**递归与非递归实现的性能比较及优化策略**

卢嘉霖1

（1.同济大学 软件学院，2050633）

**摘要：** 本文比较了5个经典递归算法分别使用递归函数、非递归函数实现的性能差异，并基于实验结果总结在实际开发中是否有必要将递归函数改写为非递归函数。对于每个递归算法，设计实验计算在不同数据规模下，递归实现和非递归实现的平均运行时间及两者比值。实验中主要的关注点是两种实现途径的时间复杂度，在时间复杂度接近时，根据实现逻辑指出两者空间复杂度优劣。实验结果显示，优化递归函数能否提升性能以及性能提升幅度的大小，取决于递归函数的特性：属于多重复递归函数、尾递归函数或是一般递归函数。对于前两者，优化为非递归实现能提升程序性能，但提升的幅度大小、节约的运行时间和占用内存亦有区别；对于最后一类，改写为非递归实现反而会降低性能。

**关键词：** 递归算法，递归函数，性能优化

**1 引言**

递归算法在算法设计中十分常见，其核心思想是重复将问题分解为类似的子问题从而解决问题，即“分而治之”。递归算法可以利用高级语言的递归函数特性实现，这种实现方法编码速度快、易于调试和维护、代码可读性好，但也存在着效率低下、递归栈占用内存等缺陷。在实际开发过程中，程序优化是提升系统性能的关键环节，而递归函数的优化是其中重要的一个方面。然而，据笔者所知，当前各大博客网站、论坛等并没有关于递归函数优化的优质阐述，而对于数据结构与算法初学者而言，容易陷入要么凡是递归实现一律改写成非递归实现、要么凡是递归算法一律用递归函数实现的误区。

针对这种现状，本文选取了5种经典递归算法，包括求斐波那契数列给定项、阶乘运算、幂值运算、二叉树的后序遍历、快速排序，设计实验比较了这些算法分别使用递归函数、非递归函数实现的性能差异。为了便于阐述实验结果，本文将递归函数分为3类：多重复递归函数，尾递归函数和一般递归函数。多重复递归函数具有大量重复计算特性，一般是多分支递归函数，即递归函数中调用一次以上自身，而其是否具有尾递归特性没有影响；尾递归函数具有尾递归特性，即递归函数中的所有调用自身语句都在函数末尾，且尾递归函数一定具有单分支特性；一般递归函数既不具有多重复特性、也不具有尾递归特性，其是否多分支不产生影响。

实验结果表明：多重复递归函数运行时间远大于非递归实现方法，在注重性能的工程中必须将这种递归函数改写为非递归函数；尾递归函数运行时间与其对应非递归函数接近，但考虑到后者空间复杂度更为优越，实际工程中将前者改写为后者也是有效的优化措施；一般的递归函数运行时间往往小于其非递归函数，此时应该使用递归函数进行实现。

总而言之，本文贡献和创新点如下：

1. 提出了一种递归函数的划分方法，这种方法依据递归函数是否大量重复计算和是否尾递归的特性进行分类。
2. 比较了5种经典递归算法的递归和非递归两种实现在不同数据规模下的平均运行时间，分析了两者的时间上和空间上的性能差异。
3. 基于上述递归函数分类方法和性能测试实验结果，提出了对递归函数进行优化的策略，该优化策略在实际工程开发中对性能提升具有极大实用价值。

**2 方法**

本文设计实验比较了5种递归算法分别使用递归函数、非递归函数实现的性能差异，测试了两种方法在不同数据规模下的平均运行时间。程序使用C++语言编写，分别实现了5种递归算法的递归和非递归实现，总计10个函数，并在主体程序中测得10个函数的平均运行时间。以下为本程序需要注意的声明：

1. 程序主要采用<time.h>头文件中的clock()函数测量时间，该函数返回的时间值只表示程序执行的CPU时间，不包括系统调用和等待时间，因此clock()函数测量的时间可能远小于实际感受的运行时间。
2. 重复实验次数约定为20次，利用以下宏定义规定该常量：

