环从第一张牌开始遍历到倒数第二张牌，内层循环从外层循环变量加一开始遍历到最后一张牌。在内层循环中，如果当前牌的花色比第k张牌的花色小，则将k更新为当前牌的下标。内层循环结束后，如果k不等于i，则交换第i张牌和第k张牌的位置。这样就完成了按花色排序的过程。接着，函数使用一个指针p指向牌数组的起始位置，然后使用一个循环遍历四种花色的牌。循环变量为i，从零开始遍历到三。在循环内部，函数使用两个循环将当前花色的牌按面值排序。外层循环变量为i，内层循环变量为j。外层循环从第一张牌开始遍历到倒数第二张牌，内层循环从外层循环变量加一开始遍历到最后一张牌。在内层循环中，如果当前牌的面值比第k张牌的面值小，则将k更新为当前牌的下标。内层循环结束后，如果k不等于i，则交换第i张牌和第k张牌的位置。这样就完成了按面值排序的过程。最后，函数调用print函数输出排序后的前15张牌。函数内部定义了四个整型变量i、j、k和l，分别用于循环计数和记录当前花色的牌的数量。

void sort(pai\_t pai[], int n) {

int i, j, k, l;

for (i = 0; i < n - 1; i++) {

k = i;

for (j = i + 1; j < n; j++) {

if (pai[k].huase > pai[j].huase) {

k = j;

}

}

if (k != i) {

swap(&pai[i], &pai[k]);

}

}

排序完成后扑克牌的结果如图4-3所示：

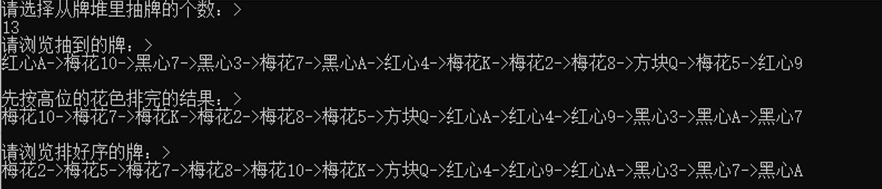


图4-3 按高位优先排完扑克牌的结果

所以上述步骤是一个基数排序的思想对于多关键字分析的一个典型案例，先对最高位(花色)进行排序分桶，然后对最低位(面值)进行整理，最终完成对扑克牌的总体排序。这样排序体现了该算法对于多关键字的排序，解决了其他一些算法对于多关键字类型数据排序无从下手的问题。

## 解决电话号码排序问题

在生活中在移动营业厅办理注册手机号码的时候，都会看到手机业务员拿着几张密密麻麻写满手机号的纸张，提供给消费者来挑选，这时候如果对于所提供的纸张留意的朋友会不会惊奇的发现，在大量的手机号里面，所有号码都是有规律整齐排放的，比如137\*\*\*\*7861，137\*\*\*\*7862，137\*\*\*\*7863以此类推，排列这些手机号码的数字，写一个希尔排序或者是快速排序就行，但是这些排序算法反复的比较手机号数字就显得特别麻烦，但是使用高位优先基数排序算法它可以用于对电话号码、身份证号码等这类问题进行排序。与普通排序算法相比，最高位优先基数排序算法在数字方面排序具有更加高效的优势。提高排序的效率和稳定性。

通过高位优先的排序思想解决大量手机号的排序算法，解决此类问题，不至于挑选手机号时候让顾客眼花缭乱。

所以上述问题实质是一个基数排序的思想的又一个典型案例，先对手机号码最高位进行排序分桶和收集，然后对下一位进行整理，由于这种处理方法是相同的，所有这使程序不得不对上述的重复性动作进行递归操作，最终直到切分的内容是一个单位或者零个单位时候就是我们递归的终止，从而完成我们对电话号码的总体排序。

为了更加直观的理解该算法对于手机号码的排序，接下来便对它进行直观分析。如图4-4所示：

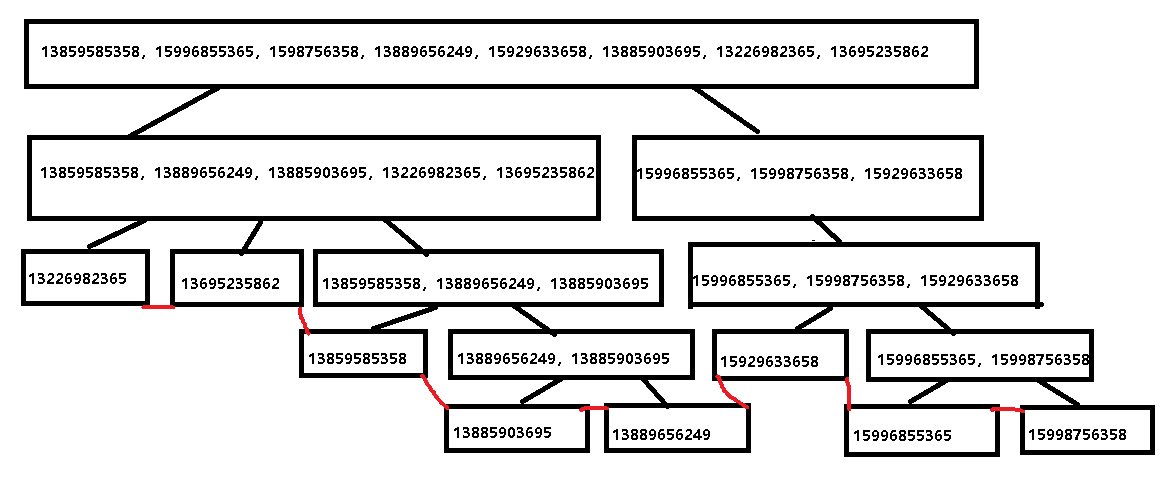


图4-4 按高位优先排电话号码过程直观图

手机号排序的最终结果，如图4-5所示：

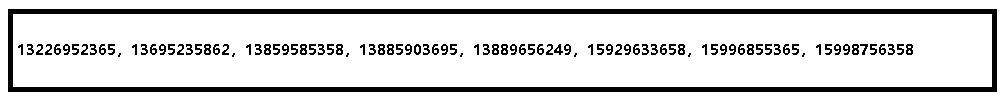


图4-5按高位优先排序最终排好的预期图

按高位排抽到的乱序手机号的流程图如图4-6所示：

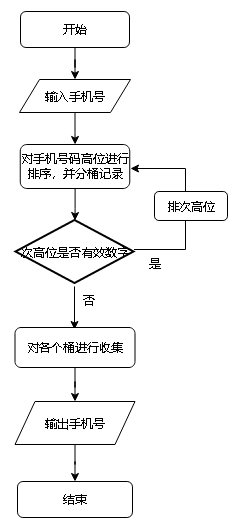


图4-6 按高位排抽到的乱序手机号的流程图

下面我通过C语言来实现上面的思路。

#define MAXN 1000000 + 7

int arr[MAXN];

arr[MAXN]是一个数组，它的作用是存储待排序的整数。在给定的代码中，arr[MAXN]被定义为全局变量，其大小为MAXN，即数组中最多可以存储MAXN个整数。在msdradix\_sort()函数中，会使用arr[MAXN]数组中的元素进行排序。具体实现中，msdradix\_sort()函数会将arr[MAXN]数组中的元素按照当前位数的数字进行分组，并将它们存储在桶数组bucket中。

排序完成后，将排序后的结果复制回arr[MAXN]数组中。因此，arr[MAXN]数组在最高位优先基数排序算法中扮演着重要的角色，它用于存储待排序的整数，并参与到排序的过程中。

int getdigit(int x, int d)

{

int a[] = { 1,1,10,100,1e3,1e4,1e5,1e6,1e7,1e8,1e9 };

return ((x / a[d]) % 10);

}

getdigit()函数是一个用于获取一个整数的某一位数字的函数。它接受两个参数，分别是一个整数x和一个整数d，其中x表示待获取数字的整数，d表示待获取数字的位数。

在函数内部，首先定义了一个大小为11的数组a，用于存储10的幂次方。然后，根据待获取数字的位数d，从数组a中获取对应的幂次方，将其除以x，并对10取模，即可得到待获取数字的某一位数字。最后，将该数字作为函数的返回值返回。

总的来说，getdigit()函数是一个用于获取一个整数的某一位数字的函数，它的实现比较简单，只需要使用数组和取模运算即可。

void msdradix\_sort(int begin, int end, int d){

const int radix = 10;

int count[radix], i, j;

for (i = 0; i < 10; ++i)count[i] = 0;

int\* bucket = (int\*)malloc((end - begin + 1) \* sizeof(int));

for (i = begin; i <= end; ++i) {

count[getdigit(arr[i], d)]++;

}

for (i = 1; i < radix; ++i) {

count[i] += count[i - 1];

}

for (i = end; i >= begin; --i){

j = getdigit(arr[i], d);

bucket[count[j] - 1] = arr[i];

--count[j];

}

for (i = begin, j = 0; i <= end; ++i, ++j){

arr[i] = bucket[j];

}

然后，遍历整个数组arr，将每个数字按照当前位数的权值进行分组，并将每组中数字出现的次数记录在计数数组count中。接着对计数数组count进行前缀和操作，以便确定每个数字在排序后的数组中的位置。然后，从后往前遍历数组arr，将每个数字按照当前位数的权值分配到对应的位置上，并将计数数组count中对应数字的计数减1。最后，将排序后的数组存储在桶数组bucket中，并将其复制回原数组arr中。

接下来，递归调用msdradix\_sort()函数，将当前位数的权值加1，以便对下一位数进行排序。这个过程会一直持续到所有位数都被排序完毕为止。

总的来说，msdradix\_sort()函数是一个使用最高位优先基数算法对数组进行排序的函数，它的实现思路比较巧妙，使用了桶排序和递归的方式进行排序。

for (i = 0; i < radix; i++) {

int p1 = begin + count[i];

int p2 = begin + count[i + 1] - 1;

if (p1 < p2 && d>1) {

msdradix\_sort(p1, p2, d - 1);

}

}

既然有了该算法主题部分的实现那么，主函数遍迎刃而解，为了贴近现实生活中对电话号码的排序这里使用了文件操作，这样在使用该算法的过程中，方便使用者高效的对文件进行修改。

int main() {

int len = 10;

FILE\* input\_file = fopen("input.txt", "r");

if (!input\_file){

puts("input.txt Error");

exit(-1);

}

while (fscanf(input\_file, "%d", &arr[size]) != EOF) {

printf("%d ", arr[size]);

size++;

}

puts("");

fclose(input\_file);

//for (int i = 0; i < size; i++) printf("%d ", arr[i]);

puts("");

msdradix\_sort(0, size - 1, len);

FILE\* output\_file = fopen("output.txt", "w");

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

fprintf(output\_file, "%d\n", arr[i]);

}

puts("");

fclose(output\_file);

}

排序完成后手机号的结果如图4-7所示：

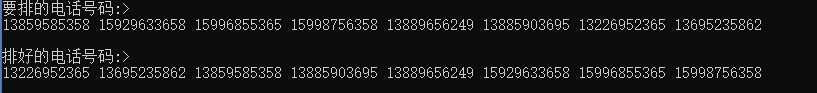


图4-7 手机号排序的结果

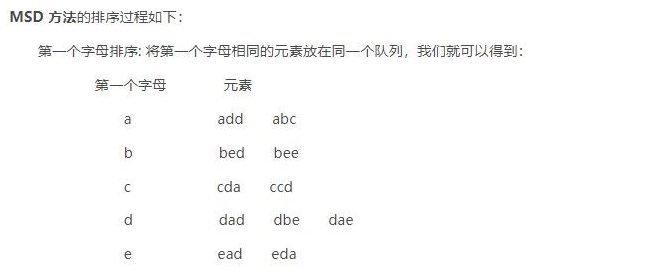
最高位优先基数算法排序算法是一种高效、稳定的排序算法，它可以用于对等长数字进行排序。那么在此情境下相对于最低位优先基数算法，高位优先就体现了优秀的一面，既然最高位已经排出来高低次序了那么就不用在一步一步的去比较低的位数，在实际状况中就节省了时间。

假如计算机要处理一个小区的所有人口的身份证信息，再按最高位优先基数排序来排列身份证号码，这可足足有十八位，那么如果从低位开始整理要整整递归十八次，这还是一个小区的人口普查，那么如果我放到全区，全县甚至是全省，计算机对如此庞大的数据源进行处理，那么高位优先对于身份证号这个关键字的排序的高效性便是不言而喻的。

## 解决单词表排序问题

在现如今的学校中，大学对于英语课程学习的重视程度是非常高的，同学肯定也会接触背单词这件事情，翻到英语书后面的单词表，会看到第一个开始肯定是从A开始的，例如abrupt，abstraction，acquisition…等等这些单词。这些单词确实看起来很整齐，首字母从a开始到的下一个单词也是从a开始的，例如：abrupt和abstraction同样是a开头的字母，为什么abrupt排在前面，而abstraction才排到第二个位置？因为这样排出来可以更加快捷的找到要查找的单词，只要在含有大量单词的词汇书上，从首位开始找对应的字母，然后锁定符合首位的单词列表，再对次一位的字母位置找到想要找的字母，最后循环上述的过程，最终会找到要查找到对应的单词并找到它所对应的汉语翻译。

