论反汇编在时间常数优化中的应用

四川省成都七中　　周以苏

摘要：本文阐述了时间常数优化的重要性，并且在 Visual C++语言环境下，从特定编译器生成的汇编代码出发，探讨了反汇编在时间常数优化中的应用，提出了若干优化改进方案。

关键字：C++ 反汇编 时间常数 优化

程序的优化是无止境的。其中，时间常数是在相同时间复杂度下决定程序运行快慢的关键。然而在竞赛中，渐进时间复杂度是关注的重点，而同样能够决定程序运行快慢的时间常数优化问题却缺乏重视。本文阐述了时间常数优化的重要性，并且在 Visual C++语言环境下，从特定编译器生成的汇编代码出发，探讨了反汇编在时间常数优化中的应用，提出了若干优化改进方案。

一、基本概念

汇编语言（Assembly language），是一种与硬体紧密相关的程序设计语言，是一种低级语言。汇编语言是机器语言的便于记忆和理解的符号化形式。

由于汇编语言的各种语法规范互不兼容，本文以下部分以 Intel 语法为基础，并以使用 Intel 语法的 Visual C++作为实验平台。

汇编语言源程序是用伪代码创建的。一个伪代码是一条处理器可以理解的指令。例如：

```
add
```

add 指令把两个数加到一起。大部分伪代码带有参数

```
add eax, edx
```

add 有两个参数。一个源参数，一个目标参数。它把源值加到目标值中，并把结果保存在目标中。参数有很多不同的类型，如：寄存器，内存地址，直接数值。下面的介绍以寄存器为例。

有几种大小的寄存器：4 位，8 位，16 位，32 位（在 MMX 处理器中有更多的种类）。在 16 位程序中，你仅能使用 4 位、8 位和 16 位的寄存器。在 32 位的程序中，你还可以使用 32 位的寄存器。

一些寄存器是别的寄存器的一部分：例如，如果 eax 保存了值 EA7823BBh，那么，它的子寄存器 ax，ah 和 al 也会保存其值的一部分，如下表所示：

```
eax             EA 78 23 BB
```

```
ax      23 BB
ah      23
al      BB
```

ax，ah，al 是 eax 的一部分。eax 是一个 32 位的寄存器（仅在 386 以上存在），ax 包含了 eax 的低 16 位（2 字节），ah 包含了 ax 的高字节，而 al 包含了 ax 的低位字节。因而 ax 是 16 位的，al 和 ax 是 8 位的。

让我们来分析下面的代码：

```
mov     eax, 12345678h
        ;Mov 把一个值载入寄存器（注意：12345678h 是一个十六进制值，因为 h 这个后缀。
mov     cl, ah
        ;把 ax 的高字节移入 cl
sub     cl, 10
        ;从 cl 的值中减去 10（十进制）
mov     al, cl
        ;并把 cl 存入 eax 的最低字节
```

mov 指令可以把一个值从寄存器，内存或直接数值移入另一个寄存器。在上面的例子中，eax 包含了 12345678h，然后 ah 的值（eax 左数第三个字节）被复制入了 cl 中（ecx 寄存器的最低字节）。然后，cl 减 10 并移回 al 中（eax 的最低字节）

关于汇编指令的其它基础知识，请参看相关的入门书籍。

编译器(Compiler)，是将便于人编写，阅读，维护的高级计算机语言程序翻译为计算机能识别，运行的低级机器语言的程序。编译器将源程序（Source program）作为输入，翻译产生使用目标语言（Target language）的等价程序。源程序一般为高级语言（High-level language），如 Pascal，C++等所写的程序，而目标程序则是机器能直接执行的目标代码（Object code），有时也称作机器代码（Machine code）。

编译器友好(Compiler friendly)，是指所写的代码符合编译器的规范，其功能的完整性不依赖于特定编译器的特性或隐藏功能，因此具有通用性和兼容性。

本文的结论不一定需要用汇编实现，因为嵌入汇编不是所有语言都能够支持的，况且这样做不是文章的本意。

二、数据分析

在特定的平台上，不同的汇编伪代码有着不同的执行速度。但是，每种功能的伪代码的相对执行速度是较为恒定的。比如，执行乘除法伪代码的时间一定大大长于执行加减法伪代码的时间。

