论程序底层优化的一些方法与技巧

成都七中 骆可强

摘要：本文以优化程序运行的时间效率为目地，从编译器、汇编代码、CPU 特性等较为底层的概念着眼，对程序优化进行了全方位的探讨，总结了在优化中实用的思想、原则、方法和技巧，并对它们在竞赛中的应用价值做出了一些尝试。

关键字：优化 CPU 汇编语言 编译器

目录

序言

信息学奥林匹克竞赛（Olympiad in Informatics）是研究怎样编写计算机程序来解决特定问题的竞赛。考察的关键点，在于怎样利用有限的系统资源（CPU 时间片与系统内存）来求解规模庞大的数学模型。在“正确”这一前提下，“效率”自然是考虑问题的第一要素。

效率，分为时间效率与空间效率，如何对时间效率进行优化是本文将要研究的主题。

算法是决定时间效率的关键

优化程序的时间效率，简单地讲，就是用尽一切手段，在保证正确的前提下让程序的运行时间更短。那么，有些什么手段呢？最重要的自然是：使用尽可能高效的算法。

算法（Algorithm），是一系列解决问题的机械步骤，它采用明确定义的语义，描述了求解特定数学模型的一般方法。算法的好坏，直接决定了程序的运行效率。采用低效的算法或高效的算法，其差别就好像选择走路或是坐飞机，完全不在一个数量级。

时间复杂度的概念及其局限性

为了衡量一个算法时间效率上的优劣，计算机科学中引入了时间复杂度的概念。回忆我们习惯使用的大 O 表示法，我们说一个算法运行时间的界是 O(f(n))，所表示的意义是，假设这个算法的实际运行时间关于输入规模 n 的函数是 T(n)，那么存在正常数 n0、c，使得对于 n≥ n0，有 0≤T(n)≤c×f(n)。有了这样一个上界，我们就能知道 T(n)的增长速度，从而能够大致判断对于给定的 n，我们的算法能不能在一个合理的时间内出解。

然而另一方面不可否认的是，这样一个工具是极其粗略的。我们注意到刚才的定义式中存在一个常数 c，它在渐进意义上是无关紧要的，但回到现实世界中，它却不可忽视。同样是 O(n)阶的算法，有些可以快到只消耗 2×n 个 CPU 时钟周期，而另一些甚至需要 1000×n 个时钟周期还要多。就好像同样是乘坐飞机，也有快慢的极大差别。使用相同时间复杂度的算法，因为这个常数 c 的不同，实际运行所需要的时间，也可能有天壤之别。

算法并不是时间效率的全部

那么，这个常数受哪些因素的影响呢？无疑，它同样受制于算法：不同的算法，可能有着相同的复杂度，但是实际效果截然不同。相同的算法，可能有着不同的实现方式，一些逻辑上的简化也能大大降低运行所需花费的时间。那么，算法就是程序运行效率的全部了么？答案是否定的，有些东西是隐藏在逻辑层面之下的，它们同样显著地影响着程序的运行效率，而我们却很难看到。举例来说，你能想象当我们在 C 语言中书写 a/=7 这一语句时，实际上处理器并没有做缓慢的除法，而使用了乘法和位移取而代之么？因为这样，我喜欢把大 O 定义式中的常数 c 一分为二的来看待：c＝c1×c2，c1 代表逻辑层面（算法）的消耗，而 c2 表示每一句程序语句在底层运行的消耗，那么程序的实际运行时间，约为 c2×[c1×f(n)]，方括号中的部分 c1×f(n)就完全由算法来决定，而 c2 则取决于程序的底层实现。前者固然重要，后者也同样不可忽视。本文将要研究的，就是怎样在程序运行的底层对细节做出优化，以提升程序的运行效率。

为什么要学习底层优化

在 OI 竞赛中，算法是考察的重点。底层实现看起来并不重要，确实提升空间也相对较小，但当我们设计的算法有一些先天缺陷时，或许对底层做细致的优化能对我们有很大的帮助，在后文中会展示一些实际的例子。