#define REPEAT 20

（3） 对于需要使用栈这一数据结构的地方，程序未采用标准C++库STL中的stack类模板，而是采用了笔者基于链表结构实现的定义在命名空间my\_std中的stack类模板。经测试该实现方式比采用标准C++库的stack类模板性能较低，但不存在数量级上的差距和时间复杂度的差异。若希望基于标准C++库实现，可包含头文件<stack.h>并将程序中的所有my\_std命名空间改为std即可。

**2.1 代码结构**

程序代码结构如图1所示。

（1）main.cpp是主函数所在文件，且定义了求斐波那契数列给定项、阶乘运算、幂运算、快速排序4种递归算法的递归和非递归两种实现的函数。在主函数中对每个待测试函数取不同数据规模下的参数，多次运行取平均运行时间，并将结果以表格的形式打印。

（2）bi\_tree.h定义了二叉树类BiTree，其主要方法：有接收节点个数参数生成一棵完全二叉树、递归法后序遍历二叉树和非递归法后序遍历二叉树。其中，为了节省输出到屏幕的时间开销，遍历过程不会对节点值进行打印。

（3）my\_stl.h实现了STL中的list、vector、string、queue、stack和pair类，基本采用了STL源码的实现机理、还原了STL的功能，具有良好的复用性、可读性、健壮性。本程序仅用到其中的stack类模板。

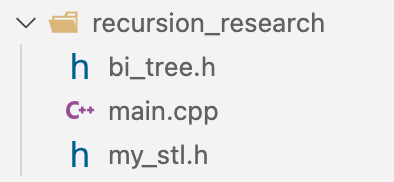


图1 项目代码结构

**2.2 主体程序算法**

本文选取了5种经典递归算法，即：求斐波那契数列给定项、阶乘运算、幂值运算、二叉树的后序遍历、快速排序。对于每一种算法，给定一组相距较大的参数，分别计算出每种数据规模下递归实现、非递归实现分别的运算时间及两者比值。主体程序核心算法伪代码如下所示，其中REPEAT为宏名，是为求平均值的重复实验次数，取值为20。rec\_func()和unrec\_func()分别为递归算法的递归实现函数、非递归实现函数。两种实现的运行时间rec\_time和unrec\_time实际计算方法与伪代码有一定区别。输出表格采用标准流IO，输出表格呈现在控制台。该程序段对上述5种递归算法分别实现一次，可输出5张表格，记录了两种实现在不同数据规模下的平均运行时间。

params = [p\_1, p\_2, p\_3, ...];

for(p in params)

{

// 计算递归实现平均运行时间

start = clock();

for(循环REPEAT次)

{

rec\_func(p);

}

end = clock();

rec\_time = (end - start) / REPEAT;

// 计算非递归实现平均运行时间

start = clock();

for(循环REPEAT次)

{

unrec\_func(p);

}

end = clock();

unrec\_time = (end - start) / REPEAT;

// 计算两者比值

ratio = rec\_time / unrec\_time;

在表格中输出rec\_time, unrec\_time, ratio三个值;

}

**2.3 递归算法的两种实现方法**

1. 求斐波那契数列给定项

递归函数接收整数n作为要求的斐波那契数列的项的下标，函数内部判定是否满足递归终止条件n ≤ 2，若满足则返回1，否则调用自身，求下标为n - 1和下标为n - 2两项之和并返回。具体代码如下：

int rec\_Fibonacci(int n)

{

if (n == 1 || n == 2)

return 1;

else

return rec\_Fibonacci(n - 1) + rec\_Fibonacci(n - 2);

}

非递归函数参数与递归函数一致。函数内部先定义了斐波那契数列前两项a、b，值为1，接着进行n - 2次循环，每次循环中计算a、b两项后面一项，并分别为a、b赋值等于其后面一项。循环结束后a、b后面的项即为要求的下标为n的项。具体代码如下：

int unrec\_Fibonacci(int n)

{

int a = 1, b = 1, c = 1;

for (int i = 3; i <= n; i++)

{

c = a + b;

a = b;

b = c;

}

return c;

}

1. 阶乘运算

递归函数接收整数n作为要做阶乘运算的数，函数内判定是否满足递归终止条件n == 1，若满足则返回1，否则调用自身，计算n与n - 1的阶乘运算结果之积并返回。具体代码如下：

int rec\_factorial(int n)