表 1 为 Aoa[1]中 8286 系统各种伪代码的理论执行时间：

表 1：8286 指令集的执行时间

Instruction ⇒ Addressing Mode ⇓	mov (both forms)	add, sub, cmp, and, or,	not	jmp	jxx
reg, reg	2	4			
reg, xxxx	1	3			
reg, [bx]	3-4	5-6			
reg, [xxxx]	3-4	5-6			
reg, [xxxx+bx]	4-5	6-7			
[bx], reg	3-4	5-6			
[xxxx], reg	3-4	5-6			
[xxxx+bx], reg	4-5	6-7			
reg			3		
[bx]			5-7		
[xxxx]			5-7		
[xxxx+bx]			6-8		
xxxx				1+pfd[a]	2[b] 2+pfd

为了验证上表的结论以及探讨从反汇编角度对程序进行改进的可行性，本文对实际语句的运行情况进行了数据搜集。

本文采用的实验环境如下：

试验环境：

软件： Windows XP sp2 050301-1519 MSVC++ 69514-335-0000007-18650

硬件： Intel Pentium 4 CPU 2.66GHz

Debug 模式等同于非优化模式等同于不带参数进行编译

Release 模式等同于优化模式约等于 G++中-O2 优化

注意：本文的某些结论并不适用于其他处理器、平台和编译器，需要再次进行测试。

测试方法：在速度测试中，实验使用了一个基准的 C++程序模版。

```
#include <time.h>
#include <stdio.h>
clock_t start,finish;
int a;
int main()
{
```

```
    start=clock();
    __asm
    {
        mov ecx,2000000000    //实验次数
testBegin:
        mov eax,10
        mov ebx,10
        cdq
//这里插入实验语句
        dec ecx
        jz testEnd
        jmp testBegin
testEnd:
    }
    finish=clock();
    printf("%.3lf",double(finish-start)/CLOCKS_PER_SEC);
    return 0;
}
```

本文对附录中 Configure.in 文件中引用的操作进行了测试。通过执行附录中的 Python 脚本，得出了表 2 的结果：

表 2：在测试环境下各个指令实际运行时间及比例

测试操作	平均用时	相对用时	比例
EMPTY（无实验语句）	2.29	0.00	0.0
NORMAL IMUL	3.81	1.52	11.0
NORMAL IDIV	25.94	23.65	171.2
NORMAL TEST	**2.43**	**0.14**	**1.0**
NORMAL MOV	2.29	0.00	(0.0)
NORMAL ADD	2.41	0.12	0.9
NORMAL INC	3.05	0.76	5.5
NORMAL SUB	2.41	0.12	0.9
NORMAL DEC	3.05	0.76	5.5
NORMAL XOR	2.35	0.06	0.4
NORMAL AND	2.35	0.06	0.4
NORMAL SHL	2.44	0.15	1.1
NORMAL SAR	2.44	0.15	1.1
NORMAL SHR	2.44	0.16	1.1
NORMAL NOT	2.29	0.00	(0.0)
NORMAL XCHG	3.04	0.76	5.5
PTR MOV	2.28	0.00	(0.0)
PTR ADD	3.05	0.76	5.5
PTR SUB	3.05	0.76	5.5
PTR XOR	3.05	0.76	5.5
PTR AND	3.05	0.76	5.5
PTR XCHG	67.39	65.10	471.2
JMP	3.14	0.85	6.2
PUSH POP	3.05	0.76	5.5

CALL RET JMP	9.16	6.88	49.8
EMPTY(Verify)	2.29	0.00	0.0

注：

1、测试结果有误差，并且会随着操作系统、编译器和系统配置的变化而不同程度的波动。

2、具体测试语句见附录 Configure.in

3、由于处理器对程序的执行并不是像想象中那样一句一句直接执行，其中还涉及到代码解释缓存、L1、L2 变量长度(16,32)等因素的影响，本试验的数据只作为理论的近似。