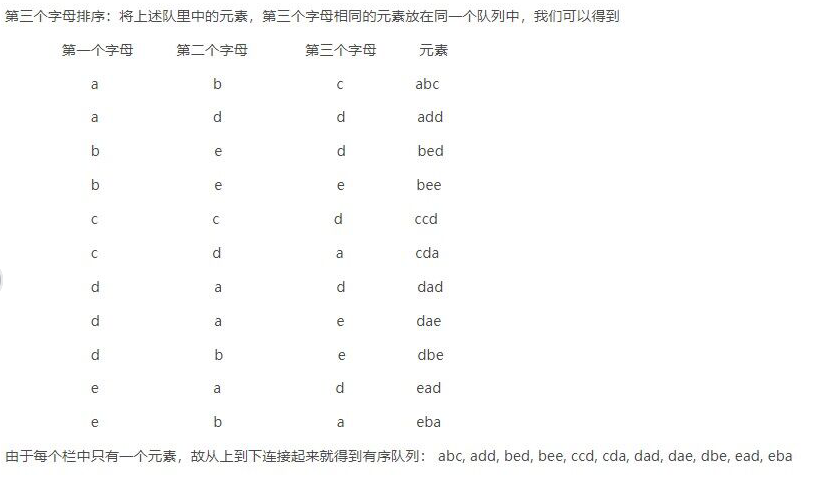
为了解决这字符串排序类问题，可以使用最高位优先基数排序算法。在检索单词的时候也是从最高位开始检索，正因如此，高位优先排序便十分的恰当的去解决大量的且不同长度的字符串的排序问题。虽然最低为优先基数算法同样可以解决大量字符串排序问题，但是高位优先算法从字符串的最高位开始进行数据间的对比，更加的节约排序所花费的时间 ，这一点是低位优先排序所做不到的。下面我们对含有a、b、c、d、e，5个字母的字符串进行排序，由于只排下面这五个字母，所以使用的关键字是5。过程如图4-8（a）（b）（c）所示：



（a）排第一个字母



（b）排第二个字母



（c）排第三个字母

图4-8 最高位优先基数算模拟排列字母序列

按高位排抽到的乱序单词表的方法流程图如图4-9所示：

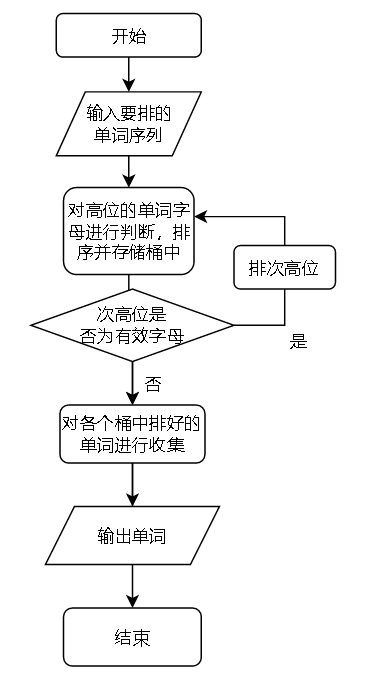


图4-9 按高位排抽到的乱序单词表的方法

下面通过C语言来实现上面的思路

void msdradix\_sort(int begin, int end, int d) {

const int radix = 127;

int count[radix], i, j;

代码行的const int radix=127；将恒定整数变量基数初始化的值为127。此变量用作代码中实现的MSD基数排序算法的基数。基数决定了排序过程中使用的桶数，并根据输入数据中的值范围进行选择。在这种情况下，使用基数127，这是对ASCII字符进行排序的常见选择。MSD基数排序算法是一种稳定高效的排序算法，用于对等长字符串进行排序。

代码行int count[radix]，i，j；声明了三个整数变量：count、i和j。count是一个基数大小的整数数组，基数是代码中初始化为127的常量整数。这个数组用于在排序过程中存储每个bucket中的元素数量。int count[radix]；count是一个整数数组，用于在排序过程中存储每个bucket中的元素数量。

for (i = 0; i < radix; ++i)count[i] = 0;

string\* bucket = new string[end - begin + 1];

for (i = begin; i <= end; ++i) {

count[arr[i][d]]++;

}

for (i = 1; i < radix; ++i) {

count[i] += count[i - 1];

}

for (i = end; i >= begin; --i) {

j = arr[i][d]; bucket[count[j] - 1] = arr[i];

--count[j];

}

接下来初始化count数组，以存储每个bucket中的元素数量。对字符串数组进行排序。具体来说，它初始化计数数组，为桶数组分配内存，并根据当前数字d计算每个桶中的元素数量。这是最高有效数字基数排序算法中的关键步骤，该算法用于根据字符串或整数的数字对其进行排序。for循环从头到尾迭代数组中的元素，对于每个元素，它会根据字符串的第d个字符递增相应bucket的计数。此步骤对于确定分拣过程中将放置在每个桶中的元素数量是必要的。随后的for循环计算每个bucket的右边界，并将元素分类为各自的元素。

for (i = begin, j = 0; i <= end; ++i, ++j) {

arr[i] = bucket[j];

}

delete[] bucket;

for (i = 0; i < radix; i++) {

int p1 = begin + count[i];

int p2 = begin + count[i + 1] - 1;

if (p1 < p2 && d < len) {

msdradix\_sort(p1, p2, d + 1);

}

紧接着对字符串数组进行排序。具体来说，它从每个bucket中收集已排序的元素，并将它们存储回原始数组中。这是最高有效数字基数排序算法的关键一步，该算法用于根据字符串或整数的数字对其进行排序。for循环从头到尾迭代数组中的元素，对于每个元素，它将bucket数组中的相应元素分配给arr数组。这个步骤对于从每个bucket中收集已排序的元素并将它们存储回原始数组是有必要的。

随后的delete[]语句释放为bucket数组分配的内存。最后，如果有更多的数字要排序，函数会在每个bucket上递归调用自己。if语句检查是否有更多的位数要排序，以及当前bucket的左边界是否小于右边f界。如果这两个条件都为真，则函数在当前bucket上递归调用自身，d参数递增1。此步骤对于根据下一个数字对每个bucket中的元素进行排序是必要的。

排序完成的英文单词结果如图4-10所示：

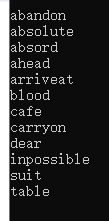
输入： 输出：

图4-10 英文单词排序结果

在主函数中通过对文件的处理方式对单词进行录入，然后以文件方式再输出。

在分析完上述所举的例子之外，可以理解最高位优先技术算法(MSD)是种基数排序算法，特别适用于处理字符串和数字的列表。该算法开始于最高位，从高到低依次进行排序，依据每个数字在各个位上的大小关系。它在现实生活中有许多应用。

# 结论与展望

本次对最高位优先基数排序算法对其应用和其思想有了深刻的了解，也使得自己的知识更加丰富，知道最高位优先基数算法的应用可以运用到我们实际的生活中去。因为之前也只是课堂上在数据结构课中对于排序的各个算法有了一些基本了解但是还是不够深刻的。也正是这次的灵活运用排序的基本算法最高位优先基数排序让自己更加深刻的了解这个算法的魅力和作用。能够灵活的实现和解决一些生活之中出现的一些难题。而且该问题的源码实现运用是结合了平时所学的基础知识来完成的，这不仅巩固了之前所学习的知识并且还对自己会有所提高。使得看待问题的角度和方式方法产生了细微的变化，对自己来说是比较好的有益于自己的。在进行这个系统完成的时候也出现了一些问题，但是在不懈的努力下还是被攻克了，也让自己学会在出现问题的时候要保持冷静然后慢慢一条一条的发现问题逐层递减来看待和处理，而不是焦头烂额的急躁这样反而对问题的解决是没有任何的帮助。在此次的毕业设计中我相对于以前的自己是进步的相信未来也会继续的进步提高，这次过程带给自己的是对以后的路程中要循序渐进，对自己以后的发展还是有很大的帮助的。

这次的毕业设计也参考了其他优秀同学他们的建议取长补短，取其精华弃其糟粕，使得自己的设计更加的尽善尽美。在我看来这次是非常好的经历，在不断的尝试中逐步的成长，自己也学会了很多在老师同学的帮助下我相信自己会越来越好，正是他们的鼓励和支持，使得自己的自信心增加。

# 致 谢

时光荏苒，如白驹过隙，转眼间就到了大学生涯结束的时刻。毕业论文的完成也随之接近了尾声，在次期间完成的艰辛和遇到的困难，在老师，同学，朋友的帮助和鼓励支持下也顺顺利利的完成了。在此聊表心意十分感谢，这个论文过程中提过的建议和方法，使得进度更加快，整体完成度有所提高。

这个毕业设计和论文也是第一次来进行，开始的时候懵懵懂懂总会出一些小小的差错，从选题开题再到设计完成论文一系列的程序。就像一个小白的一样慢慢摸索，相应的在此期间的问题也浮现出来比如什么格式不懂呀没有理解透彻或者设计过程中遇到的难题，出现的bug之类总是干扰着自己进度的前行。但是在老师的帮助和指导提的意见和详细的讲解，这些问题也逐个的被解决被攻克，自己心头里的石头也就悄然无声的落地了。十分感谢我们学校的老师对自己的帮助和提供的建议使得自己在面对这些疑难和问题面前充满着信心。

在整个的过程中，自己有过好多次失败，就是在很小的细节里没有注意使得最后的结果不得人意。但是在我们学校老师的耐心解决一点一点的解决了问题使得自己的进程也逐步顺利的进行，也使得后续我自己在设计和编写的流程中更加的得心应手，所有很感谢老师无私的教诲和帮助，就像蜡烛一样点燃自己照亮别人。

回首整个大学生涯，自己过的还算是充实的，参加过学生会社联认识了精气神和干劲十足的学长学姐使得自己的生活变的更加的充实和美好也学会运用一些软件的技能。对自己的提升还是十分的明显也十分感谢他们的帮助和包容，这使得自己变得更好，向好的方向继续前进。

最后十分感谢在这四年的岁月中，老师们，同学们，朋友们对自己的帮助也希望他们在以后的道路上一路高歌猛进。

# 参考文献

1. 谭浩强. C程序设计(第四版)[J]. 计算机教育, 2011(15):1.
2. 王春森. 系统设计师(高级程序员)教程[M]. 清华大学出版社, 2001.
3. 高一凡. 《数据结构》算法实现及解析——配合严蔚敏,吴伟民编著的《数据结构》(C语言版)[M]. 西安电子科技大学出版社, 2002.
4. 苏光奎, 李春葆. 数据结构导学[M]. 清华大学出版社, 2002.
5. 黄水松, 董红斌编. 数据结构与算法习题解析[M]. 电子工业出版社, 1996.
6. 陈小平. 数据结构导论[M]. 经济科学出版社, 2000.
7. 朱建莉, 刘宏强. 常用排序算法综述[J]. 中国石油大学胜利学院学报, 2002(4):3. :27-29.

1. [Thomas H.Cormen](https://book.douban.com/author/163939)/[Charles E.Leiserson](https://book.douban.com/search/Charles%20E.Leiserson)/[Ronald L.Rivest](https://book.douban.com/search/Ronald%20L.Rivest)/[Clifford Stein](https://book.douban.com/search/Clifford%20Stein),，Introduction to Algorithms[M]. [机械工业出版社](https://book.douban.com/press/2793) 2012-12.
2. 殷人昆. 数据结构:用面向对象方法与C++描述[M]. 清华大学出版社, 1999.
3. 郭龙源, 胡虚怀, 何光明. 数据结构与算法(C语言版)[M]. 清华大学出版社, 2010.

# 附录

# 正文代码附录1

扑克牌排序：

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct pai {

int huase;

int mianzhi;

} pai\_t;

char PokerDecor[4][8] = { "梅花", "方块", "红心", "黑心" };

char FaceValue[13][3] = { "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };

void swap(pai\_t\* a, pai\_t\* b)

{

pai\_t t = \*a;

\*a = \*b;

\*b = t;

}

void init(pai\_t pai[]) //初始化

{

int i, j;

for (i = 0; i < 4; i++)

{

for (j = 0; j < 13; j++)

{

pai[i \* 13 + j].huase = i;

pai[i \* 13 + j].mianzhi = j;

}

}

}

void upset(pai\_t pai[]) //随机数生成

{

int r;

srand((unsigned)time(NULL));

for (int i = 0; i < 51; i++)

{

int ii = i + 1;

r = rand() % (51 - ii + 1) + ii;

swap(&pai[i], &pai[r]);

}

}

void print(pai\_t pai[], int n)

{

for (int i = 0; i < n; i++)

{

if (i == n - 1)

printf("%s%s", PokerDecor[pai[i].huase], FaceValue[pai[i].mianzhi]);

else

printf("%s%s->", PokerDecor[pai[i].huase], FaceValue[pai[i].mianzhi]);

}

puts("");

}

void sort(pai\_t pai[], int n) //排花色

{

int i, j, k, l;

for (i = 0; i < n - 1; i++)

{

k = i;

for (j = i + 1; j < n; j++)

{

if (pai[k].huase > pai[j].huase)

{

k = j;

}

}

if (k != i)

{

swap(&pai[i], &pai[k]);

}

}

int nums[4] = { 0 };

for (i = 0; i < n; i++)

{

nums[pai[i].huase]++;

}

printf("\n");

printf("先按高位的花色排完的结果：>\n");

print(pai, n);

printf("\n");

pai\_t\* p = pai; //初始0偏移量

for (l = 0; l < 4; l++) //排面值

{

for (i = 0; i < nums[l] - 1; i++)

{

k = i;

for (j = i + 1; j < nums[l]; j++)

{

if (p[k].mianzhi > p[j].mianzhi)

{

k = j;

}

}

if (k != i)

{

swap(&p[i], &p[k]);

}

}

p += nums[l]; //每次偏移一个对应数组指针大小的偏移量对下一个花色排序(数组指针偏移大小对应抽到的该花色牌数量)