{

if (n == 1)

return 1;

else

return n \* rec\_factorial(n - 1);

}

非递归函数参数与递归函数一致。函数内部先定义了结果变量，初值为1，接着将循环变量i从1逐步自增到n来进行n次循环，每次循环中计算结果变量与当前循环变量的积。循环结束结果变量的值即为要求的n的阶乘运算结果。具体代码如下：

int unrec\_factorial(int n)

{

int result = 1;

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

result \*= i;

return result;

}

1. 幂值运算

递归函数接收整数x和整数n作为幂运算的底数和指数，函数内判定是否满足递归终止条件n == 0，若满足则返回1，否则调用自身，计算x与以x为底数、以n - 1为指数的幂运算结果之积并返回。具体代码如下：

int rec\_power(int x, int n)

{

if (n == 0)

return 1;

else

return x \* rec\_power(x, n - 1);

}

非递归函数参数与递归函数一致。函数内部先定义了结果变量，初值为1，接着将循环变量i从1逐步自增到n来进行n次循环，每次循环中计算结果变量与底数的积。循环结束结果变量的值即为要求的幂运算结果。具体代码如下：

int unrec\_power(int x, int n)

{

int result = 1;

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

result \*= x;

return result;

}

1. 二叉树的后序遍历

递归函数rec\_PostOrder是二叉树类BiTree的成员函数，不接收参数，内部仅将根节点作为参数调用其实现函数rec\_PostOrder\_impl，后者接收节点类指针参数root，函数内判定是否满足递归终止条件root为空指针，若满足则返回，否则分别将当前节点的左孩子、右孩子作为参数调用自身。为节省输出流打印节点数据的运行时间，此处的遍历不做任何操作，以强调递归函数本身的开销。具体代码如下：

void BiTree::rec\_PostOrder\_impl(Node \*root)

{

if (root)

{

rec\_PostOrder\_impl(root->lchild);

rec\_PostOrder\_impl(root->rchild);

// cout << root->data << endl;

}

}

void BiTree::rec\_PostOrder()

{

rec\_PostOrder\_impl(root);

}

非递归函数unrec\_PostOrder是二叉树类BiTree的成员函数，不接收参数。算法主要思路是用栈模拟递归，将后序遍历中较后访问的节点先存入栈中，如根节点、右孩子，在访问完需要优先访问的节点后从栈中弹出节点并访问，在栈为空且当前要访问节点为空时，即完成后序遍历。

函数内部先定义了两个节点指针p和q，分别赋初值为指向根节点的指针、空指针，以及一个模板参数为节点指针的栈s。接着进行循环，循环条件为p不为空指针或s不为空，换言之，当p为空指针且s为空时退出循环。在循环体中，先判断p是否为空指针，若不是空指针，则将p压入栈s中，p指向其左孩子，进入下次循环；若p是空指针，则从s的栈顶元素赋值给p，接着判断是否p有右孩子且该右孩子不是q，若是则p指向其右孩子，p压入栈s中，再将p指向其左孩子，进入下次循环，若不是则从s中弹出节点指针赋值给p，访问p指向的节点（此处省去访问操作以节约开销），p赋值给q，p赋值为空指针，进入下次循环。

具体代码如下：

void BiTree::unrec\_PostOrder()

{

Node \*p = root, \*q = nullptr;

my\_std::stack<Node \*> s;

while (p || !s.empty())

{

if (p)

{

s.push(p);

p = p->lchild;

}

else

{

p = s.top();

if (p->rchild && p->rchild != q)

{

p = p->rchild;

s.push(p);

p = p->lchild;

}

else

{

p = s.top();

s.pop();

// cout << p->data << endl;

q = p;

p = nullptr;

}

}

}

}

1. 快速排序

递归函数接收整数数组arr和整数left、整数right表示要处理的数组的范围，其基本思想是选择一个基准元素，将数组分成两个部分，一部分包含所有比基准元素小的元素，另一部分包含所有比基准元素大的元素。然后，对两个部分分别使用快速排序，直到每个部分只包含一个元素为止。