基于上面得到的结果，本文将各类语句时间常数归类为了下列 6 个层次（见表 3）：

表 3：汇编语句速度分类

层次	运算	比例时间
1、	**mov**，**lea**（未进行测试）数据移动运算	1
	and or xor not 逻辑运算	
	add，**sub** 加减法运算，**test** 置位运算（内部为 sub 运算）	
2、	**shl**，**shr**，**sal**，**sar** 位运算	1.5~2
3、	ptr 取地址值（不是运算），**push+pop** 堆栈运算*2，**jmp** 跳转运算	4
4、	**mul**，**imul** 乘法	5
5、	**div**，**idiv** 除法	25
6、	**call+ret** 调用子函数+返回	27

从这 6 个层次中，我们可以看到汇编代码执行的快慢程度，因此，如果我们能够从汇编角度出发，把比例时间较大的操作消去或转化为比例时间较小的操作组合，那么，我们就达到了对时间常数进行优化的目的。

本文通过如下实例分析进一步说明了竞赛中反汇编在时间常数优化中的应用。

三、实例分析

3.1 关于 memset 函数的小实验

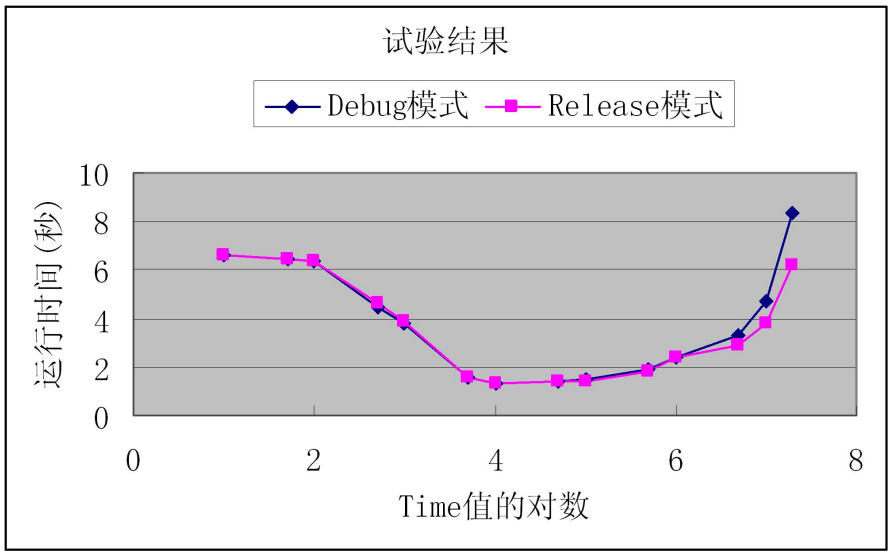
已知 memset 函数为一个 O(N)复杂度的语句。让我们先做一个小实验，观看下面的 C++程序（假设计算机具有足够大的内存）：

```
#include <string.h>
const int Total=1000000000;
const int Time=一个你喜欢的合法的数值;
char field[Total/Time];
int i,j;
int main()
{
```

```
    for (j=0;j<10;j++)
        for (i=0;i<Time;i++)
            memset(field,0,sizeof(field));
    return 0;
}
```

先不忙运行这段程序。你能直接答出 Time 值与运行速度的关系吗？

可能你认为 Time 值对速度无影响，因为程序的时间复杂度守恒。但是，当上机实验后，你会发现，在 Time 值为 100000 左右时程序运行最快，而 Time 值更小或更大时程序运行速度会显著降低，如下图所示：



