剖析报告的第一部分列出了执行各个函数花费的时间，按照降序排列。作为一个示例，下面列出了报告的一部分，是关于程序中最耗费时间的三个函数的：

```
  %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
97.58    203.66   203.66        1   203.66   203.66  sort_words
 2.32    208.50     4.85   965027     0.00     0.00  find_ele_rec
 0.14    208.81     0.30 12511031     0.00     0.00  Strlen
```

每一行代表对某个函数的所有调用所花费的时间。第一列表明花费在这个函数上的时间占整个时间的百分比。第二列显示的是直到这一行并包括这一行的函数所花费的累计时间。第三列显示的是花费在这个函数上的时间，而第四列显示的是它被调用的次数(递归调用不计算在内)。在例子中，函数 sort_words 只被调用了一次，但就是这一次调用需要 203.66 秒，而函数 find_ele_rec 被调用了 965 027 次(递归调用不计算在内)，总共需要 4.85 秒。函数 Strlen 通过调用库函数 strlen 来计算字符串的长度。GPROF 的结果中通常不显示库函数调用。库函数耗费的时间通常计算在调用它们的函数内。通过创建这个“包装函数(wrapper function)” Strlen，我们可以可靠地跟踪对 strlen 的调用，表明它被调用了 12 511 031 次，但是一共只需要 0.30 秒。

剖析报告的第二部分是函数的调用历史。下面是一个递归函数 find_ele_rec 的历史：

```
                                158655725             find_ele_rec [5]
                4.85    0.10  965027/965027       insert_string [4]
[5]      2.4    4.85    0.10  965027+158655725 find_ele_rec [5]
                0.08    0.01  363039/363039       save_string [8]
                0.00    0.01  363039/363039       new_ele [12]
                                158655725             find_ele_rec [5]
```

这个历史既显示了调用 find_ele_rec 的函数，也显示了它调用的函数。头两行显示的是对这个函数的调用：被它自身递归地调用了 158 655 725 次，被函数 insert_string 调用了 965 027 次(它本身被调用了 965 027 次)。函数 find_ele_rec 也调用了另外两个函数 save_string 和 new_ele，每个函数总共被调用了 363 039 次。

根据这个调用信息，我们通常可以推断出关于程序行为的有用信息。例如，函数 find_ele_rec 是一个递归过程，它扫描一个哈希桶(hash bucket)的链表，查找一个特殊的字符串。对于这个函数，比较递归调用的数量和顶层调用的数量，提供了关于遍历这些链表的长度的统计信息。这里递归与顶层调用的比率是 164.4，我们可以推断出程序每次平均大约扫描 164 个元素。

GPROF 有些属性值得注意：

- 计时不是很准确。它的计时基于一个简单的间隔计数(interval counting)机制，编译过的程序为每个函数维护一个计数器，记录花费在执行该函数上的时间。操作系统使得每隔某个规则的时间间隔 δ，程序被中断一次。δ 的典型值的范围为 1.0～10.0 毫秒。当中断发生时，它会确定程序正在执行什么函数，并将该函数的计数器值增加 δ。当然，也可能这个函数只是刚开始执行，而很快就会完成，却赋给它从上次中断以来整个的执行花费。在两次中断之间也可能运行其他某个程序，却因此根本没有计算花费。

  对于运行时间较长的程序，这种机制工作得相当好。从统计上来说，应该根据花费在执行函数上的相对时间来计算每个函数的花费。不过，对于那些运行时间少于 1 秒的程序来说，得到的统计数字只能看成是粗略的估计值。