在竞赛之外，学习底层的东西，能让我们更深入地认识眼前的机器，即使在使用高级语言书写程序时，脑海中也会自然投射出底层发生的事情，从而能够写出质量更高的代码。

在这篇论文前期准备、实验研究、总结规律到最终成文的过程中，我学到了太多的东西，这些东西在我们平时为 OI 竞赛编程的过程中是很难看到的，而我也相信，这些东西在为大家所广泛认识之后，同样能够服务于竞赛，实际地提升成绩。

说了这么多，或许是该展现一个实例的时候了。下面我们从一个极其简化的编程任务入手，来看看什么是底层优化，有些什么样的工具，能做到什么程度。

平台简介

条件所限，本文中的所有研究、编码、测量工作均在一台 Thinkpad X61 笔记本上进行，下面简要给出工作平台上硬件和软件与优化工作相关的规格参数：

CPU:	Intel Core 2 Duo CPU T7100 @ 1.80GHz
Cache:	L1 - 64KB L2 - 2MB
主存:	SODIMM Synchronous 667 MHz 1GB × 2
操作系统:	Ubuntu 8.10 - Intrepid Ibex
内核版本:	Linux 2.6.27-8-generic
编译器版本:	gcc 4.3.2
汇编器版本:	GNU assembler 2.18.93.20081009
连接器版本:	GNU ld 2.18.93.20081009

当然，一种优化方法的实际效果，和平台规格密切相关。如果只在一种平台（主要指 CPU）进行测试就妄下结论，是不负责任的。为此，我也将文中所涉及的程序在不同的平台下进行了广泛的测试，跨越了不同的操作系统（windows 与 linux）和不同厂家生产的 CPU（intel 与 AMD）。测量结果的绝对值自然千差万别，但采取不同优化方式所取得的效果基本上是一致的，这得益于现代 CPU 都采用了一套相似的内部架构与优化引擎，为开发人员提供了方便。不过另一方面，也要求我们在学习 CPU 开发时应以把握大原则为重点，而不要钻牛角尖。如果沉溺于为某种特定的 CPU 内部特性做古怪的优化，会导致在不同的 CPU 上运行效果天差地别，这样的优化是没有价值的。遗憾的是，后文将提及的 SIMD 系列优化方法，在 AMD 生产的芯片上，虽然可以兼容，但是优化效果并不突出，值得大家留意。

引例

让我们先来看一个非常简单的例子：

假设我们被要求编写这样一个函数，接受一个指向 32 位正整数数组头部的指针，以及数组的长度，要求函数返回这些整数中的最大值。在 C 语言中，函数头应该是这样：

```
int get_max(int* a,int l);
```

朴素的实现

若要在算法层面研究这个问题几乎没有任何价值，进行一遍线性的扫描即可得到 O(n) 级别的算法，另外由于数组中的每一个元素必须被访问到才能保证出解正确，所以算法的复杂度下界也是 O(n)。我们可以立即写出下面这个 C 语言程序：

```
//最初版程序
int get_max(int* a,int l){
    int mx=0,i;
    for(i=0;i<l;i++){
        if(a[i]>mx)mx=a[i];
    }
    return mx;
}
```

编译说明

众所周知，现代编译器已经不再局限于简单地把一句句高级语言翻译为对应的汇编语言，而能够智能地完成许多以前只有人手工才能完成的优化，特别是各个编译器提供的高级优化选项，更是常常能够提供接近人类手工优化的效率。

但是，在我们 OI 竞赛的评测中，这些优化选项是不会被开启的。所以，本文中所有程序的编译，都没有打开优化选项。诚然，打开这些优化选项，本文中介绍的各种人工优化的效果可能会打些折扣，但我仍然觉得这样做是合理的，一来更贴近比赛中使用的评测环境，二来既然是在研究优化，我们自然应该亲自去探究其中的方法和原理，而不应该让别人的程序来代劳，否则可能永远也不会清楚其中的奥妙。