这是为什么呢？

关于 Time 值较小时程序运行速度变慢的问题，我们可以用 Windows 进行内存分配需要时间这个理论来解释。

当 Time 值变大时，由于循环和 memset 函数在时间常数上存在差异最终导致了运行速度的变慢。

下面从反汇编角度证明 Time 值较大时程序运行速度变慢的理论。

rep stos 的指令速度很快，每次执行大概只占用 1 个单位的时间，而能完成填值和循环变量减 1 的操作，而循环需要使用对变量的赋值操作，还需要进行跳转，大概一次占用 14 个单位的时间，这样由于常数存在差异，所以导致了 Time 值较大时程序运行速度慢。

为什么 Debug 模式和 Release 模式在右端存在效率差异呢？

在 C++语言中，我们把 memset 作为一个函数进行调用。但其实，系统中有专门的汇编伪代码实现相同的工作(rep stos)。例如，在 Release 模式下，编译器对 memset 进行了优化，编译为如下所示的汇编代码：

```
    memset(table,0,sizeof(table));
00401001  xor       eax,eax
00401003  mov       ecx,9C40h
00401008  mov       edi,offset table (4072C8h)
0040100D  rep stos  dword ptr [edi]
```

在 Debug 模式下，虽然 memset 过程的核心也是 rep stos，但由于 memset 是一个函数，系统将使用 push+call+ret 语句组合进行调用，所以实际消耗了更多的时间，最终导致了在 Time 值较大的时候程序运行时间较长。Debug 模式下编译器对 memset 过程处理如下：

```
    memset(field,0,sizeof(field));
00411A6B  push        2710h
00411A70  push        0
00411A72  push        offset field (4284E8h)
00411A77  call        @ILT+350(_memset) (411163h)
00411A7C  add         esp,0Ch
```

在 Debug 模式下，memset 的实现耗费了更多的常数复杂度，因此 Debug 模式时的运行曲线在右端高于 Release 模式。

memset 过程的实现（原代码）参见附录。

上述分析可见，时间常数对程序的速度有一定的影响。通过 memset 不同的实现效率不同这个现象，我们可以发现，对程序进行优化是可能的。

3.2 关于调用时间常数的优化

调用时间常数是指在函数调用过程中 push pop（有的编译器如 VS.NET 的 cl 用 mov 实现）和 call ret 等函数语句在调用过程中的耗费。

调用过程在 Debug 和 Release 模式下有着截然不同的差别。下面分两部分分别叙述：

Debug 模式：

我们常使用 inline 关键字对代码进行优化，但是，inline 关键字对编译器的作用是提示性质的而不是强制性质的（在 Visual Studio 环境下是这样，在竞赛环境(GCC 不带参数)下 inline 关键字为折衷模式：使用即编译为内联函数，不使用则编译为普通函数）。

测试调用的函数原形：`inline void swap(int&a,int&b){int t=a;  a=b; b=t;}`

测试代码：

```
    swap(a,b);
004133AD  lea         eax,[b]
004133B0  push        eax
004133B1  lea         ecx,[a]
004133B4  push        ecx
004133B5  call        swap (41158Ch)
```

```
004133BA  add     esp,8
```

转向经过下表：

```
0041157D  jmp     _aligned_free_dbg (41B4D0h)
00411582  jmp     _RTC_GetErrorFunc (416C30h)
00411587  jmp     GetStringTypeW (425788h)
0041158C  jmp     swap (411B20h)
00411591  int     3
00411592  int     3
```

这样，程序的常数就因为额外的 call+ret 语句而变大了。不仅如此，微软为了方便建立断点进行程序的调试，在 Debug 模式下会在内存中创建跳转表，但是这进一步加大了程序的常数☹。

我们可以稍加想象。比如编译器对 stl 库函数的实现。stl 库函数的 C++代码实现中使用了大量的继承和重载，因此就算是一个简单的操作都要进行很多很多层的迭代（用反汇编模式调试对 stl 库函数的调用时可以很直观的理解这一点）。所以，在我们的程序中，应该针对这个问题进行优化。这里，我提供了两种替代方案：

