- 假设没有执行内联替换，则调用信息相当可靠。编译过的程序为每对调用者和被调用者维护一个计数器。每次调用一个过程时，就会对适当的计数器加 1。
- 默认情况下，不会显示对库函数的计时。相反，库函数的时间都被计算到调用它们的函数的时间中。

5.14.2　使用剖析程序来指导优化

作为一个用剖析程序来指导程序优化的示例，我们创建了一个包括几个不同任务和数据结构的应用。这个应用分析一个文本文档的 n-gram 统计信息，这里 n-gram 是一个出现在文档中 n 个单词的序列。对于 $n=1$，我们收集每个单词的统计信息，对于 $n=2$，收集每对单词的统计信息，以此类推。对于一个给定的 n 值，程序读一个文本文件，创建一张互不相同的 n-gram 的表，指出每个 n-gram 出现了多少次，然后按照出现次数的降序对单词排序。

作为基准程序，我们在一个由《莎士比亚全集》组成的文件上运行这个程序，一共有 965 028 个单词，其中 23 706 个是互不相同的。我们发现，对于 $n=1$，即使是一个写得很烂的分析程序也能在 1 秒以内处理完整个文件，所以我们设置 $n=2$，使得事情更加有挑战。对于 $n=2$ 的情况，n-gram 被称为 bigram(读作“bye-gram”)。我们确定《莎士比亚全集》包含 363 039 个互不相同的 bigram。最常见的是“I am”，出现了 1892 次。词组“to be”出现了 1020 次。bigram 中有 266 018 个只出现了一次。

程序是由下列部分组成的。我们创建了多个版本，从各部分简单的算法开始，然后再换成更成熟完善的算法：

1) 从文件中读出每个单词，并转换成小写字母。我们最初的版本使用的是函数 lower1(图 5-7)，我们知道由于反复地调用 strlen，它的时间复杂度是二次的。

2) 对字符串应用一个哈希函数，为一个有 s 个桶(bucket)的哈希表产生一个 $0 \sim s-1$ 之间的数。最初的函数只是简单地对字符的 ASCII 代码求和，再对 s 求模。

3) 每个哈希桶都组织成一个链表。程序沿着这个链表扫描，寻找一个匹配的条目。如果找到了，这个 n-gram 的频度就加 1。否则，就创建一个新的链表元素。最初的版本递归地完成这个操作，将新元素插在链表尾部。

4)一旦已经生成了这张表，我们就根据频度对所有的元素排序。最初的版本使用插入排序。

图 5-38 是 n-gram 频度分析程序 6 个不同版本的剖析结果。对于每个版本，我们将时间分为下面的 5 类。

Sort：按照频度对 n-gram 进行排序

List：为匹配 n-gram 扫描链表，如果需要，插入一个新的元素

Lower：将字符串转换为小写字母

Strlen：计算字符串的长度

Hash：计算哈希函数

Rest：其他所有函数的和

如图 5-38a 所示，最初的版本需要 3.5 分钟，大多数时间花在了排序上。这并不奇怪，因为插入排序有二次的运行时间，而程序对 363 039 个值进行排序。

在下一个版本中，我们用库函数 qsort 进行排序，这个函数是基于快速排序算法的[98]，其预期运行时间为 $O(n\log n)$。在图中这个版本称为“Quicksort”。更有效的排序算