函数内先判断是否有递归终止条件left ≥ right，即数组当前部分已处理完毕，返回。否则，定义临时变量i、j分别赋值为left、right，以及基准元素pivot随意选取数组中的一个值（此处选取为left指向的值）。接着进入循环，不断调整数组中的元素使得小于pivot的值在一边、大于pivot的值在另一边，形成两个子数组。最后调用自身，参数变为两个子的最左、最右索引号。

具体代码如下：

void rec\_quickSort(int \*arr, int left, int right)

{

if (left >= right)

return;

int i = left, j = right, pivot = arr[left];

while (i < j)

{

while (i < j && arr[j] >= pivot)

j--;

if (i < j)

arr[i++] = arr[j];

while (i < j && arr[i] <= pivot)

i++;

if (i < j)

arr[j--] = arr[i];

}

arr[i] = pivot;

rec\_quickSort(arr, left, i - 1);

rec\_quickSort(arr, i + 1, right);

}

非递归函数参数与递归函数一致。其主要思想与递归法基本一致，只是使用栈代替递归。区别在于标识最左、最右元素索引号的left、right以栈元素的形式被每次子数组处理过程使用、更新，而非以函数参数的形式传递。

函数内先定义栈s存放边界索引号，然后将数组的最高位、最低位索引压入栈中。接着进入循环，该循环退出条件是栈s为空。在循环体中，先从栈s中弹出两个元素作为当前要处理数组的边界索引号，并类似地取一个值作为基准元素pivot。然后将数组划分为两个子数组，其中元素分别都小于pivot、都大于pivot。然后将子数组的边界索引号压入栈s中，每次压栈需要压入一对元素（分别为下次循环的left、right）。

具体代码如下：

void unrec\_quickSort(int \*arr, int left, int right)

{

my\_std::stack<int> s;

s.push(left);

s.push(right);

while (!s.empty())

{

int r = s.top();

s.pop();

int l = s.top();

s.pop();

int i = l, j = r, pivot = arr[l];

while (i < j)

{

while (i < j && arr[j] >= pivot)

j--;

if (i < j)

arr[i++] = arr[j];

while (i < j && arr[i] <= pivot)

i++;

if (i < j)

arr[j--] = arr[i];

}

arr[i] = pivot;

if (i - 1 > l)

{

s.push(l);

s.push(i - 1);

}

if (i + 1 < r)

{

s.push(i + 1);

s.push(r);

}

}

}

**2.4 递归函数的分类**

本文的创新点之一是提出了一种递归函数的分类方法。该分类方法的依据是递归函数的两个特性：是否大量重复计算和是否尾递归。本文测试的5种递归算法对应的递归函数实现中，5个递归函数可以划分为以下3个类别：

1. 多重复递归函数：运行过程中存在大量重复计算，一般具有多分支特性，而其是否具有尾递归特性没有影响。其递归树中存在大量相同节点，如图2所示，诸如fib(1)、fib(2)等节点多次出现，意味着多重复递归函数运行过程中存在重复函数调用。多分支特性是指递归函数中调用一次以上自身，多分支递归函数的递归树是多叉树。本文的求斐波那契数列给定项的递归函数实现是多重复递归函数。

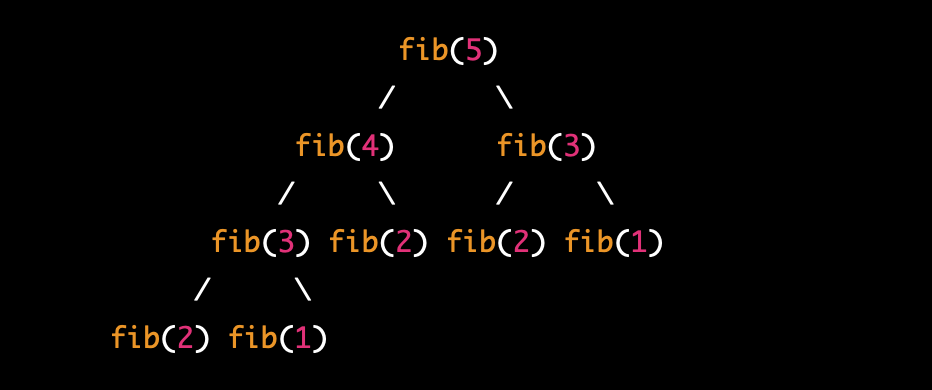


图2 多重复递归函数的递归树

1. 尾递归函数：具有尾递归特性，即所有调用自身语句都在函数末尾，因此尾递归函数一定具有单分支特性，其递归树通常只有一条链，如图3所示。尾递归函数一定不存在大量重复计算。本文的阶乘运算和幂运算的递归函数实现都是尾递归函数。