}

}

int main()

{

int n = 0;

printf("请选择从牌堆里抽牌的个数：>\n");

scanf("%d", &n);

pai\_t pai[52];

init(pai);

upset(pai);

printf("请浏览抽到的牌：>\n");

print(pai, n);

sort(pai, n);

printf("请浏览排好序的牌：>\n");

print(pai, n);

}

# 正文代码附录2

#include<stdio.h>

#include<iostream>

#include<algorithm>

#include<string>

#include<malloc.h>

const int maxn = 1000;

long long arr[maxn];

int size;

int getdigit(long long x, int d) {

long long a[] = { 1,1,10,100,1e3,1e4,1e5,1e6,1e7,1e8,1e9,1e10,1e11 }; //因为待排数据最⼤数据位数

return ((x / a[d]) % 10);

}

void msdradix\_sort(int begin, int end, int d) {

const int radix = 10; //进制数

int count[radix], i, j; //count表⽰每个桶中元素个数

//置零

for (i = 0; i < 10; ++i)count[i] = 0;

//给桶分配空间

long long\* bucket = (long long\*)malloc((end - begin + 1) \* sizeof(long long));

//统计各桶需要装的元素的个数

for (i = begin; i <= end; ++i) {

count[getdigit(arr[i], d)]++;

}

//求出桶的边界索引，count[i]值为第i个桶的右边界索引+1

for (i = 1; i < radix; ++i) {

count[i] += count[i - 1]; //每个桶的边界，便于下步将arr重新放⼊bucket⾥

}

//这⾥要从右向左扫描，保证排序稳定性

for (i = end; i >= begin; --i) {

j = getdigit(arr[i], d); //求出关键码的第d位的数字，例如：576的第3位是5

bucket[count[j] - 1] = arr[i]; //放⼊对应的桶中，count[j]-1是第j个桶的右边界索引

--count[j]; //第j个桶放下⼀个元素的位置(右边界索引+1)

}

//注意：此时count[i]为第i个桶左边界

//从各个桶中收集数据

for (i = begin, j = 0; i <= end; ++i, ++j) {

arr[i] = bucket[j];

}

//释放存储空间

free(bucket);

//对每个桶再次排序

for (i = 0; i < radix; i++) {

int p1 = begin + count[i]; //第I个桶的左边界

int p2 = begin + count[i + 1] - 1; //第i个桶的右边界

if (p1 < p2 && d>1) {

msdradix\_sort(p1, p2, d - 1); //对第i个桶递归调⽤，进⾏基数排序，数位降 1

}

}

}

int main() {

int len = 11;

printf("要排的电话号码:>\n");

// 从文件中读取数据

FILE\* input\_file = fopen("input.txt", "r");

while (fscanf(input\_file, "%lld", &arr[size]) != EOF) {

printf("%lld ", arr[size]);

size++;

}

puts("");

fclose(input\_file);

//for (int i = 0; i < size; i++) printf("%d ", arr[i]);

puts("");

msdradix\_sort(0, size - 1, len); // len表⽰最⾼位数

printf("排好的电话号码:>\n");

// 将排序后的结果写入文件

FILE\* output\_file = fopen("output.txt", "w");

for (int i = 0; i < size; i++) {

printf("%lld ", arr[i]);

fprintf(output\_file, "%lld\n", arr[i]);

}

puts("");

fclose(output\_file);

}

# 正文代码附录3

模拟字典排序

#include<stdio.h>

#include<iostream>

#include <fstream>

#include<algorithm>

#include<string>

#include<malloc.h>

using namespace std;

int len;//最大字符串的长度

const int maxn = 1000;

string arr[maxn];

int length;

void msdradix\_sort(int begin, int end, int d) {

const int radix = 127; //进制数

int count[radix], i, j; //count表⽰每个桶中元素个数

//置零

for (i = 0; i < radix; ++i)count[i] = 0;

//给桶分配空间

string\* bucket = new string[end - begin + 1];

//统计各桶需要装的元素的个数

for (i = begin; i <= end; ++i) {

count[arr[i][d]]++;

}

//求出桶的边界索引，count[i]值为第i个桶的右边界索引+1

for (i = 1; i < radix; ++i) {

count[i] += count[i - 1]; //每个桶的边界，便于下步将arr重新放⼊bucket⾥

}

//这⾥要从右向左扫描，保证排序稳定性

for (i = end; i >= begin; --i) {

j = arr[i][d]; //求出关键码的第d位的数字第i个字符串第d位

bucket[count[j] - 1] = arr[i]; //放⼊对应的桶中，count[j]-1是第j个桶的右边界索引

--count[j]; //第j个桶放下⼀个元素的位置(右边界索引+1)

}

//注意：此时count[i]为第i个桶左边界

//从各个桶中收集数据

for (i = begin, j = 0; i <= end; ++i, ++j) {

arr[i] = bucket[j];

}

//释放存储空间

delete[] bucket;

//对每个桶再次排序

for (i = 0; i < radix; i++) {

int p1 = begin + count[i]; //第i个桶的左边界

int p2 = begin + count[i + 1] - 1; //第i个桶的右边界

if (p1 < p2 && d < len) { //p1<p2 1 个字符停止'a'

msdradix\_sort(p1, p2, d + 1); //对第i个桶递归调⽤，进⾏基数排序，数位降 1

}

}

}

int main() {

// 从文件中读取数据

char buf[50];

FILE\* input\_file = fopen("input.txt", "r");

while (fscanf(input\_file, "%s", buf) != EOF) {

printf("%s\n", buf);

arr[length] = buf;

if (len < arr[length].size())

len = arr[length].size();

length++; //取了多少字符串

}

puts("");

fclose(input\_file);

//for (int i = 0; i < size; i++) printf("%d ", arr[i]);

puts("");

msdradix\_sort(0, length - 1, 0); // len表⽰最⾼位数

// 起始下标 下标 最左边开始

// 将排序后的结果写入文件

FILE\* output\_file = fopen("output.txt", "w");

for (int i = 0; i < length; i++) {

printf("%s\n", arr[i].c\_str());

fprintf(output\_file, "%s\n", arr[i].c\_str());

}

puts("");

fclose(output\_file);

}

# **Engineering Radix Sort**

Peter M. Mcllroy and Keith BosticUniversity of California at Berkeley;And M. Douglas McllroyAT&Bell Laboratories

ABSTRACT：

Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily prograÍrmed algorithms. Three ways to sort strings by bytes left to right-a stable list sort, a stable two-array sort, and an in-place "American flag" sor¿-are illus-trated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

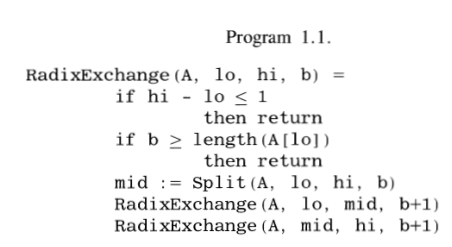
# 1. Introduction

For sorting strings you can't beat radix so the theory says.The idea is simple. Deal the strings into piles by their first letters. Onepile gets all the empty strings. The next gets all the strings that begin with A-; another gets B- strings, and so on. Split these piles recur-sively on second and further letters until the strings end. Vy'hen there are no more piles to split, pick up all the piles in order. The strings are sorted.

In theory radix sort is perfectly efficient. It looks at just enough letters in each string to distinguish it from all the rest. There is no way to inspect fewer letters and still be sure that the strings are properly sorted. But this theory doesn't tell the whole story: it's hard to keep track of the piles.

Our main concern is book keeping, which can make or break radix sorting as a practical method. The paper may be read as a thorough answer to excercises posed in Knuth chapters 5.2 and 5.2.5, where the general plan is laid out.Knuth also describes the other classical sorting methods that we refer to: radix exchange, quicksort, insertion sort,Shell sort, and little-endian radix sort.

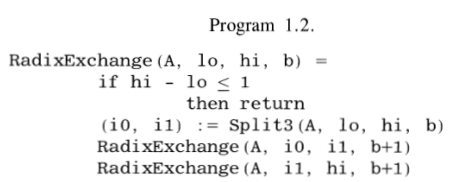
## 1.1. Radix Exchange

For a binary alphabet, radix sorting specializes to the simple method of radix exchange. Split the strings into three piles: the empty strings, those that begin with 0, and those that begin with 1. For classical radix exchange assume further that the strings are all the same length. Then there is no pile for empty strings and splitting can bedone as in quicksort, with a bit test instead of quicksort's comparison to decide which pile a string belongs in.Program 1.1 sorts the part of array A that runs from A[lo]toA[hi - 1]. All the strings in this range have the same å-bit prefix, say

x-. The function split moves strings with prefix -r0- to the beginning of the array, from A [0] through Almid - 1], and strings with prefix x 1- to the end, from A [mid] throtgh Al[hi – 1]To sort an n-element array, call



When strings can have different lengths, a full three-way split is needed, as in Program 1.2. The pile of finished strings, with valuex,say ,begins at A[lo]; the x 0- pile begins at A[i0]; the x 1 -pile begins at Ali1l.



Three-way splitting is the famous problem of the Dutch national flag: separate three mixed colors into bands like the red, white and blue of the flag. For us, the three colors are Ø (no bit), 0 and 1. A recipe for splitting is given in Figure 1.1 and Program 1.3. The index l0 points to the beginning of the 0- pile, i l points just beyond the end of the 0- pile, and i2 points to the beginning of the 1- pile. The notation A[i].å denotes the åth bit, counted from 0, in string A[i]. When split3 finishes, il points to the beginning of the 1- pile as desired.The test for Ø is figurative; it stands for a test for end of string.

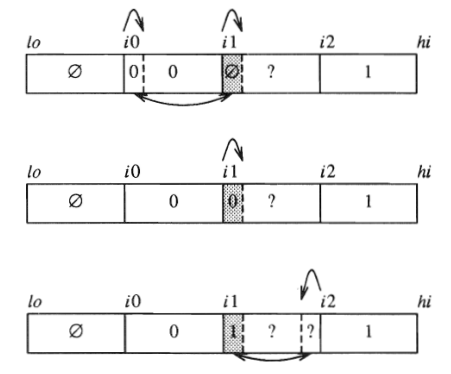
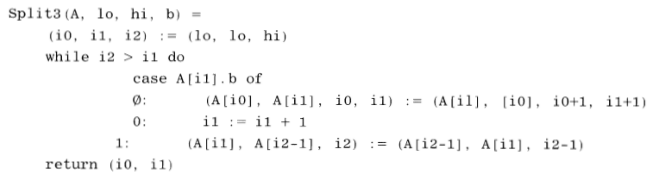


Figure 1.1 How split3 works. The four parts of the array hold strings known to have ended (Ø), strings known to have 0 in the selected position, unknown strings, and strings known to have 1 there. Repeatedly look at the selected position of the first unknown string-the shaded box. Update according to the matching diagram

Program 1.3.



## 1.2.Quiclcsort to the Fore

After enjoying a brief popularity, radix exchange was upstaged by quicksort.Not only was radix exchange usually slower than quicksort, it was not good for programming in Fortran or Algol, which hid the bits that it depends on. Quicksort became known as simple and nearly unbeatable; radix sorting disappeared from textbooks.

Nevertheless, radix exchange cannot be bettered in respect to amount of data looked at. Quicksort doesn't even come close. Quicksort on average performs O (log n) comparisons per string and inspects Ct(log n) bits per comparison. By this measure the expected running time for quicksort is 0(n log2 n), for radix exchange only O(n log n). Worse, quicksort can "go quadratic," and take time A@2 bgn) on unfortunate inputs. This is not just an abstract possibility. Production quicksort routines have gone quadratic on perfectly rea-sonable inputs.

The theoretical advantages of radix exchange are usually swampedby the cost of bit picking. Quicksort is nimbler in several ways.

Quicksort can use machine instructions to compare whole words or bytes instead of bits.

Quicksort splits adaptively. Because it picks splitting values from the data, quicksort can be expected to get roughly 50-50 splits even on skewed data. Such splits are necessary to realize minimal expected times in either quicksort or radix exchange.

Quicksort can sort anything, not just strings. Change thecomparison routine and it is ready to handle different data.Because radix sort intrinsically assumes string data on a finite alphabet, it requires one to make the data fit the routine, not vice versa. For example, to sort dates with quicksort, one might provide code to parse ordinary notation (e.g. February 11,1732)as each key is looked at, while with radix sort one would preconvert all dates to a canonical form (e.g. I732O211).

In other ways, quicksort and radix exchange are quite alike. They both sort in place, using little extra space. Both need a recursion stack, which we expect to grow to size O (log n).In either method, if the strings are long or have different lengths, it is well to address strings through uniform descriptors and to sort by rearranging small descriptors instead of big strings.

The wisdom that blesses quicksort dates from the era of small memories. With bigger machines, the difference between nlog n and n log2 n becomes more significant. And with bigger machines we can afford more space. Thus the wisdom deserves to be reexamined.

# 2. List-Based Sort

For most modern machines, the 8-bit byte is a natural radix, which should overcome the bit-picking slowness of radix exchange. A byte radix makes for 256- or 257 -way splitting, depending on how the ends of strings are determined. This raises the problem of managing space for so many piles of unknown size at each level of recursion. An array of linked lists is an obvious data structure. Dealing to the piles is easy;just index into the array. Picking up the sorted piles and getting them hooked together into a single list is a bit tricky, but takes little code. Program 2.1 does the job. It is written in C rather than pseudocode, because the troubles with radix sort are in implementation, not in con-ception. The input variables are

a linked list of null-terminated strings.

b the offset of the byte to split on; the strings agree in all earlier bytes.