1、**不使用子函数**：这是最直接的方法。如果没有子函数，当然也就没有了调用的时间开销。但是，这样的程序会产生代码过长和调试纠错困难等问题，所以请酌情使用。

2、**使用宏定义**：宏定义有很多的局限，比如只是机械展开，但是一些简单的，但又经常调用的功能，用宏定义实现就比用函数实现减少了调用操作的消耗。虽然不推荐在 define 块中定义变量（很可能会遇到难以估计的问题），但是用全局变量可以解决同样的问题。如下例就巧妙地解决了 swap 的临时变量问题。

```
int tmp;
#define swap(A,B) tmp=A,A=B,B=tmp
int main()
{
    int a=3,b=4;
    swap(a,b);
    return 0;
}
```

但是，宏定义也有很多不足，比如只是机械展开等，下面的代码可以展示什么叫做**机械展开**：

代码：

```
#include <stdio.h>
#include <stdlib.h>
int getRand()
{
```

```
    int _t=rand();
    printf("Call getRand return %d\n",_t);
    return _t;
}
#define max(A,B) ((A)>(B)?(A):(B))
int main()
{
    srand(543210);
    printf("Get the max value: %d\n",max(getRand(),getRand()));
    return 0;
}
```

运行结果（恒定）：

```
Call getRand return 4461
Call getRand return 14545
Call getRand return 3994
Get the max value: 3994
```

Release 模式：

与 Debug 模式不同的是，在 Release 模式下，任何函数会被优先尝试作为 inline 函数，所以在代码中显式指定 inline 关键字仍然没有实际的作用。

测试 `void swap(int&a,int&b){ int t=a; a=b; b=t;}`

尽管没有 inline 关键字，在反汇编中已经看不到对 swap 的调用了

```
    a++;
0040105B  add     eax,1
    swap(a,b);
0040105E  mov     dword ptr [esp+8],eax
    a*=b;
    printf("%d%d\n",a,i);
    return 0;
00401062  imul    eax,ecx
```

但是这并不是一个沮丧的消息，因为，在 stl 库中，几乎所有的函数都没有带 inline 关键字，尽管这样，这些函数都将被编译为内联函数。这就减少了 call+ret 带来的时间常数。正因为这样，在 Debug 模式和 Release 模式下，stl 库的运行效率才会有巨大的差别。

3.3 除法（求余）的优化

求余运算 **c=a%b** 等效于 **c=a-a/b** 但是，其内部实现直接使用除法的第二个返回值：

```
    a%=b;
00411B53  mov     eax,dword ptr [a]
00411B56  cdq
00411B57  idiv    eax,dword ptr [b]
00411B5A  mov     dword ptr [a],edx
```

所以求余运算的速度和对应的除法相等，优化方法也相似。

按照上面测试的结果，除法指令 idiv 是一种比例时间很大的指令。编译器的设计者也知道这一点。所以大多数情况下编译器都能将常数除法转化为快得多的位运算。（注：编译器同样也会把特定的乘法转化为位运算，比如乘以 2 等）比如，对于 **a/=2**（**a** 为 32 位整数）这句语句在 **Debug** 模式下的解释：

```
    a/=2;
00411B4F  mov     eax,dword ptr [a]
00411B52  cdq
00411B53  sub     eax,edx
00411B55  sar     eax,1
00411B57  mov     dword ptr [a],eax
```

这相对于执行 **idiv** 操作要快很多。但是，乘除法需要额外的特殊情况判断，如正负数、溢出等。这在代码中直接反映为冗余的汇编代码。所以，如果运算直接可用位运算代替，推荐使用位运算。

但是，编译器的智能有很大的局限☹，比如在变量除变量时，编译器根本无法判断变量的特殊性，以至于编译器直接将语句翻译为 **idiv** 操作。这样，如果除数有着特殊性，潜在的性能优化就没有被用到。正确的方法是，判断出特殊性，使用手工的优化方式，如：

原始代码：

```
const a[]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096};
.....
c=b%a[i];
d=e/a[i];
```

优化后的代码：

```
const a[]={1,2,4,8,16,32,64,128,256,512,1024,2048,4096};
.....
c=b&(a[i]-1);
d=e>>(i-1);
```