为此，本文中所有的程序均使用竞赛时的编译选项进行编译并测量效果，对于 c 语言程序，编译命令为:

```
gcc a.c -o a
```

这个程序显然能够正确地完成所要求的任务，而且，算法的复杂度也达到了理论下界。

效率怎么样呢？

让我们来检验一下程序运行的速度。我在测试平台将此函数应用于一个存有 10000000 个整数的数组，连续运行 100 次并求平均占用的时钟周期数。测量值为 75305510 个周期。处理一个数平均就占用了 7~8 个时钟周期，结果不令人满意。

关于文中程序所使用的计时方法

为了测试程序的运行效率，需要一种测量时间的工具，有许多不同的库函数，操作系统 api，命令行工具可以进行时间测量，本文主要使用 IA32 处理器提供的 rdtsc 指令来获取程序运行消耗的时钟周期，此方法精确度较高，缺点在于无法排除掉程序运行过程中操作系统和后台运行程序所占用的时钟周期，结合 linux 系统命令 time 一起使用可以弥补这一缺憾.

下面是本文所有程序都会用到的计时函数的 C 代码：

```
#define ull unsigned long long
ull get_clock(){
    ull ret;
    __asm__ __volatile__ ("rdtsc\n\t":"=A"(ret):);
    return ret;
}
```

第一次优化

开始着手寻找程序可优化之处，首先发现'a[i]'被提及了两次，重复计算地址在这个小循环中的开销也是不可忽略的，那么第一版的优化我们尝试来进行普通的循环和寻址的优化：

```
//第一次优化
int get_max(int* a,int l){
    int mx=0,*ed=a+l;
    while(a!=ed){
        if(*a>mx)mx=*a;
        a++;
    }
    return mx;
}
```

测试一下运行时间，平均占用时钟周期测量值为:66047005，处理一个数平均下降了一个时钟周期，比第一版程序在时间效率上优化了 12%。

来回顾一下这次优化，他解决了在初版程序代码中出现的重复寻址的问题，而获得了期望中效率的提升，从本质上说仍然属于逻辑层面的优化。这次优化是在 C 语言层面编写，也能够在 C 语言层面进行解释的。

第二次优化

继续前面的思路，很难再想出什么有效的优化了，这就是我们的终点吗？

其实游戏才刚刚开始，下面给出第二个优化：

```
//第二次优化
int get_max(int* a,int l){
    assert(l%8==0);
    #define D(x) mx##x=0
    int D(0),D(1),D(2),D(3),
        D(4),D(5),D(6),D(7),*ed=a+l;
    #define CMP(x) if(*(a+x)>mx##x)mx##x=*(a+x);
    while(a!=ed){
        CMP(0);CMP(1);
        CMP(2);CMP(3);
        CMP(4);CMP(5);
        CMP(6);CMP(7);
        a+=8;
    }
    #define CC(x1,x2) if(mx##x1>mx##x2)mx##x2=mx##x1;
    CC(1,0);CC(3,2);
    CC(5,4);CC(7,6);
    CC(2,0);CC(6,4);
    CC(4,0);
    return mx0;
}
```

这个程序使用了许多宏定义，可读性较差。不过思路非常简单：将原来单线的求最大值进程分为八路，最后再来汇总总的最大值。要说明一点的是，正如程序第一行的 assert 所显示，这个函数只能在 l 是 8 的倍数的情况下工作，要让它可以对任何 l 工作很容易，不

过那不是我们关注的重点，为了让代码简短一些，略去不表。

这个程序的效率怎样呢？同样平台下的测量值为：34818706，比最初版本优化了 54%。

是什么让这个程序拥有了如此大的效率提升？难道是指针 a 的递增次数只有原来 1/8 导致的？从 C 语言的层面来看，只有这一个解释。但这是说不通的，因为相比循环体里面的语句，这一个递增显得无足轻重，至少不会带来这么大的效率改善。而除了这里，从 C 代码上看和第二个程序又没有本质上的区别。难道计算机程序的运行是完全不可预测和理解的吗？