Sequel a sorted linked list of strings that compare greater than the strings in list a.

Three in-line functions are coded as macros:

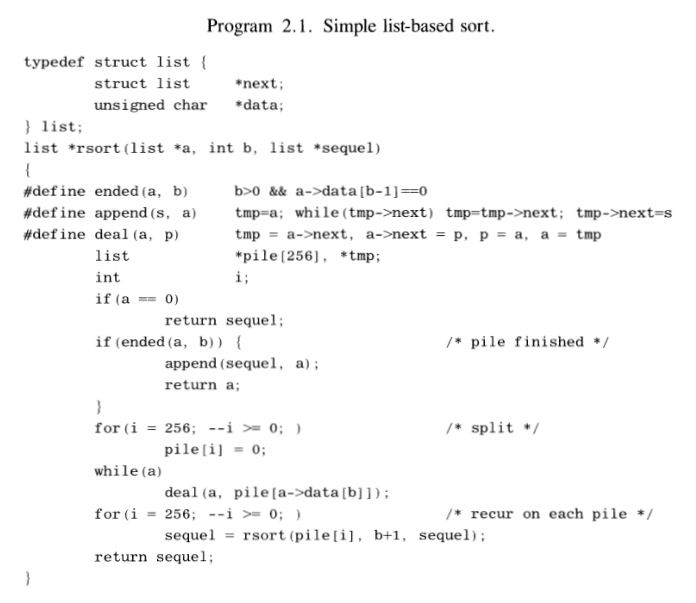
ended (a, b) tells whether byte position b is just beyond the null byte at the end of string a.

append (s, a) appends list s to the last element of non-empty list a.

deal (a, p) removes the first string from list a and deals it to pile p.

Program 2.1 has four parts. First, if the list is empty, then the result of sorting it together with sequel is sequel. Next, at "pile finished," if the last byte seen in the strings in list d was null (0), they cannot be sorted further. Put them in front of the sequel and return the combined list. At "split," all strings have a bth byte. Clear all the piles and then deal the strings out according to byte b of each string. Finally, at "recur on each pile," sort the piles from last to first. At each stage append to the sorted current pile the sorted list accumulated from all following piles.

Program 2.1 works-slowly. Empty piles are the root of the trouble. Except possibly at the first level or two of recursion, most piles will be empty. The cost of clearing and recursively "sorting" as many.



as 255 empty piles for each byte of data is overwhelming. Some easy improvements will speed things up.

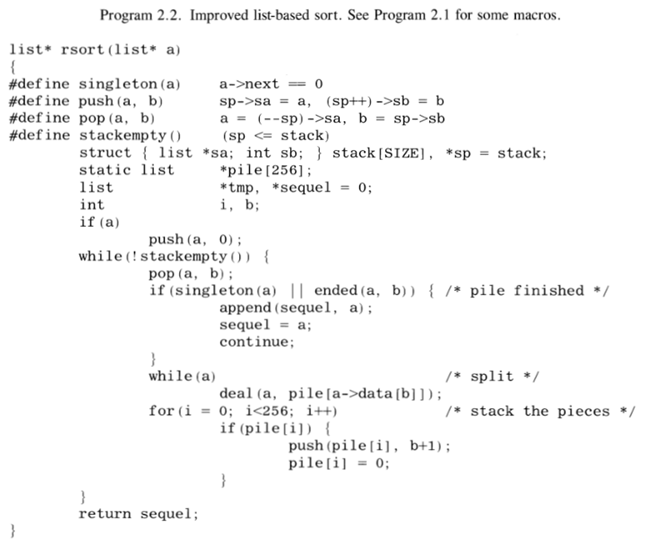
12.1Always deal into the same array and clear only the occupiedpiles between deals, meanwhile stackirig the occupied piles out of the way.

12.2 Manage the stack directly. Since the number of occupied piles is unpredictable, and probably small except at the first level or two of recursion, much space can be saved. The piles may be stacked in first-to-last order so they will pop off in last-to-first order just as in Program 2.1.

12.3 Don't try to split singleton piles.

12.4 Optimize judiciously: eliminate redundant computation;replace subscripts by pointers.

12.5 Avoid looking at empty piles.



Program 2.2 implements improvements l2.I-I2.3. Here the statecarrying parameters b and sequel become hidden, as they should be.On typical aþabetic data Program2.2 runs about 15 times as fastas Program 2.1-not a bad return from such simple optimizations, but not yet good enough. Most of the time for Program 2.2 is still wasted scanning empty piles. Thus we turn to improvementI2.5, avoiding looking at empty piles, which can be done in many ways. For textual keys, such as names or decimal numbers, the piles are likely to be bunched. Single-case letters span only 26 of the 256 piles, digits only 10. To exploit bunching, we use a simple pile-span heuristic: keep track of the range of occupied piles. The finished 0- pile is an expected outlier and is kept track of separately.

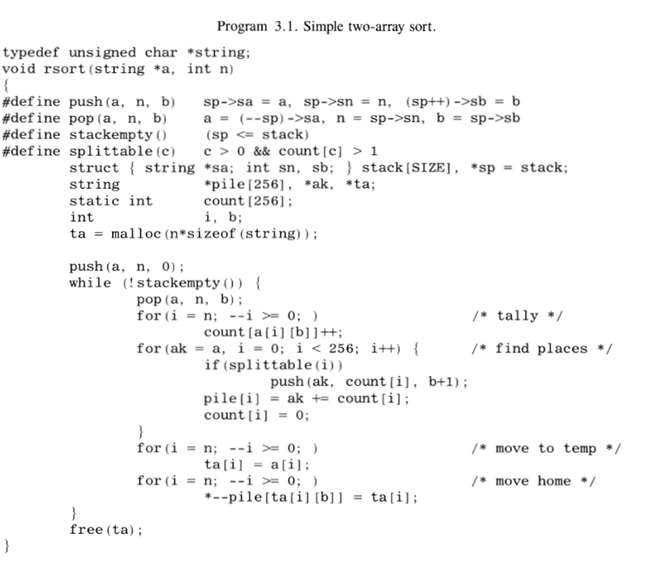
Thken together, improvements 12.l-I2.5 speed up Program 2.1 by a factor of 100 on typical inputs. The result, Program A in the appendix, is a creditable routine. It usually sorts arrays of 10,000 to100,000 keys twice as fast as do competitive quicksorts.

None of our programs so far sorts stably. Because piles are built by pushing records on the front of lists, the order of equal-keyed records is reversed at each deal. To stabilize the sort we can reverse each backwards pile as we append to it. Alternatively we can maintain the piles in forward order by keeping track of head and tail of each pile. Program A does the latter. Sorting times differ negligibly among forward/reverse and stable/unstable versions.

# 3. Two-Array Sort

Suppose the strings come in an arÍay as for radix exchange. In basic radix exchange, the two piles live in known positions against the bottom and top of the array. For larger radixes, the positions of the piles can be calculated in an extra pass that tallies how many strings belong in each pile. Knowing the sizes of the piles, we don't need linked lists.

Program 3.1 gets the strings home by moving them as a block to the auxiliary afiay ta, and then moving each element back to its proper place. The upper ends of the places are precomputed in array pile as shown in Figure 4. 1. (This "backward" choice is for harmony with the programs in section 4.) Elements are moved stably; equal elements retain the order they had in the input. As in Program 2.2, the



stack is managed explicitly; the stack has a third field to hold the length of each subarray.

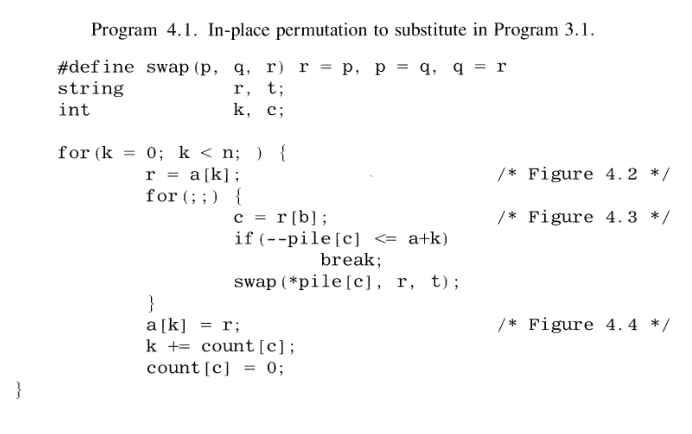
Program 3.1 is amenable to most of the improvements listed in section 2; they appear in Program B. In addition, the piles are independent and need not be handled in order. Nor is it necessary to record the places of empty piles. These observations are embodied in the "find places" step of Program B.

As we observed in the introduction, radix sorting is most advantageous for large arrays. When the piles get small, we may profitably divert to a simple low-overhead comparison-based sorting method such as insertion sort or Shell sort. Diversion thresholds between 10 and 50 work well; the exact value is not critical. Program B in the appendix is such a hybrid two-array sort. It is competitive with list-based sort; which of the two methods wins depends on what computer, compiler, and test data one measures. For library purposes, an array interface is more natural than a list interface. But two-array sort dilutes that advantage by using O(n) working space and dynamic storage allocation. Our next variant overcomes this drawback.

# 4. American Flag Sort

Instead of copying data to an auxiliary array and back, we can permute the data in place. The central problem, a nice exercise in practical algorithmics, is to rearrange into ascending order an array of ninteger values in the range 0 to m - 1. Here m is a value of moderatesize, fixed in our case at 256, andn is arbitrary. Special cases are the partition step of quicksort (m = 2) and the Dutch national flag problem (m = 3). By analogy with the latter, we call the general problem the American flag problem. (The many stripes are understood to belabeled distinctly, as if with the names of the several states in theoriginal American union.)

American flag sort differs from the two-array sort mainly in its final phase. The effect of the "move to temp" and "move home" phases of Program 3.1 is attained by the "permute home" phase shown in Program 4.1 and Figures 4.1-4.4. This phase fills piles from the top, making room by cyclically displacing elements from pile to pile.



Let alkf be the first element of the first pile not yet known to be completely in place. Displace this element out of line to r (Figure 4.2).Let c be the number of the pile the displaced element belongs to. Find in the c- pile the next unfilled location, just below pílelc] (Figure4.3). This location is the home of the displaced element. Swap the displaced element home after updating pilelcl to account for it.

Repeat the operation of Figure 4.3 on the newly displaced element, following a cycle of the permutation until finally the home of the displaced element is where the cycle started, at alk]. Move the displaced element to ølk].Its pile, the current cpile, is now filled (Figurea.a). Skip to the beginning of the next pile by incrementing ft. (Values in the count array must be retained from the "find places" phase.) Clear the count of the just-completed pile, and begin another permutation cycle. It is easy to check that the code works right when a + k =pilel[c] initially, that is, when the pile is already in place.

When all piles but one are in place, the last pile must necessarily be in place, too. Progrum 4.2, otherwise a condensed Program 4.1,exploits this fact. Program 4.2 and Program B form the the basis of Program C in the appendix.

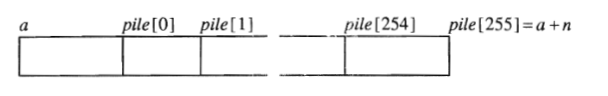


Figure 4.1. Array a before permuting home.

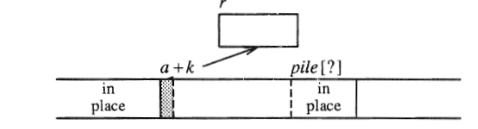


Figure 4.2. After first displacement. Arrow shows completed action.

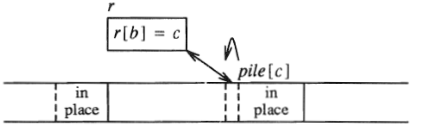


Figure 4.3. During displacement cycle. The bth byte of the string

pointed to by r is c. Arrows show actions to do, except no swap

happens in last iteration.

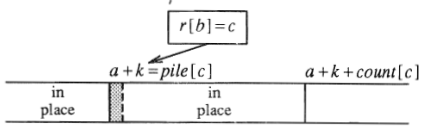
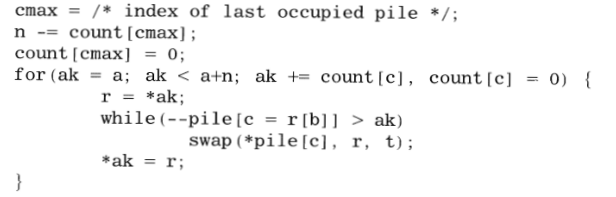


Figure 4.4. Last move.

Program 4.2. lmproved in-place permutation.



## 4.1. Stack Growth

In the programs so far, the stack potentially grows linearly with running time. We can bound the growth logarithmically by arranging to split the largest pile at each level last-a trick well known from quicksort. This biggest-pile-last strategy is easy to install in array-based sorts, but inconvenient for list-based, where stack order matters and pile sizes are not automatically available.

Even with a logarithmic bound, the stack can be sizable. In the worst case, a split produces no 0- piles, 253 little (size 2) piles, and two big piles of equal size for the rest of the data. One of the big piles is immediately popped from the stack and split similarþ. For n : 1,000,000, the worst-case stack has 2800 entries (33,000 bytes on a 32-bit machine). By contrast, a stack of about 254logrru n entries (only 630 when n : I,000,000) suffices for uniform data. Still smaller stacks work for realistic data. Instead of a worst-case stack, we may allocate a short stack, say enough for two levels of full 256way splitting, and call the routine recursively in the rare case of overflow.