3.4 关于多维数组的性能优化

数组是高级语言中几乎必然具备的数据结构。数组将多个数据收集到一个变量中。单个索引号（用于一维数组）或者多个索引号（用于嵌套的数组或多维数组）引用数组中的数据。若要引用数组中的单个元素，可以使用后面跟数组索引（数组索引放在方括号 ([]) 中）的数组标识符。若要引用整个数组，则只需使用数组标识符。将数据收集到数组中可简化数据管理。例如，通过使用数组，向函数传递参数时使用一个标识符便能够表示一组数据。

多维数组在内存中是呈平坦展开的，如下图所示：

有如下二维数组

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

在内存中的存储方式如下

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,2

由于在结构上需要进行转换，多维数组的引址操作在 Debug 编译时被翻译成了乘法操作。Release 模式编译时会进行和乘法运算相同的优化。

数组定义：a[10][10]

Debug 模式下，对数组的取值操作使用了 **imul** 运算：

```
    return a[i][j];
00411B6F  mov     eax,dword ptr [i]
00411B75  imul    eax,eax,28h
00411B78  lea     ecx,a[eax]
00411B7F  mov     edx,dword ptr [j]
00411B85  mov     eax,dword ptr [ecx+edx*4]
```

Release 模式下，数组的取值操作被优化为了 lea 以及变址运算，这类似于乘法的优化：

```
    return a[i][j];
0040102E  mov     eax,dword ptr [esp+18h]
00401032  lea     edx,[eax+eax*4]
00401035  mov     eax,dword ptr [esp+14h]
00401039  add     esp,0Ch
0040103C  lea     ecx,[eax+edx*2]
0040103F  mov     eax,dword ptr [esp+ecx*4+10h]
```

由于 imul 是一种比例时间较大的指令，因此如果能在运算中消去这一指令，便能够产生较大幅度的速度的优化。所以，一种很巧妙的优化方式就产生了：如果操作的变址方法固定（比如像宽度优先搜索，变址操作为+1,-1,+N,-N），那么用 type*指针加减操作以及辅助记录要更快一些（省去了乘法操作）。下面是实

验的证实：

数组定义：d[10][10]

使用[]操作取值：

```
    a=d[i][j];
00411B55  mov     eax,dword ptr [i (42B694h)]
00411B5A  imul    eax,eax,28h
00411B5D  mov     ecx,dword ptr [j (42B690h)]
00411B63  mov     edx,dword ptr [eax+ecx*4+42B500h]
00411B6A  mov     dword ptr [a (42B6A0h)],edx
```

使用指针取值：

```
    b=*dd;
00411B8B  mov     eax,dword ptr [dd (42B69Ch)]
00411B90  mov     ecx,dword ptr [eax]
00411B92  mov     dword ptr [b (42B698h)],ecx
```

对指针的滑动操作可以用一个常量数组定义或者用多段代码的方式实现，如下面所示：

定义表和指针：

```
int table[200][200];
int*ptr,*ptr2;
```

定义滑动常数：

```
//East,South,West,North
const go[]={1,200,-1,-200};
```

使用时的代码：

```
    //假设 ptr 已赋值
    ptr2=ptr+go[0];
00411A4C  mov     eax,dword ptr [go (42401Ch)]
00411A51  mov     ecx,dword ptr [ptr (44F5E8h)]
00411A57  lea     edx,[ecx+eax*4]
00411A5A  mov     dword ptr [ptr2 (4284E0h)],edx
    return *ptr2;
00411A60  mov     eax,dword ptr [ptr2 (4284E0h)]
00411A65  mov     eax,dword ptr [eax]
```

这样本来隐藏的乘法操作就被消去了。

这种操作被我称为指针的“行走”操作。使用这个优化有个条件，就是指针变化方式固定。让我们通过一个例子来了解这种优化的作用：

例：adv1900 (NOI2005)