深入汇编语言

让我们来回顾一下 C 语言编写的程序是怎样运行的吧，首先编译器(compiler)将 C 代码编译为汇编语言(assembly language)代码，再经过汇编，连接等一系列步骤转化为可执行文件。这里最关键的一部，自然在编译环节，因为一旦代码编译为机器指令后，在 CPU 中执行它的方式就已经确定了。编译的质量，也直接决定了程序运行的效率。如果仅仅把目光集中在 C 代码的层次，而完全不在汇编语言层面进行思考，优化的过程就像被蒙住了眼睛。既不可能真正看清程序运行效率的本质，也不能进一步进行更强的优化。

对第二次优化的程序进行编译，得到相应的汇编代码，经过查看，发现除了循环变量的累加之外，其他语句对应的汇编代码的面目并无甚差别，仅仅是简单的重复并列起来。但确实获得了极大的效率提升，要完整的解释这个问题，会提及处理器中的各种优化机制：乱序执行、流水线机制、指令预取、分支预测、寄存器重命名、高速缓存，这些主题都将在后文中作研究。简单的讲，在最初两个程序中，每次计算新的 mx 都会依赖于上一步的计算结果，相关的计算指令也必须依次运行，而将求值过程分为多路处理，mx0,mx1 等变量的相关指令之间互相没有关联，让处理器有更大的机会将他们并发。过于底层的细节固然不容易完全掌控，但是遵循一些基本的原则，总有机会使处理器为我们作出优化。

第三次优化

既然已经深入到了汇编语言的层次，那不如直接用汇编语言来编写这个函数。

汇编语言的利与弊

在 CPU 中重要的概念如寄存器(register)、状态标志(status flag)、指令(instruction)等，在高级语言中全部被隐藏，取而代之的是，高级语言过于依赖内存变量这一概念，而读写内存，是处理器最低效的操作之一。而且高级语言过于丰富的语义也造成了翻译过程自然出现极大浪费。这一切，让我们在 C 语言层面作优化，就像带着脚镣跳舞，一旦使用起汇编语言，这一切就豁然开朗了，我们不再被内存变量与高级语句所束缚，可以直接操纵最底层的部件，更有额外的丰富指令可供使用。可以说，使用汇编语言是获得极致效率的基础。

汇编语言既然如此之好，那我们为什么不用汇编语言来编写所有的程序呢。这主要是因为汇编语言编码成本太高，要用来编写大型程序虽并非完全不可能，也是极其困难的，实际的程序中并非所有地方都需要如此极致的效率，而真正需要的是正确地设计代码架构并编写程序逻辑，要用汇编语言来做这些，就好像用小颗粒的颜料逐点绘制一副油画，是很难把握成品的全貌的。另一方面，汇编语言编写的程序会有较强的平台依赖性，可移植性很差，也使它被拒于应用程序开发的门外，而高级语言(如 C 语言)被发明出来较好地解决了这些缺陷。

那么汇编语言又一钱不值了吗？当然也不是，就算在现今的实际程序开发过程中，程序员们仍然使用汇编语言来编写程序中的效率瓶颈部分，而程序的主要部分使用高级语言来

编写。要实现这一点，最简便的方法是使用编译器各自提供的内嵌汇编机制。使我们可以直接在 C 代码中书写汇编代码，并且可以通过 C 编译器的编译。这样做还有一个最直接的好处：在竞赛中，我们可以直接提交这样的程序并通过编译。

