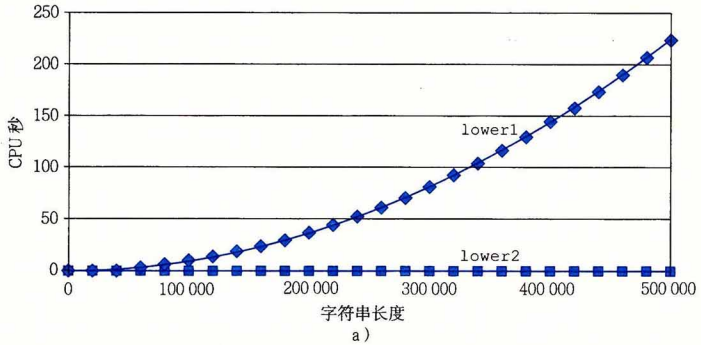
个序列，直到遇到 null 字符。对于一个长度为 $n$ 的字符串，strlen 所用的时间与 $n$ 成正比。因为对 lower1 的 $n$ 次迭代的每一次都会调用 strlen，所以 lower1 的整体运行时间是字符串长度的二次项，正比于 $n^2$。

如图 5-8 所示(使用 strlen 的库版本)，这个函数对各种长度的字符串的实际测量值证实了上述分析。lower1 的运行时间曲线图随着字符串长度的增加上升得很陡峭(图 5-8a)。图 5-8b 展示了 7 个不同长度字符串的运行时间(与曲线图中所示的有所不同)，每个长度都是 2 的幂。可以观察到，对于 lower1 来说，字符串长度每增加一倍，运行时间都会变为原来的 4 倍。这很明显地表明运行时间是二次的。对于一个长度为 1 048 576 的字符串来说，lower1 需要超过 17 分钟的 CPU 时间。




a)

| 函数 | 字符串长度 | | | | | | |
|---|---|---|---|---|---|---|---|
| | 16 384 | 32 768 | 65 536 | 131 072 | 262 144 | 524 288 | 1 048 576 |
| lower1 | 0.26 | 1.03 | 4.10 | 16.41 | 65.62 | 262.48 | 1 049.89 |
| lower2 | 0.0000 | 0.0001 | 0.0001 | 0.0003 | 0.0005 | 0.0010 | 0.0020 |

b)

图 5-8　小写字母转换函数的性能比较。由于循环结构的效率比较低，初始代码 lower1 的运行时间是二次项的。修改过的代码 lower2 的运行时间是线性的

除了把对 strlen 的调用移出了循环以外，图 5-7 中所示的 lower2 与 lower1 是一样的。做了这样的变化之后，性能有了显著改善。对于一个长度为 1 048 576 的字符串，这个函数只需要 2.0 毫秒——比 lower1 快了 500 000 多倍。字符串长度每增加一倍，运行时间也会增加一倍——很显然运行时间是线性的。对于更长的字符串，运行时间的改进会更大。

在理想的世界里，编译器会认出循环测试中对 strlen 的每次调用都会返回相同的结果，因此应该能够把这个调用移出循环。这需要非常成熟完善的分析，因为 strlen 会检查字符串的元素，而随着 lower1 的进行，这些值会改变。编译器需要探查，即使字符串中的字符发生了改变，但是没有字符会从非零变为零，或是反过来，从零变为非零。即使是使用内联函数，这样的分析也远远超出了最成熟完善的编译器的能力，所以程序员必须自己进行这样的变换。

这个示例说明了编程时一个常见的问题，一个看上去无足轻重的代码片断有隐藏的渐近低效率(asymptotic inefficiency)。人们可不希望一个小写字母转换函数成为程序性能的限制因素。通常，会在小数据集上测试和分析程序，对此，lower1 的性能是足够的。不过，当程序最终部署好以后，过程完全可能被应用到一个有 100 万个字符的串上。突然，