For completeness, both stacking tactics are shown in program C,though they will almost surely never be needed in practice. Stack control adds one-third to the executable code, but only about one percent to the running time.

## 4.2 . Tricks for Tallying

The pile-span heuristic for coping with empty piles presupposes a not unfavorably distributed aþabet. Other ways to avoid looking at empty piles may be found in the literature. One is to keep a list of occupied piles. Each time a string goes into an empty pile, record that pile in the list. After the deal, sort the list of occupied piles by pile number. If the list is too long, ignore it and scan all the piles.A version of Program C with this strategy instead of pile-span ran slightly faster on adverse data, but slower on reasonable data.

Alternatively, an occupancy tree may be superimposed on the array of piles. Then the amount of work to locate occupied piles will di minish with diminishing occupancy. The best of several tree-tallying schemes that we have tried is quite insensitive to the distribution of strings and beats pile-span decisively on adverse data, but normally runs about one-third to one-half slower than pile-span.

Noting that only the identity and not the order of the piles matters in splitting, Paige and Tarjan propose to scan piles in one combined pass after all splits are done. Their method favors large radixes; it runs faster with radix 64K than with radix 256, Unfortunately, overhead-from 4n to 8n extra words of memory-swamps the theoretical advantage.

Little-endian (last-letter first) sorting mitigates the problem of scanning empty piles. In little-endian sorts the number of splits is equal to the number of letters in the longest key, whereas in bigendian sorts like ours the number of splits typically exceeds the number of keys. Aho, Hopcroft, and Ullman show how to eliminate pile scanning at each deal of a little-endian sort by using a O ("total size")presort of all letters from all keys to predict what piles will occur. Alittle-endian radix sort, however, must visit all letters of all keys instead ofjust the letters of distinguishing prefixes.

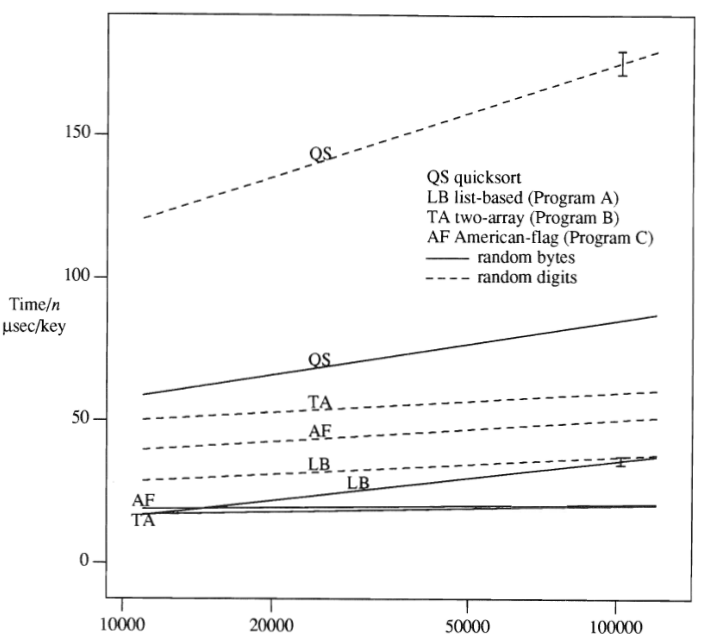
In practice, exotic tricks for tallying are rendered moot by diverting to an alternate algorithm for small r. For it is only when n is small that the time to scan piles is noticeable in comparison to the time to deal and permute. Nevertheless, we still like the extremely cheap pile-span heuristic as a supplemental strategy, for it can improve running times as much as IOVo beyond diversion alone.

# 5. Performance

The merits of the several programs must be judged on preponderant, not decisive, evidence. In theory, all have the same worst-case asymptotic running time, O(S), where .S is the size of the data measured in bytes. None is clearly dominant in practical terms, either. The relative behaviors vary with data, hardware, and compiler.

In assessing performance, we shall consider only large data sets,where radix sorting is most attractive. Just how attractive is indicated by comparison with quicksort. The tested quicksort program, which compares strings in line, chooses a random splitting element, and diverts to a simple sort for small arrays, was specialized from a model by Bentley and Mcllroy. The routine is not best possible, but probablywithin l/3 of the ultimate speed for C code. (We recoiled from adapting their fastest model, which would require 23 in-line string comparisons.)

Figure 5.1 shows the variation with size for 15 tests of each offour routines on one computer for two kinds of random key: (1) strings of 8 random decimal digits and (2) strings of random bytes, exponentially distributed in length with mean 9. The range of this experiment is too narrow to reveal quicksort's nlog2 n depafture from linearity,or to fully smooth quantizing effects. (Across this range the expected



Number of keys, n

Figure 5.1. Leasrsquares fits to sorting time per key versus log n for

a DEC VAX 8550. Representative t I cr eÍÍor bars are shown; other

curves fit comparably.

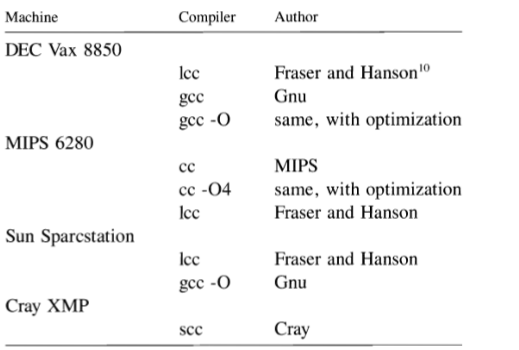


Table 5.1. Machines and compilers tested.

length of comparisons varies by only one digit for the decimal data and half a byte for the byte data.) Nevertheless the figure clearly shows that the radix sorts run markedly faster than quicksort. This observation is robust. No comparable generalization can be drawn about the relative performance of the three radix sorts. As the figure shows, the rank order depends on the kind of data. Other experiments show similar variation with hardware.

The sensitivity of the pile-span heuristic to key distribution showsup in Figure 5.1 as a large variation of slope for list-based sort. In the other two routines, diversion almost completely damps the variation. The sensitivity is even greater on sparse random data. In the extreme case of random keys strings containing just two byte values, the listbased Program A took about 5 times as long to sort keys with distant byte values as with adjacent values. The hybrid Programs B and C varied by a factor of 1.2 or less. Quicksort was unaffected.

Realistic sorting problems are usually far from random. We sorted the 73,000 words of the Merriam-Webster Collegiate Dictionary, 7thedition, using the machines and compilers listed in Thble 5. 1. The word list consists mainly of two interleaved alphabetical lists, of capitalized and uncapitalized words. We sorted, into strict ASCII order,three input configurations: (1) as is, (2) two copies of the list concatenated, and (3) ordered by reversed spelling, which mixes the data well. The running times of programs A, B, and C were usually within a factor of 1.2 of each other, with no clear winner. American-flag sortwon consistently on the MIPS, two-array sort on the Cray, and list based on the Vax, a result roughly consonant with the degree of pipelining on the several machines. The list-based program was the most erratic. It lost consistently on the MIPS, and decisively-by a factor of 1.6-on some Cray and MIPS runs. As in Figure 5.1, quicksort fell far behind the radix sorts, usually by a factor of two or more.

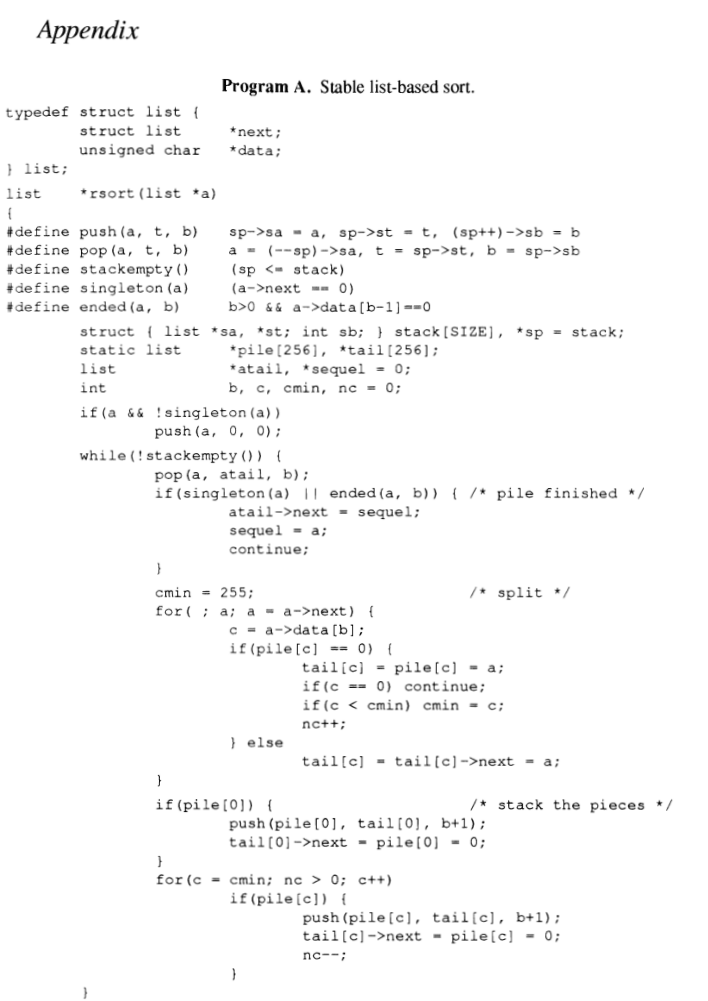
# 6. Discussion

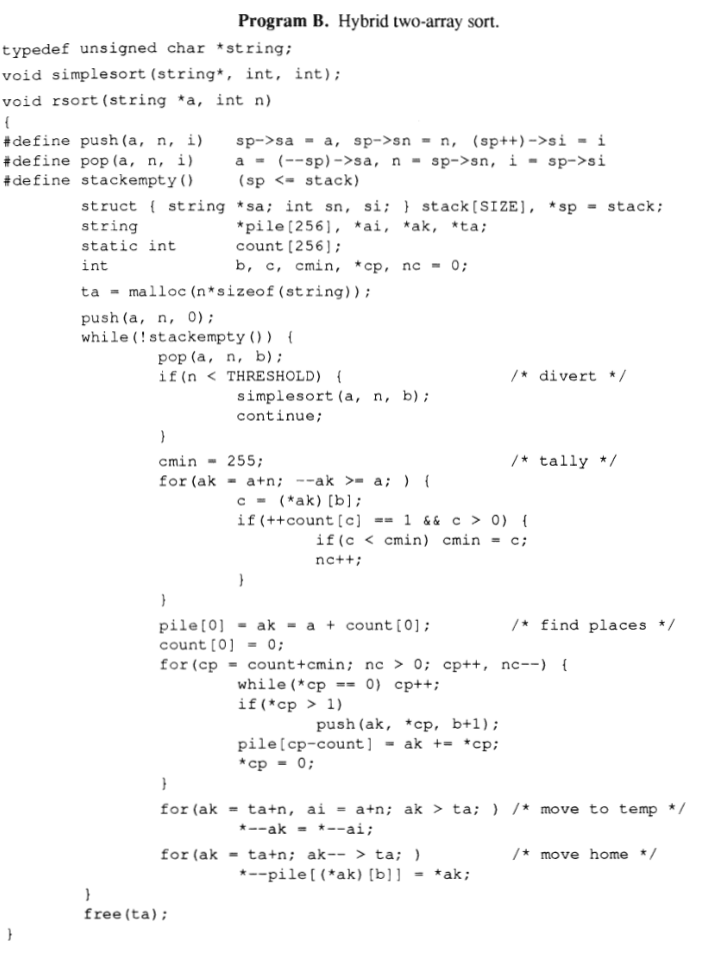
Our programs synopsize experiments that we have made jointly and severally over the past few years. Bostic wrote a two-array radix sort similar to Program B for the Berkeley BSD library, based in part on a routine by Dan Bernstein. P. Mcllroy adapted that routine for use in the BSD version of the Posix standard sort utility. P. Mcllroy also conceived American flag sort as a replacement for the two-array library routine. Independently, D. Mcllroy wrote a Posix utility around a list-based radix sort, and installed it on research systems at AT&T.Both the Berkeley and the AT&T utilities typically run twice as fast overall as the venerable quicksort-based programs that they replace.

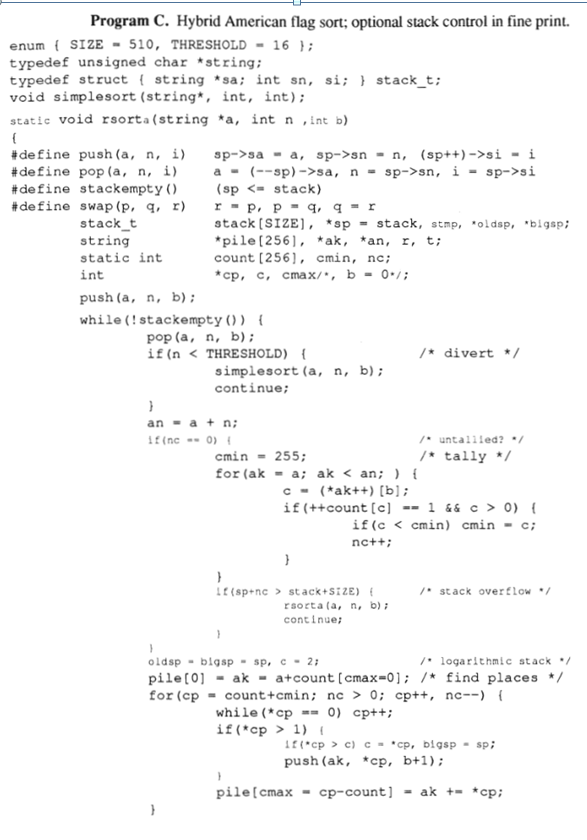
Although radix sorts have unbeatable asymptotic performance,they present problems for practical implementation: (1) managing scattered piles of unpredictable size and (2) handling complex keys. W'e have shown that the piles can be handled comfortably. Our utilities cope with complex keys by preconverting them into strings. Although it costs memory roughly proportional to the volume of keys, this strategy is simple and effective for sorting records after the fashion of the proposed Posix standard.