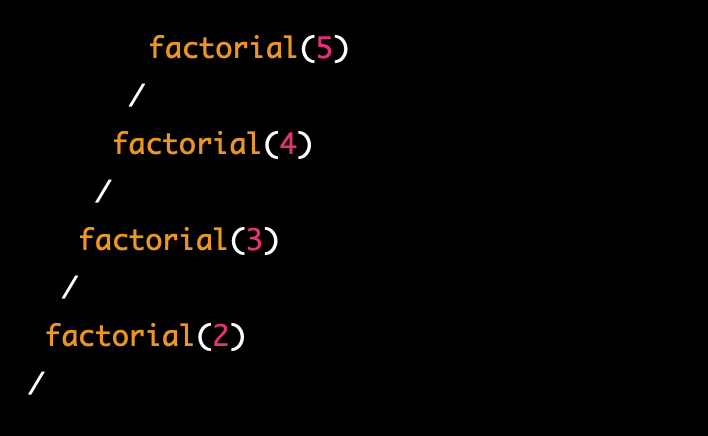


图3 单分支函数的递归树

（3）一般递归函数：不具有尾递归特性，且无论是多分支还是单分支，不会出现大量重复计算，即递归树的节点两两互异。本文的二叉树的后序遍历和快速排序的递归函数实现都是一般递归函数。

**3 实验结果及分析**

对上文5种递归算法分别实现主体程序中的算法后，程序输出5张表格，记录了该算法两种实现在不同数据规模下的平均运行时间。其中每张表格的表头指出了递归算法；第一列是函数参数，具有不同数据规模，其取值考虑了测试时间的有限性和计算结果是否超过其数据类型表示范围；第二、三列分别是递归函数、非递归函数实现的平均运行时间，单位为毫秒（ms），在实际运行过程中可能出现时间太短而测得为0的情况，可以通过多次运行或增大重复实验次数来避免；第四列是前面两列数值之比，值越大表示非递归函数实现的性能越优于递归函数实现。下面根据递归函数的三种分类对输出表格进行分析。

测试环境：

OS ：macOS Monterey (Version 12.6)

编辑器：Visual Studio Code 1.68.1(Universal)

编译器：Apple clang version 13.1.6 (clang-1316.0.21.2.5)

**3.1 多重复递归函数**

本文测试的多重复递归函数为求斐波那契数列给定项的递归函数实现。测试结果表格如图4所示。

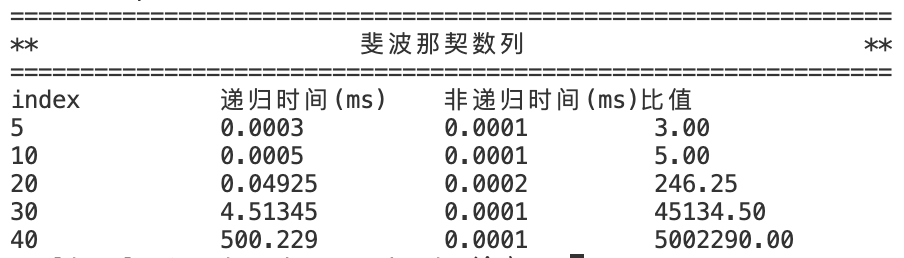
****

图4 求斐波那契数列给定项测试结果

由表格横纵比较可知，随着数据规模增大，递归函数运行时间急剧增长，从index=5到index=40耗时增加了上百万倍，而非递归函数运行时间基本不变。观察两者比值，容易发现递归函数性能始终差于非递归函数，且数据规模越大差距越大。这表明该算法的两种实现方法具有不同的时间复杂度，且非递归法要远远优于递归法。