声明

本文并非以讲授汇编语言与计算机体系结构为目的，所以在这里假设读者对这两者有最基本的了解，否则阅读会有较大障碍。

使用内嵌汇编进行优化

下面正式开始转向汇编语言进行优化，本文中的程序均使用 gcc 内嵌汇编，采用 AT&T 格式的汇编语法，后面不再做说明。

首先，我们直接朴素地将原始意图使用汇编语言实现一遍，得到如下代码：

```
//第三次优化
int get_max(int* a,int l){
    int ret;
    __asm__ __volatile__ (
        "movl    $0,  %%eax\n\t"
        ".p2align 4,,15\n"
        "LP1:\n\t"
        "cmpl    -4(%1,%2,4),  %%eax\n\t"
        "jge ED\n\t"
        "movl    -4(%1,%2,4),  %%eax\n"
        "ED:\n\t"
        //"loop    LP1\n\t"
        "decl    %2\n\t"
        "jnz LP1\n\t"
        "movl    %%eax,  %0\n\t"
        :"=m"(ret)
        :"r"(a),"c"(l)
        :"%eax");
    return ret;
}
```

这个程序的效率如何呢？测量结果为:平均 21322853 个时钟周期。处理一个数据平均只需要 2 个时钟周期了，相比最初的程序，优化了 72%，结果十分令人满意。

打量一下这个程序，核心循环中，有 5 条指令，其中甚至有两条是条件分支指令，还有两条需要访问内存，而且使用了最复杂的 sib 寻址方式。感觉起来，平均 2 个时钟周期，是没有道理的，其实这主要得益于现代 CPU 各种强大的优化机制：高速数据 cache 使两次访问同一内存如同访问寄存器一般迅速，第一个条件跳转大部分时间不会成立，而相反第二个跳转总会成立，这让 CPU 的分支预测发挥到极致。而强大的乱序执行引擎使得循环中的这些小指令得以以接近双倍的时间运行（以上提到的这些名词在后文都会有详细的介绍）。现代 CPU 的优化如此强大，是否我们可以胡乱书写汇编代码了呢？绝对不是。

优化中的一些小插曲

注意到在上面的程序中，有一行注释掉的 loop 指令，如果没有经验，或许会想，使用 loop 指令，指令码体积更小，而且使用复杂指令，一次性指示 CPU 完成更复杂的工作，应该具有更高的效率。那么我们使用 loop 指令取代递减与跳转指令试一试，结果令人大跌眼镜：平均耗费 56457348 个时钟周期，整整慢了一倍还多。

RISC 与 CISC

要解释这个问题，不得不提到 RISC 与 CISC 架构。CISC 全称为“Complex Instruction Set Computer”，即“复杂指令系统计算机”，它从计算机诞生之初一直沿用至今，他的指令集极其庞大，功能繁杂，导致制作工艺复杂，成本高昂，而且速度缓慢。看看我们现在使用的 intel pentium 处理器就知道了，虽然看似其中有很多一次性完成复杂工作的指令，事实上还是在 CPU 内部被翻译成多条微指令，才能真正在 cpu 上运行。这个过程本身造成的性能损耗可想而知，虽然处理器厂商采用了各种强大的办法来试图优化这一过程，也不能弥补设计上先天的劣势。后来工程师们发现，事实上人们所使用的 80%的指令都处于 20%的指令集中，于是设计了理念完全不同的 RISC(精简指令集计算机)。通过采用一个较小但功能完备的指令集，大大简化处理器的设计。RISC 中不再需要微指令的概念，而直接硬件执行指令码，在一个时钟周期执行一条指令，性能极高且容易控制。到今天，只有 intel,AMD 等少数厂家还在生产 CISC 芯片。

虽然 RISC 有如此多的优势，但不可回避的是，我们的普通桌面应用已经使用 CISC 架构许多年，对以前软件的兼容性不能放弃。为了兼顾性能与兼容性，处理器厂商使用了折中的方案：处理器的外层继续兼容老式的 x86 指令集，而内核尽量向 RISC 靠拢：使用一套类似 RISC 的微指令集，内部采用 128 个物理寄存器，在外层通过微指令解码器，RAT(Register Alias Table)等把对外部指令映射到 RISC 核心上去。除此之外，处理器厂商继续大力优化指令集中类 RISC 的那一部分，使他们在流水线上能有最好的表现，而对于某些复杂指令，在很早以前就已经被处理器厂商放弃，cmps、cmov、lea 等复合指令不再具备速度优势，loop 当然也是其中一员。所以，我们应该了解现今做汇编优化，应该尽量使用最基本的那一个指令子集，CPU 会帮助我们尽量高效地运行。当然也有例外，那就是各 cpu 厂商都投入大量精力研发的 SMID 指令集，这在后面会提到。