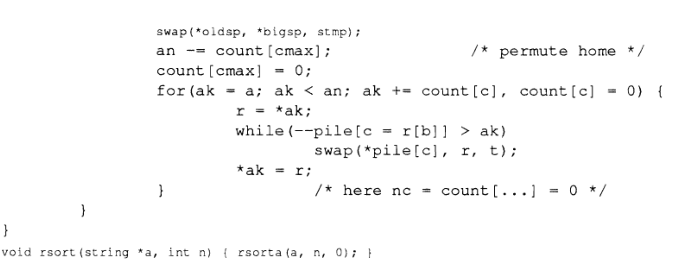
List-based radix sort is faster than pure array-based radix sorts.The speed disparity is overcome by hybrid routines that divert from radix sorting to simple comparison-based sorting for small arrays. Thenatural array-argument interface makes them attractive for library purposes. Both list-based and two-array sorts entail O(n) space overhead.That overhead shrinks to O(log n) in American flag sort, which, like quicksort, trades off stability for space efficiency.We recommend American flag sort as an all-round algorithm for sorting strings.

We have profited, even to the wording of our title, from the adviceand exemplary style of Jon Bentley.









# 7. Addendum

While this paper was in press, another radix sort appeared, recursive like Program 3.1, with diversion and in-place permutation. [. J.Davis, A fast radix sort, Computer J. 35 (1992) 636-642.1Although wasteful of storage, that program can be easily modified to run as fast as Program C, which stands as a good benchmark for radix sorting.

We are grateful to Peter McCauley for critical reading of our programs.

# References

1. Knuth, D 8., The Art of Computer Programming, 3, Sorting and Searching, Addison Wesley (1973).
2. Hildebrandt, P. and Isbitz, H., "Radix exchange-an internal sorting method for digital computers," JACM 6, pp. 156-163 (1959).
3. Bentley, J. L., Programming Pearls, Addison-Wesley (1986). Solution 10.6.
4. Dijkstra, E. W., A Discipline of Programming, Prentice-Hall (1976).
5. Hoare, C. A. R., "Quicksort," Computer Journal5, pp. 10-15 (1962).
6. Bentley, J. L., "The trouble with qsort," UnixReview L0(2), pp. 85-93 (Feb. 1992).
7. McCauley, P. 8., Sorting method and apparatøs, U.S. Patent 4,809,158 (1989).
8. Paige, R. and Tärjan, R. E., "Three partition refinement algorithms," SIAM J. Comput. 16, pp. 973-989 (1987).
9. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., The Design and AnaIysis of Computer Algorithrøs, Addison-Wesley (1974).
10. Fraser, C. W. and Hanson, D. R., "A retargetable compiler for ANSIC," ACM SIGPLAN Notices 26(10), pp. 29-43 (October l99I).
11. IEEE, Draft Standard for Information Technology-OperatingSystem Interface (POSX) Part 2: Shell and Utilities, Vol. P1003.2 /Dll (1991).
12. Mcllroy, P. M., "Intermediate files and external radix sort," submitted for publication.
13. Linderman, J. P., "Theory and practice in the construction of a working sort routine," AT&T Bell Laboratories Technical Journal 63, pp.1827-1844 (1984).[submitted Sept. 3, L992; accepted Oct. 3, 1992]

外文翻译

# 工程基数排序

Peter M. Mcllroy和Keith Bostic加州大学伯克利分校;道格拉斯·麦克罗伊和AT&贝尔实验室

摘 要

基数排序方法在字符串数据上有很好的渐近性能，因为比较不是单位时间的操作。在大字节可寻址存储器中使用这些方法很有吸引力，然而，这些方法长期以来一直被更容易的算法所掩盖。用实际的C程序说明了按字节从左到右排序字符串的三种方法——稳定的列表排序、稳定的双数组排序和就地“美国国旗”排序。对于繁重的排序，这三种排序的性能相当，通常运行速度至少是快速排序的两倍。我们建议一般使用美国国旗分类。

## **1. 介绍**

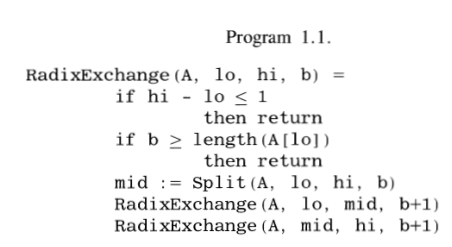
这个想法很简单。将字符串按首字母分组。一堆包含所有空字符串。下一个是所有以A-开头的字符串;另一个得到B，以此类推。在第二个和更多的字母上递归地拆分这些堆，直到字符串结束。当没有更多的堆要分开时，按顺序拣起所有的堆。字符串被排序。

理论上，基数排序是非常有效的。它在每个字符串中查找足够的字母，以便将其与其他字符串区分开来。没有办法检查更少的字母，同时仍然确保字符串被正确排序。但这一理论并不能说明全部情况:很难跟踪这些桩。

我们主要关心的是簿记，它可以使基数排序成为一种实用的方法，也可以使其失效。本文可以作为对Knuth第5.2章和5.2.5章中提出的练习的全面回答来阅读，其中列出了总体计划。[1] Knuth还描述了我们提到的其他经典排序方法:基数交换、快速排序、插入排序、Shell排序和小端基数排序。

## 1.1. 基数交换

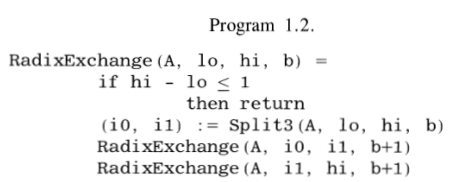
对于二进制字母表，基数排序专门用于基数交换的简单方法。我将字符串分成三堆:空字符串，以0开头的字符串，以及以1开头的字符串。对于经典的基数交换，进一步假设字符串都是相同的长度。这样空字符串就没有堆了，可以像在快速排序中那样进行分割，用位测试代替快速排序的比较来决定字符串属于哪个堆。

程序1.1对数组A中从A[lo]到A[hi - 1]的部分进行排序。例如，此范围内的所有字符串都具有相同的位前缀

x -split函数将前缀为- 0-的字符串从A[0]移动到数组的开头，从A[0]移动到A[mid - 1]，将前缀为- 1-的字符串移动到末尾，从A[0]移动到A[hi – 1]。要对包含n个元素的数组排序，调用



当字符串可以有不同的长度时，需要一个完整的三向分割，如程序I.2所示。一堆完成的字符串，值为x，从Allo]开始;→0-桩从Alí01开始;x 1 -堆从ali1开始。



三向分裂是荷兰国旗的著名问题:将三种混合颜色分成像国旗的红、白、蓝一样的条带。对于我们来说，这三种颜色分别是Ø(无位)、0和1。图1.1和程序1.3给出了分割的方法。索引10指向0-桩的开始，i指向0-桩的末尾，i2指向1-桩的开始。符号A[i]。表示字符串A[i]中从0开始计数的第位。当split3完成时，il指向所需的堆的开始。对Ø的检验是形象化的;它表示对字符串结束的测试。

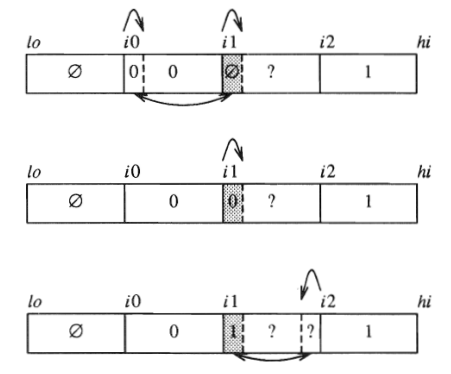
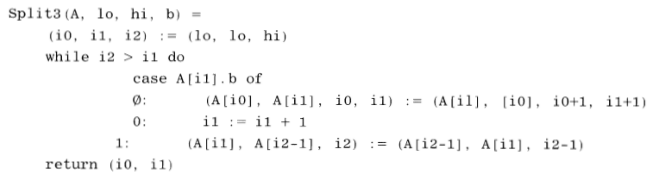


图1.1 split3的工作原理数组的四个部分包含已知结束的字符串(Ø)、已知在选定位置有0的字符串、未知字符串和已知在那里有I的字符串。反复查看第一个未知字符串(阴影框)的选定位置。根据匹配图进行更新。

Program 1.3.



## 1.2. 快速排序到前列

在享受了短暂的流行之后，基数交换被快速排序所取代。基数交换不仅通常比快速排序慢，而且不适合用Fortran或Algol编程，因为它们隐藏了它所依赖的位。快速排序以简单和几乎无敌而闻名;基数排序从教科书中消失了。

然而，就查看的数据量而言，基数交换再好不过了。快速排序甚至都不接近。快速排序平均对每个字符串执行O (log n)次比较，每次比较检查Ct(log n)位。通过这种度量，快速排序的预期运行时间为0(n log2 n)，对于基数交换仅为O(n log n)。更糟糕的是，快速排序可能是“二次的”，并且在不合适的输入上花费时间n 2log n)。这不仅仅是一个抽象的可能性。生产快速排序例程在完全合理的输入上已经变成了二次的。[6]

基数交换的理论优势通常被比特选取的成本所淹没。快速排序在几个方面更灵活。

快速排序可以使用机器指令来比较整个单词或字节，而不是位快速排序自适应拆分。因为它从数据中选择拆分值，所以即使在倾斜的数据上，快速排序也可以得到大约50-50的拆分。为了在快速排序或基数交换中实现最小的预期时间，这种分割是必要的。 快速排序可以排序任何东西，不仅仅是字符串。更改比较例程，它就可以处理不同的数据了。因为基数排序本质上假定字符串数据在有限的字母表上，所以它要求人们使数据适合例程，而不是相反。例如，要使用快速排序对日期进行排序，可能需要提供解析普通表示法的代码(例如9。二月日，1732)，而使用基数排序可以将所有日期预先转换为规范形式(例如I732O2II)。在其他方面，快速排序和基数交换非常相似。它们都在原地排序，占用很少的额外空间。这两种方法都需要一个递归堆栈，我们预计它的大小会增长到O (log n)。在这两种方法中，如果字符串很长或长度不同，最好通过统一的描述符来寻址字符串，并通过重新排列小描述符来排序，而不是大字符串。

快速排序的智慧可以追溯到小记忆时代。在更大的机器上，nlog n和nlog2n之间的差异变得更加显著。有了更大的机器，我们可以负担得起更大的空间。因此，这种智慧值得重新审视。

# 2. 基于列表的排序

对于大多数现代机器，8位字节是一个自然基数，这应该克服基数交换的拣位速度慢。字节基数可以进行256或257路分割，具体取决于如何确定字符串的结尾。这就提出了在递归的每一层上为这么多未知大小的堆管理空间的问题。链表数组是一种明显的数据结构。处理桩很容易;只要索引到数组。捡起排序好的一堆并将它们连接到一个单独的列表中有点棘手，但只需要很少的代码。

程序2.1完成了这项工作。它是用C而不是伪代码编写的，因为基数排序的问题是在实现中，而不是在概念上。输入变量是

a 以空结尾字符串的链表

b 要分割的字节的偏移量;字符串在所有早期字节中一致。

后续 一个排序后的链表，其中的字符串比较值大于列表A中的字符串。三个内联函数被编码为宏:

ended (a, b) 告诉字节位置b是否刚好超出字符串a末尾的空字节。.

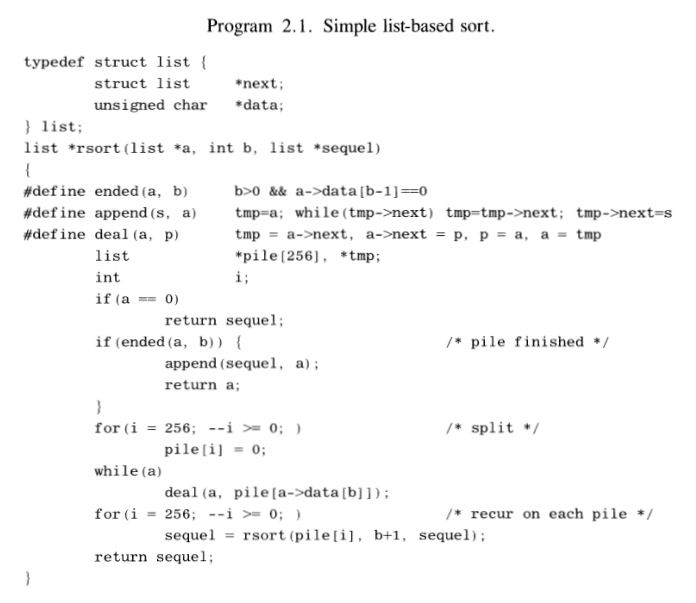
append (s, a) 将列表s追加到非空列表ø的最后一个元素

deal (a, p) 从列表a中移除第一个字符串并将其处理到堆p中.