分析算法可知，斐波那契数列的递归函数时间复杂度为O(n2)，非递归函数时间复杂度为O(n)。从2.4节的递归树中可以看出，求斐波那契数列给定项的递归函数实现会进行大量重复计算，这是因为它在调用自身时没有考虑到利用已知或计算过的数据，两次调用自身带来的后续计算存在大量重叠部分。而非递归函数实现则避免了这一点，定义3个存储中间量临时变量和利用循环语句不断更新，逐步计算出结果，不会产生大量重复计算，节约了时间开销。另外，递归函数需要递归栈的大量空间开销，且由于不满足尾递归特性而无法被编译器优化，数据规模越大，递归树节点越多，递归栈空间开销越大；而非递归函数的空间开销为常数且可以忽略不计。

综上，多重复递归函数的性能远远低于其非递归函数实现，实际开发中进行改写非常有必要。

**3.2 尾递归函数**

本文测试的尾递归函数为阶乘运算和幂运算的递归函数实现。测试结果表格如图5和图6所示。

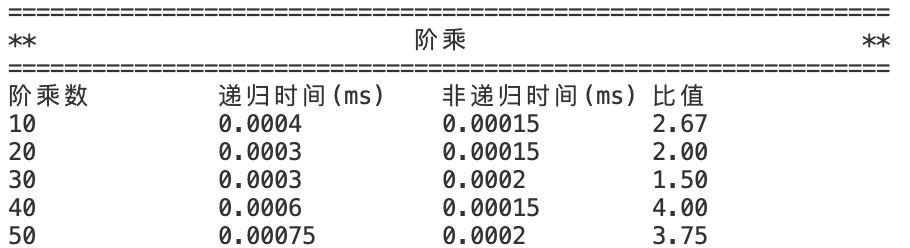
****

图5 阶乘运算测试结果

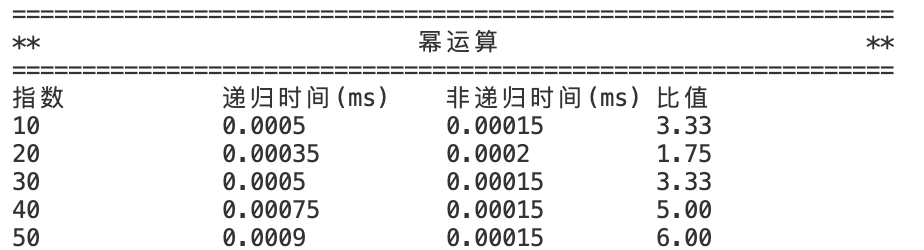
****

图6 幂运算测试结果

由表格横纵比较可知，随着数据规模增大，递归函数和非递归函数平均运行时间比值变化幅度不大，可以认为两者时间复杂度相当。但从该比值始终大于1可以得知，递归函数的时间开销仍然大于非递归函数。这是因为尾递归函数在进行递归调用之前不需要保存任何信息，因而编译器会对尾递归函数进行优化，在进行递归调用时，直接重用当前函数的执行环境，而不是将它保存在调用栈中。但对比非递归函数，递归函数进行了较多次函数调用，而函数调用本身是存在较大时间花费的，因此递归法运行时间要长于非递归法。

分析算法可知，阶乘运算和幂运算的递归和非递归函数时间复杂度都为O(n)。从空间开销角度看，本文实验中两个递归算法的递归函数实现和非递归函数实现都具有常数的空间复杂度：递归函数不需要递归栈保存信息，非递归函数也不需要使用栈保存任何变量。但是，在实际开发中可能遇到这样的递归函数，它本身是尾递归函数，因而不需要递归栈的空间开销，但将其改写为非递归函数后，需要用到栈保存大量信息，这时递归法的空间复杂度优于非递归法。

综上，尾递归函数的时间开销与其非递归函数实现差距不大，但运行时间仍然较后者长。尾递归函数会被编译器优化，不需要像其他递归函数一样使用递归栈，空间开销较小。在实际开发中，如果尾递归函数及其非递归函数具有相同空间开销，且编译器支持对尾递归函数的优化，则可以出于性能优化考虑将递归函数改写为非递归函数，也可以出于程序可读性、开发效率等考虑使用递归函数实现。