回到我们的程序，loop 指令低效的问题已经有了解答。同理，如果试图用 cmov 指令优化掉条件跳转，结果同样会令人失望。

第四次优化

完成了这个朴素的汇编优化，下一步又怎么做呢？容易想到：降低程序上下文依赖性！在前面的第二次优化中，这种方法起到了非常好的效果，那我们就在汇编语言中如法炮制一次：

```
//第四次优化
int get_max(int* a,int l){
    assert(l%2==0);
    int ret;
    __asm__ __volatile__ (
        "movl    $0,   %%eax\n\t"
        "movl    $0,   %%edx\n\t"
        ".p2align 4,,15\n"
        "LP2:\n\t"
        "cmpl    (%1),     %%eax\n\t"
        "jge ED2\n\t"
        "movl    (%1),     %%eax\n"
        "ED2:\n\t"
        "cmpl    4(%1),    %%edx\n\t"
        "jge ED3\n\t"
        "movl    4(%1),    %%edx\n"
        "ED3:\n\t"
        "addl    $8,   %1\n\t"
        "subl    $2,   %2\n\t"
        "jnz LP2\n\t"
        "cmpl    %%edx,  %%eax\n\t"
        "cmovll  %%edx,  %%eax\n\t"
        "movl    %%eax,  %0\n\t"
        :"=m"(ret)
        :"r"(a),"r"(l)
        :"%eax","%edx");
    return ret;
}
```

我们期待着它能起到第二个优化一般神奇的效果，运行程序，测量结果为 17447544 个时钟周期。确实取得了优化，但效果不明显。立即想到，是不是应该展开更多的次数呢？我试着进行 4 次展开，用上了全部的通用寄存器，结果再次令人大跌眼镜：速度竟然比没有展开时还要慢。

其实仔细观察这个程序和前面第二次优化程序的汇编代码，不难发现第二次优化由于是编译器生成代码，冗余的小操作很多，乱序执行有非常大的优化空间。但是到了这个程序，代码已经十分精简，在每次循环体第一句 mov 指令 cache miss(后文会讲解)时，后面并没有指令可以提前来执行。所以优化的这点时间，本质上仅仅是循环展开所得。如果再进一步试图采用所有寄存器参与来进行 4 路求值，一来很不利于 RAT 工作，另一方面过多的条件跳转指令也让处理器吃不消。我们再一次看到，优化工作不是想当然的，如果不充分了解处理器的特性，仅仅凭想象来做优化，不会取得什么效果，甚至适得其反。

第五次优化

已经做到了这个程度，考察程序各处似乎都无利可图了，优化再次陷入了僵局。要想再取得优化，必须再打开思维才行。这里要提到的，是所谓的单指令多数据(SIMD)的方法。

什么是 SIMD?

让我们回归程序优化的本原：优化，自然是为了让程序运行效率更高，而程序运行时，%90 的时间都在运行 10%的代码，这些代码自然就是循环。我们只用找出程序的瓶颈部分，在最耗时的循环中取得微弱的优化，也胜过在外层决定性的巨大优化。而这些循环是在做什么呢，许多时候，他们是在遍历并处理大块的数据。处理器厂商发现了这一点，开发了 SIMD 指令集来专门帮助更高效地处理大量数据。