程序2.1有四个部分。首先，如果列表为空，那么与sequel一起排序的结果是sequel。接下来，在“堆完成”时，如果在列表d中的字符串中看到的最后一个字节为空(0)，则不能对它们进行进一步排序。将它们放在续集前面，并返回合并后的列表。在“拆分”时，所有字符串都有第b个字节。清除所有的堆，然后根据每个字符串的字节b处理字符串。

最后，在“recurrent on each pile”中，从最后到第一个排序。在每个阶段，将从所有后续堆中累积的排序列表附加到已排序的当前堆中。

程序2.1工作缓慢。空桩是问题的根源。除了可能在递归的第一层或第二层，大多数堆将是空的。清理和递归“排序”的成本同样多



因为每个字节的255个空堆是压倒性的。一些简单的改进将加快速度。

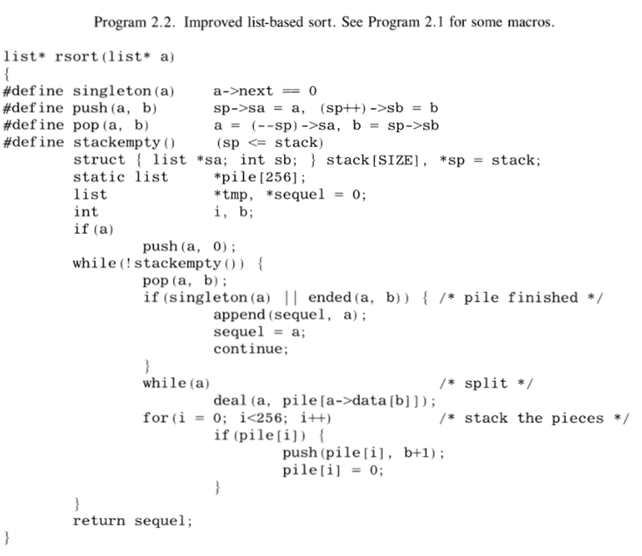
12.1总是发牌到相同的数组中，并且在发牌之间只清除已占用的牌堆，同时将已占用的牌堆移开

12.2直接管理堆栈。由于占用的桩的数量是不可预测的，并且除了递归的第一层或第二层之外可能很少，因此可以节省很多空间。这些堆可以按先到后的顺序堆叠，这样它们就会像程序2.1一样按后到先的顺序弹出。

12.3不要试图拆分一堆一堆的文件。

12.4合理优化:消除冗余计算;用指针替换下标。

12.5避免看空堆



程序2.2实现改进l1.2 - i2.3。在这里，携带状态的参数b和sequel被隐藏起来，这是它们应该做的。

在典型的ababetic数据上，Program2.2的运行速度大约是program2.1的15倍——从这种简单的优化中得到的回报还不错，但还不够好。程序2.2的大部分时间仍然浪费在扫描空堆上。因此，我们转向改进enti2.5，避免看到空堆，这可以通过许多方式实现

对于文本键，如名称或十进制数字，这些堆可能是聚在一起的。在256摞中，单字母只有26摞，数字只有10摞。为了利用聚类，我们使用一个简单的桩跨启发式方法:跟踪占用桩的范围。完成的0堆是一个预期的异常值，并被单独跟踪

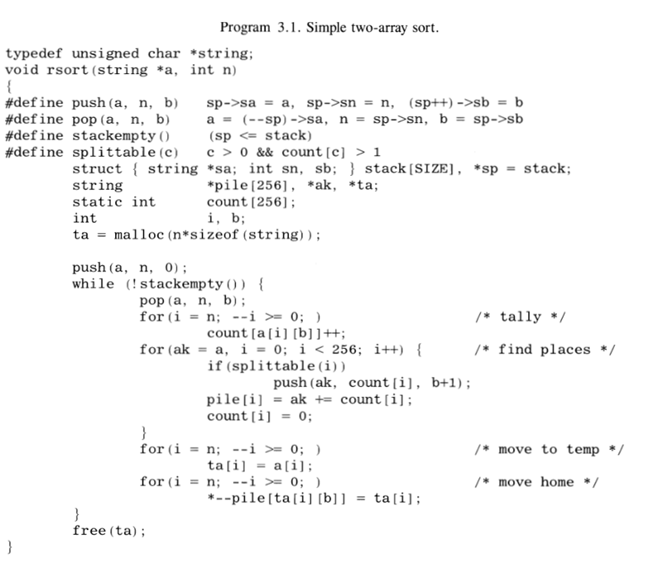
总之，在典型的输入上，12.l . 2.5的改进使程序2.1的速度提高了100倍。结果，附录中的程序A，是一个可靠的例程。它通常对包含10,000到100,000个键的数组进行排序，速度是竞争性快速排序的两倍

到目前为止，我们的程序都没有稳定排序。因为堆是通过将记录推到列表的前面来建立的，所以每次交易时等键记录的顺序是颠倒的。为了稳定排序，我们可以在追加时反转每个向后的堆。或者，我们可以通过跟踪每个桩的头部和尾部来保持桩的向前顺序。程序A是后者。正向/反向和稳定/不稳定版本之间的排序时间差异可以忽略不计。

# 3. 两个数组排序

假设字符串以arÍay形式出现，用于基数交换。在基本的基数交换中，两个桩位于数组的底部和顶部的已知位置。对于较大的基数，桩的位置可以在额外的传递中计算，计算每个桩中有多少个字符串。知道了桩的大小，我们不需要链表。

程序3.1通过将字符串作为一个块移动到辅助对象ta，然后将每个元素移动到适当的位置，从而将字符串返回。在阵列桩中预先计算位置的上端，如图4所示。1. (这种“向后”选择是为了与第4节中的程序保持和谐。)元素移动稳定;相等的元素保持它们在输入中的顺序。与程序2.2中一样，



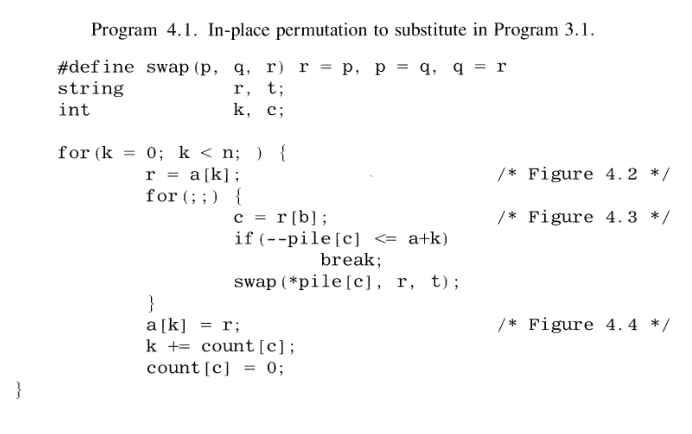
堆栈是显式管理的;堆栈还有第三个字段用来保存每个子数组的长度。

程序3.1适用于第2节中列出的大多数改进;它们出现在程序b中。此外，这些桩是独立的，不需要按顺序处理。也没有必要记录空桩的位置。这些观察结果体现在B计划的“找地方”步骤中

正如我们在介绍中观察到的，基数排序对于大型数组是最有利的。当堆变小时，我们可以使用简单的低开销的基于比较的排序方法，如插入排序或Shell排序。导流阈值在10 ~ 50之间效果较好;确切的值并不重要。附录中的程序B就是这样一个混合的双数组排序。它与基于列表的排序是竞争的;这两种方法中的哪一种胜出取决于所测量的计算机、编译器和测试数据。对于库而言，数组接口比列表接口更自然。但是双数组排序通过使用O(n)工作空间和动态存储分配来削弱这种优势。我们的下一个变体克服了这个缺点。

# 4. 美国国旗分类

与将数据复制到辅助数组并返回不同的是，我们可以对数据进行适当的排列。核心问题是将一个包含0到m - 1范围内的n个整数值的数组按升序重新排列，这是实际算法中的一个很好的练习。这里m是一个中等大小的值，在我们的例子中固定为256，而n是任意的。特殊情况是快速排序的分割步骤(m: 2)和荷兰国旗问题(m: -3)。通过与后者的类比，我们把一般问题称为美国国旗问题。(人们认为，许多条纹的标签很清楚，好像是美国最初联邦的几个州的名字。)

美国国旗排序与双数组排序的区别主要体现在最后阶段。程序3.1的“搬到临时住所”和“搬回家”阶段的效果是通过程序4.1和图4.1-4.4所示的“迁居”阶段实现的。这个阶段从顶部填充桩，通过在桩与桩之间循环置换构件来腾出空间。

设a[k]为第一桩中尚未完全就位的第一个元素。将该元素移出直线至r(图4.2)。

设c为位移单元所属的桩号。在c-桩中找到下一个未填充的位置，就在píle[c]下面[图4.3]。这个地方是流离失所者的家园。在更新pile[c]后，将移位的元素交换回家。

在新置换的元件上重复图4.3的操作，按照置换的循环进行，直到最终置换元件的家是循环开始的地方(例如)。将移位的元素移动到[k]。它的桩，当前的c-桩，现在被填充(图a)。通过增加ft跳转到下一摞的开始。(count数组中的值必须从“查找位置”阶段保留。)清除刚刚完成的一堆的计数，并开始另一个排列循环。当a + k= pile[c]开始时，也就是说，当堆已经到位时，很容易检查代码是否正常工作。

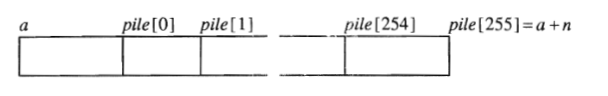
当除一桩以外的所有桩都已就位时，最后一桩也必须就位。程序4.2，或者说是程序4.1的浓缩版，利用了这个事实。方案4.2和方案B构成附录中方案C的基础。

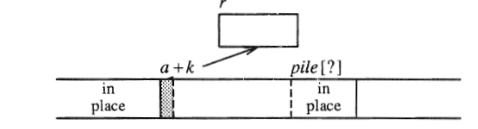
图4.1。排列回家前的数组

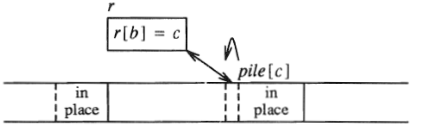
图4.2。第一次位移后。箭头表示已完成的操作。

图4.3。在位移周期中。r所指向的字符串的第b个字节是c。箭头显示要执行的操作，但在最后一次迭代中没有发生交换。

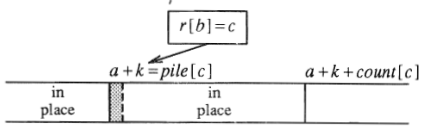
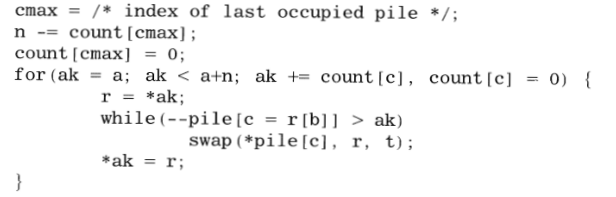


Figure 4.4. Last move.

4.2程序。改进了原地排列。

## 4.1. 堆栈增长

在目前的程序中，堆栈可能会随着运行时间线性增长。我们可以通过安排在每一层最后分割最大的堆来对数地限制增长——这是快速排序中众所周知的一个技巧。在基于数组的排序中，这种最大堆最后的策略很容易安装，但对于基于列表的排序就不太方便了，因为堆栈顺序很重要，而且堆大小不能自动获得。

即使使用对数边界，堆栈也可以相当大。在最坏的情况下，拆分不会生成0-桩，而是生成253个小桩(大小为2)，剩下的数据生成两个大小相等的大桩。其中一大堆立即从堆中弹出，并分裂成类似的。对于n: 1,000,000，最坏情况下的堆栈有2800个条目(在32位机器上为33,000字节)。相比之下，大约254logru n个条目的堆栈(当n: I,000,000时只有630个条目)足以存储统一的数据。更小的堆栈适用于实际数据。我们可以不使用最坏情况下的堆栈，而是分配一个短的堆栈，比如足以进行两层256路拆分，并在罕见的溢出情况下递归地调用例程

为了完整起见，程序C中展示了这两种堆叠策略，尽管在实践中几乎肯定不会用到它们。栈控制增加了三分之一的可执行代码，但只增加了1%的运行时间。

## 4.2 理货技巧

空桩处理的桩跨启发式假设了一个不不利分布的桥墩。其他避免看空堆的方法可以在文献中找到。一种方法是列出已占用物品的清单。每当字符串进入空堆时，将该堆记录在列表中。成交后，按桩号对已占用桩列表进行排序。如果列表太长，忽略它，扫描所有的堆。使用这种策略而不是桩跨的程序C的一个版本在不良数据上运行稍微快一些，但在合理数据上运行较慢。或者，可以在桩阵列上叠加占用树。然后，定位被占用桩的工作量将随着占用的减少而减少。在我们尝试过的数树计划中，最好的一个对树木的分布不敏感在不利的数据上，串和节拍桩跨是决定性的，但通常比桩跨慢三分之一到二分之一注意到只有身份而不是桩的顺序在分裂中起作用，Paige和Tarjan建议在所有分裂完成后，在一次合并中扫描桩。[8]他们的方法倾向于大基数;它在基数64K时比在基数256时运行得更快，不幸的是，开销——从4n到8n额外的内存字——淹没了理论上的优势。

