法使花在排序上的时间降低到可以忽略不计，而整个运行时间降低到大约 5.4 秒。图 5-38b 是剩下各个版本的时间，所用的比例能使我们看得更清楚。

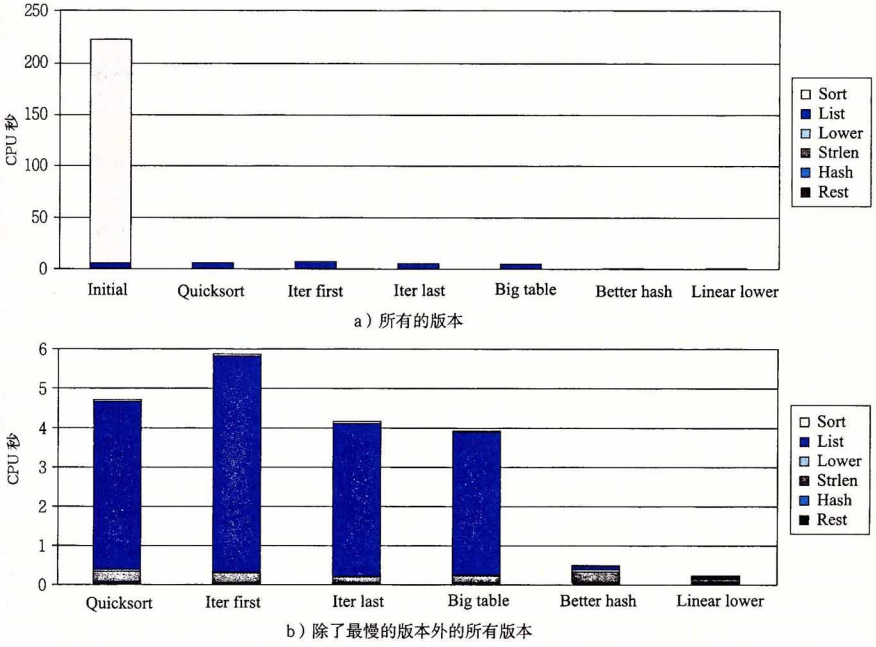

图 5-38　bigram 频度计数程序的各个版本的剖析结果。时间是根据程序中不同的主要操作划分的

改进了排序，现在发现链表扫描变成了瓶颈。想想这个低效率是由于函数的递归结构引起的，我们用一个迭代的结构替换它，显示为“Iter first”。令人奇怪的是，运行时间增加到了大约 7.5 秒。根据更近一步的研究，我们发现两个链表函数之间有一个细微的差别。递归版本将新元素插入到链表尾部，而迭代版本把它们插到链表头部。为了使性能最大化，我们希望频率最高的 n-gram 出现在链表的开始处。这样一来，函数就能快速地定位常见的情况。假设 n-gram 在文档中是均匀分布的，我们期望频度高的单词的第一次出现在频度低的单词之前。通过将新的 n-gram 插入尾部，第一个函数倾向于按照频度的降序排序，而第二个函数则相反。因此我们创建第三个链表扫描函数，它使用迭代，但是将新元素插入到链表的尾部。使用这个版本，显示为“Iter last”，时间降到了大约 5.3 秒，比递归版本稍微好一点。这些测量展示了对程序做实验作为优化工作一部分的重要性。开始时，我们假设将递归代码转换成迭代代码会改进程序的性能，而没有考虑添加元素到链表末尾和开头的差别。

接下来，我们考虑哈希表的结构。最初的版本只有 1021 个桶(通常会选择桶的个数为质数，以增强哈希函数将关键字均匀分布在桶中的能力)。对于一个有 363 039 个条目的表来说，这就意味着平均负载(load)是 363 039/1021＝355.6。这就解释了为什么有那么多时间花在了执行链表操作上了——搜索包括测试大量的候选 n-gram。它还解释了为什么性能对链表的排序这么敏感。然后，我们将桶的数量增加到了 199 999，平均负载降低到了