题意描述：

在 N*M 的矩阵中，有一些障碍，有一个物体放在某个格子上。它会按照一

个时间表向某一方向运动，一个单位 1 格。某一秒你可以让它运动，也可以让它静止。问物体最多能运动的长度。

时间表由很多个时间片段构成，在每个时间片断中，物体将向同一方向运动。

数据规模：

50%的数据中，1≤N, M≤200，时间长度(T)≤200；

100%的数据中，1≤N, M≤200，时间片段个数(K)≤200，时间长度(T)≤40000。

这道题有很多的做法，其中最优的做法是使用单调性降维。无论用什么方法，在解决这道题的过程中，都必经一个关键的步骤，这就是在不同的时间点间进行的状态转移。在最优做法中，这一步被“批处理”化了，一次操作就能够完成一个时间段（多个不同的时间点）的转移。同样，其他各式各样的优化，如堆和 RMQ 等，也是基于“批处理”化的思想。但是，在有了上文的实验结论后，我们完全可以另辟蹊径，得到一种很“另类”的解决方法。这基于此步骤具有的使用优化的典型特点：位于循环最里层，直接影响运行速度；大量使用对数组的变化方式固定的操作，可以用指针“行走”来优化。具体实现参考附录。

下面是该方法的速度与优化前的对比：

表3：“行走”优化方法	表4：非“行走”优化方法

选手名称：withWalk

试题：adv1900　文件名：adv1900

编号	评测结果	时间	内存
0	正确	0.016s	520KB
1	正确	0.016s	520KB
2	正确	0.016s	520KB
3	正确	0.047s	520KB
4	正确	0.234s	520KB
5	正确	0.703s	520KB
6	正确	0.547s	520KB
7	正确	0.734s	520KB
8	正确	0.484s	520KB
9	正确	0.797s	520KB

本题总得分100，有效用时3.594s。

选手名称：withoutWalk

试题：adv1900　文件名：adv1900

编号	评测结果	时间	内存
0	正确	0.016s	520KB
1	正确	0.016s	520KB
2	正确	0.016s	520KB
3	正确	0.063s	520KB
4	正确	0.344s	520KB
5	**正确**	**1.016s**	**520KB**
6	正确	0.781s	520KB
7	**正确**	**1.031s**	**520KB**
8	正确	0.625s	520KB
9	**正确**	**1.156s**	**520KB**

本题总得分100，有效用时5.064s。

注：该表以毫秒为单位

虽然和标准方法的速度(最大点 0.3s)有很大差距，但是，使用这种方法，能够让几乎人人都能想出的“初级方法”在时限内通过所有测试！

四、总结

本文主要从 C++语法的特定编译器生成的汇编代码以及各种汇编代码的速度角度探讨了各种操作的时间常数，并通过一些实例从反汇编角度分析了影响时间常数的原因，提出了优化改进方案。

汇编语言，是最接近于计算机本质的程序设计语言。由于其特殊的性质，它也是最难掌握和最难调试的语言之一。但是，在很好的掌握和使用后，其高速、高效、简洁的优美性质也是一把削铁如泥的利刃。俗话说，万变不离其宗，高级语言究其本质也是用汇编语言实现的。所以，在对效率日益要求严格的今天，了解语言的“习性”是至关重要的。

我相信，在了解了汇编语言的特性后，无论是一般的编程，还是在竞赛中，充分应用汇编语言的特性，一定能够使运行速度再上一个台阶，从而更快更好的

解决实际问题。

致谢：首先我要感谢张君亮老师多年来对我的悉心指导，为我的成长倾注了大量的心血；感谢电子科技大学的陈文宇老师对我论文的修改和建议；感谢班主任张建成老师对我的关心和支持；感谢闵可锐、白云峰、杨双、王凯同学与我的许多有益的讨论。

【参考文献】

[1] 《The art of assembly language》
[2] MSDN China (Visual C++ part)
[3] National Olympaid in Informatics(NOI) 2005

附录

名称	文件
测试库 (近似语句速度判定) 另外一个测试由类似过程生成	 assembleTime.rar
Visual Studio 中的 memset 函数汇编实现	 Memset.asm
Adv1900 指针行走和未优化程序	 Walk.rar