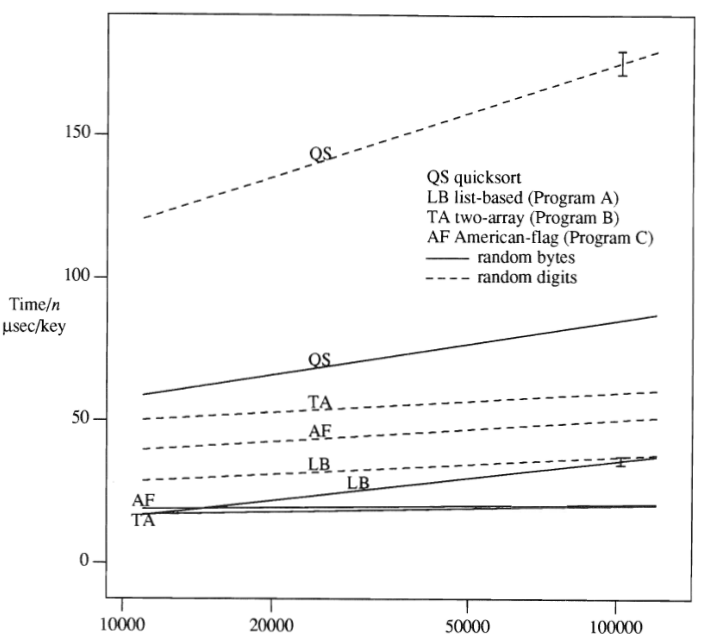
小端排序(最后一个字母优先)减轻了扫描空堆的问题。在小端排序中，拆分的次数等于最长键中的字母数量，而在像我们这样的双端排序中，拆分的次数通常超过键的数量。阿霍、霍普克罗夫特和乌尔曼展示了如何通过对所有键的所有字母进行O(“总大小”)排序来预测将发生什么摞，从而在每一副小端排序时消除摞扫描。然而，小端基数排序必须访问所有键的所有字母，而不仅仅是区分前缀的字母。

在实际操作中，对于较小的r，采用另一种算法来进行计数的奇异技巧是没有意义的。因为只有当n很小时，与处理和排列的时间相比，扫描堆的时间才明显。尽管如此，我们仍然喜欢极其便宜的桩跨启发式作为补充策略，因为它可以像IOVo一样提高运行时间，而不仅仅是转移。

# 5. 展示

这几个项目的优点必须根据优势而不是决定性的证据来判断。理论上，它们都具有相同的最坏情况渐近运行时间O(S)，其中。S是以字节为单位测量的数据大小。在实际操作中，也没有一个是明显占主导地位的。相关行为因数据、硬件和编译器而异

在评估性能时，我们将只考虑大型数据集，其中基数排序最有吸引力。通过与快速排序的比较可以看出它有多吸引人。经过测试的快速排序程序可以对字符串进行比较，选择一个随机分割的元素，并对小数组进行简单排序，它是从Bentley和Mcllroy的模型中专门设计出来的。这个套路不是最好的，但可能在C代码的最终速度的1 /3以内。(我们不愿采用他们最快的模型，因为它需要23个内联字符串比较。)

F图5.1显示了在一台计算机上对两种随机密钥(1)由8个随机十进制数字组成的字符串和(2)由长度指数分布，平均为9的随机字节组成的字符串，分别对4个例程中的每一个例程进行15次测试后的大小变化。这个实验的范围太窄，无法揭示快速排序的nlog2n偏离线性，或者完全平滑的量化效果。(在这个范围内的期望值

键数n

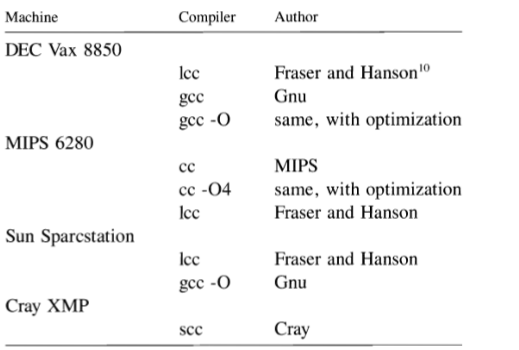
图5.1。最小二乘法适合于每个键的排序时间，而对于DEC VAX 8550来说是log n。有代表性的I表eÍÍor柱状图所示;其他曲线也差不多。

表5.1。测试的机器和编译器。

比较的长度对于十进制数据仅变化一个数字，对于字节数据仅变化半个字节。)然而，该图清楚地表明，基数排序的运行速度明显快于快速排序。这一观察结果是可靠的。对于这三种基数排序的相对性能，无法得出可比的概括。如图所示，排名顺序取决于数据的类型。其他实验也显示了硬件的类似变化

图5.1显示了基于列表排序的桩跨启发式算法对键分布的敏感性，其斜率变化很大。在另外两个动作中，转移几乎完全抑制了变化。

对于稀疏的随机数据，灵敏度甚至更高。在只包含两个字节值的随机键字符串的极端情况下，基于列表的程序A对具有远字节值的键进行排序的时间是对相邻值进行排序的时间的5倍。混合方案B和C的变化系数为1.2或更小。快速排序不受影响。

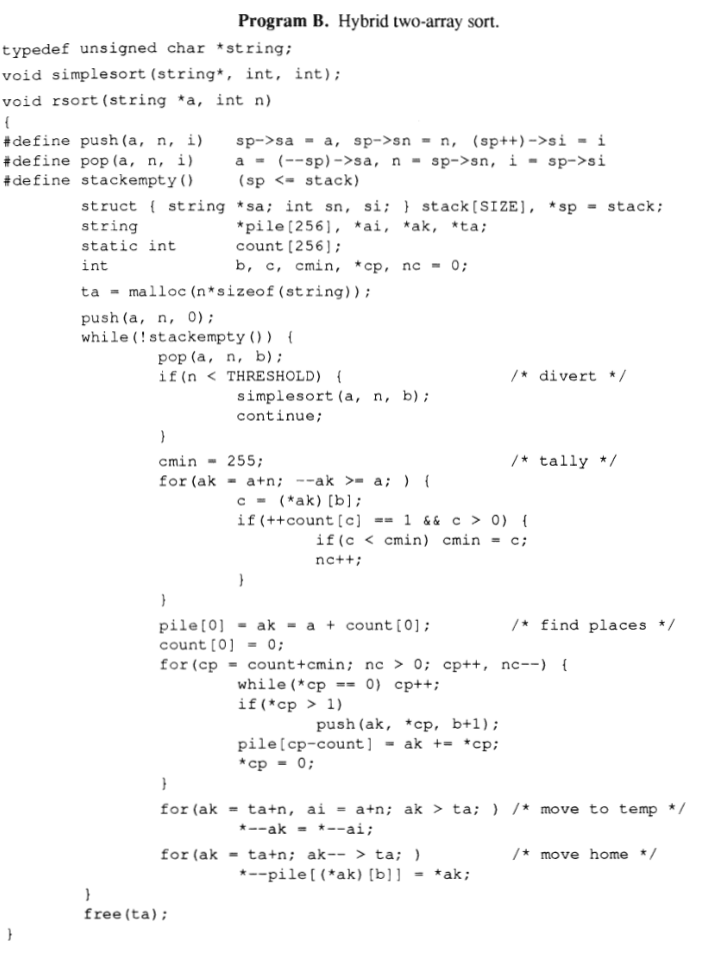
现实的排序问题通常远非随机的。我们使用表5中列出的机器和编译器对《韦氏大学词典》(the Merriam-Webster Collegiate Dictionary)第6版的73000个单词进行了排序。1. 单词列表主要由两个字母交错的列表组成，分别是大写单词和非大写单词。我们按照严格的ASCII顺序对三种输入配置进行了排序:(1)按原样排序，(2)将列表的两个副本连接起来，以及(3)按反向拼写排序，这样可以很好地混合数据。程序A、B和C的运行时间通常相差在1.2倍以内没有明确的赢家。美国国旗排序在MIPS上、在Cray上的双数组排序和在Vax上的列表上始终获胜，结果与几台机器上的流水线程度大致一致。基于名单的程序是最不稳定的。它在MIPS上持续失利，在一些Cray和MIPS上更是惨败1.6倍。如图5.1所示，快速排序远远落后于基数排序，通常落后于基数排序两倍或更多。

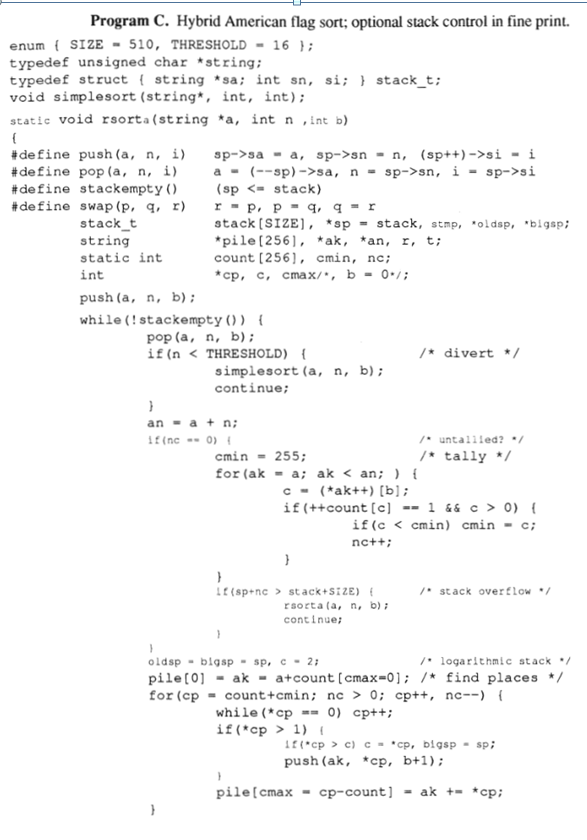
# 6. 讨论

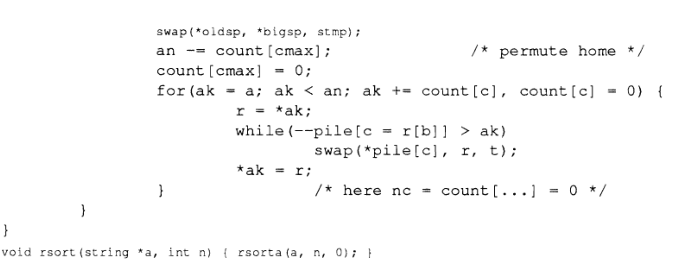
我们的程序概述了我们在过去几年中共同和单独进行的实验。Bostic为Berkeley BSD库编写了一个类似于程序B的双数组基数排序，部分基于Dan Bernstein的例程。P. Mcllroy改编了这个例程，以便在BSD版本的Posix标准排序实用程序中使用P. Mcllroy还设想用美国国旗排序来替代双数组库例程。D. Mcllroy独立地围绕基于列表的基数排序编写了一个Posix实用程序，并将其安装在AT&T的研究系统上。伯克利和美国电话电报公司的公用事业程序的总体运行速度通常是它们所取代的基于快速排序的老程序的两倍

虽然基数排序具有无与伦比的渐近性能，但它们在实际实现中存在问题:(1)管理不可预测大小的分散堆和(2)处理复杂的键。我们已经证明，这些堆是可以轻松处理的。我们的实用程序通过将复杂的键预转换为字符串来处理它们。尽管它消耗的内存大致与键的体积成正比，但对于按照Posix标准的方式对记录进行排序来说，这种策略简单而有效。

基于列表的基数排序比纯基于数组的基数排序快。这种速度差异可以通过混合例程来克服，这些例程将小数组从基数排序转移到简单的基于比较的排序。自然的数组参数接口使它们对库具有吸引力。基于列表和双数组的排序都需要O(n)空间开销。在美国国旗排序中，这种开销缩小到O(log n)，就像快速排序一样，为了空间效率而牺牲了稳定性。我们推荐美国国旗排序作为排序字符串的全面算法我们从乔恩·本特利的建议和模范风格中获益良多，甚至连我们标题的措辞都是如此。







# 7. 附录

在这篇论文出版的时候，另一种基数排序出现了，它像程序3.1一样是递归的，有转移和就地排列。Davis，快速基数排序，Computer J. 35(1992) 636- 6421 .尽管这个程序很浪费存储空间，但是可以很容易地修改，使其运行速度与C程序一样快，C程序是基数排序的一个很好的基准。

我很感谢彼得·麦考利对我们节目的批判性阅读。

# 引用

1. 张志强，张志强。，《计算机程序设计艺术》，3，排序与搜索，艾迪生·韦斯利(1973)。
2. 王志强，“基数交换-一种数字计算机内部排序方法”，《计算机科学》，第6期，pp. 56- 63(1999)。
3. 刘志强，程国强，程国强(1986)。10.6解决方案。
4. 李志强，程志强，程序设计理论与实践，计算机科学(2003)。
5. 刘志强，“快速排序”，《计算机学报》，第5期，第10-15页(1962)。
6. 张志强，“qsort的问题”，《计算机科学》vol . 2, pp. 85-93(1992年2月)。
7. 刘志强，陈志强。分选方法和装置，美国专利4809158(1989)。
8. 张志强，Tärjan，“三种划分优化算法”，中国计算机科学，第16卷，第973-989页(1987)。
9. 王晓明，王晓明，王晓明，一种基于算法的计算机算法设计[j] .计算机工程学报(自然科学版)(2004)。
10. 陈志强，“一种面向ANSI C语言的可重定向编译器”，《计算机工程学报》第26卷第10期，pp. 29-43(1999年10月)。
11. IEEE，信息技术-操作系统接口(POSX)第2部分:Shell和实用程序，Vol. P1003.2 /Dll(1991)。
12. 王晓明，“基于数据集的数据分类方法”，论文发表

[13]. 林德曼，“工作排序程序构建的理论与实践”，《美国电话电报公司贝尔实验室技术杂志》，第63页。1827 - 1844(1984)。