**3.3 一般递归函数**

本文测试的一般递归函数为二叉树的后序遍历和快速排序的递归函数实现。测试结果表格如图7和图8所示。

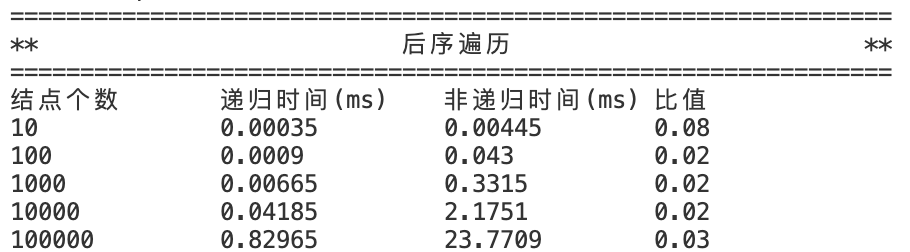
****

图7 二叉树的后序遍历测试结果

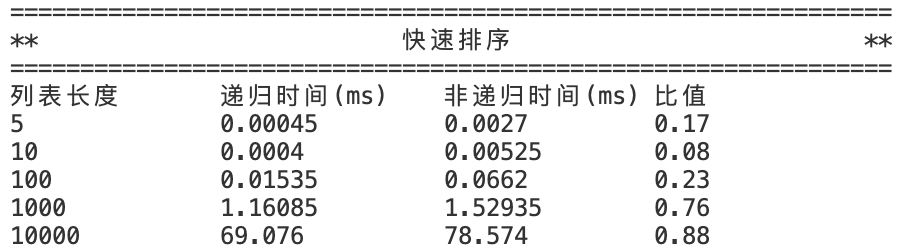
****

图8 快速排序测试结果

由表格横纵比较可知，一般递归函数的平均运行时间小于非递归函数实现，时间开销比值从几倍到几十倍不等，且不会随着数据规模变化。这是因为递归函数可以利用系统的缓存机制，在处理大量数据时让程序执行效率更高。非递归函数使用栈保存大量临时变量以模拟递归栈保存函数调用环境，在处理大量数据时系统缓存机制无法发挥作用，从而降低了程序性能。从空间开销角度考虑，递归函数需要递归栈的空间，非递归函数需要手动开辟栈空间，两者的空间开销都随着数据规模增大而增大，难以判断优劣。此外，递归函数具有编码速度快、代码可读性好、易于维护和扩展等显著优势，这些是非递归函数所不具备的。

综上，一般递归函数的性能优于其非递归函数实现，且鉴于递归法具有的其他优势，在实际开发中不建议将这类递归函数改写为非递归函数。

**4 结论**

本文选取了5种经典递归算法，包括求斐波那契数列给定项、阶乘运算、幂值运算、二叉树的后序遍历、快速排序，设计实验比较了这些算法分别使用递归函数、非递归函数实现的性能差异。为了便于阐述实验结果，本文将递归函数分为3类：多重复递归函数，尾递归函数和一般递归函数。实验结果表明，多重复递归函数的时间复杂度远高于其非递归实现，将这类递归函数改写为非递归函数可以极大提升程序性能，且数据规模越大，性能提升幅度越大。尾递归函数的时间复杂度和非递归函数实现相当，但前者实际运行时间仍然长于后者，如果程序性能需要优先考虑或编译器不支持尾递归优化，则在实际工程中将尾递归函数改写尾非递归实现也不失为明智之举。而对于不具有大量重复计算和尾递归特性的一般递归函数，改写为非递归实现会导致程序无法利用系统的缓存机制，反而极大降低了其性能，这种情况下使用递归函数实现已经是最佳选择。

综上所述，在实际开发和性能优化过程中，要避免陷入要么凡是递归算法都用递归函数实现、要么凡是递归函数都改写为非递归函数的误区中。最重要的是依据递归函数的特性判定其类别，明确将其改写为非递归函数是否能提升程序性能，再作出抉择。