我们刚才有提到，不应该使用过于复杂的指令，用经过充分优化的更适应流水线的简单指令对程序运行效率更有益处，SIMD 的理念看上去不是和这个原则背道而驰么？其实不然，打个比方，我们有三条基本指令，洗苹果，削苹果，切苹果，为了更高效地完成任务，我们把他整合成一条复杂指令：处理苹果。这对更高效地完成任务有帮助吗？一点也没有，因为接受命令者首先要先回忆，处理苹果这一条指令意味着什么，然后再按部就班地进行洗、削、切三个过程，不但执行任务的时间没有减少，反而还多出了分解复杂命令的过程，如果说这一个整合有帮助的话，那就在于创建了一条更简短的指令表达丰富的含义。使用那些“单指令多任务”的复杂语句，就好比上面一个过程，对时间效率优化毫无益处。但是，“单指令多数据”就不一样了，再设想一下，假如我们时常需要处理一大堆苹果，我们可以创建一个叫“一刀切两个苹果”的指令，那么在切的阶段，必然就能更高效地完成任务，为什么不创建“一刀切 100 个苹果”的指令呢？因为我们的刀，让我们一次切不了这么多。处理器就是我们的刀，数据就是苹果，创建一次处理多个数据的指令，可以减少发号施令所用的时间，而让处理器专注于十分高效地完成数据处理任务。同样，处理器处理数据的能力也是有限的，一次能处理多少数据，受制于处理器内部寄存器的大小。intel 在 Pentium MMX 处理器中发布了 MMX 指令集，它借用 FPU 中的八个 80 位浮点寄存器，一次只能同时处理 8 字节数据。在 Pentium 2 又发布了 SSE 指令集，加入 8 个独立的 128 位寄存器，此后又发布了 SSE2,3,4，不断加入新的 SIMD 指令，使程序员可以极其高效地完成打包整数、浮点数的处理，在科学计算、信号处理、3D 游戏中都有着广泛的应用，而 AMD 也一直紧跟 intel 的脚步，现在新一代的 AMD 处理器均能支持到 SSE3 指令集。下面我们就来尝试将 SIMD 用在这个程序的优化中：

```
//第五次优化
int get_max(int* a,int l){
    assert(l%4==0);
    assert(sse2);
    int ret,tmp[4];
    __asm__ __volatile__ (
        "\txorps   %%xmm0,    %%xmm0\n"
        "LP3:\n"
        "\tmovdqa    %%xmm0,    %%xmm1\n"
        "\tpcmpgtd   (%1),     %%xmm1\n"
        "\tandps   %%xmm1,    %%xmm0\n"
        "\tandnps (%1),     %%xmm1\n"
        "\torps      %%xmm1,    %%xmm0\n"
        "\taddl    $16, %1\n"
        "\tsubl    $4,   %2\n"
        "\tjnz     LP3\n"
        "\tmovdqu    %%xmm0,    (%3)\n"
        "\tmovl    (%3),     %%eax\n"
        "\tcmpl    4(%3),    %%eax\n"
        "\tcmovll 4(%3),    %%eax\n"
        "\tcmpl    8(%3),    %%eax\n"
        "\tcmovll 8(%3),    %%eax\n"
        "\tcmpl    12(%3),   %%eax\n"
        "\tcmovll 12(%3),   %%eax\n"
        "\tmovl    %%eax,   %0\n"
        :"=m"(ret)
        :"r"(a),"r"(l),"r"(tmp)
        :"%eax");
    return ret;
}
```

用 SSE 指令集来处理这个问题的方式有些间接，大体思路仍然是多路求值，不过这次的这四路局部最大值，打包保存在 xmm0 寄存器中。为了把内存中的四个连续值与 xmm0 取较大值，需要先使用 sse 比较指令获得一个掩码，再通过位运算进行结合，这也是 sse 处理数据的常见方式。值得注意的是，这里使用了整整 5 条 sse 指令，这么复杂的过程，真的能有优化的效果么？运行程序，这次的平均时钟周期为 15898751，还真有一定的优化效果。看来，处理器厂商确实也在 simd 指令优化上下了大力气。

到了这里，我们的程序仅仅需要最初版程序 1/5 的时间了，处理一个数据仅仅需要 1.5 个时钟周期，效率是十分惊人的。

第六次优化

优化就到此为止了吗？当然没有，有句话说得好：程序的优化是无止境的。我相信一定还存在更加优秀的方式解决这个问题，只是我才疏学浅，暂时还无法企及。不过另一方面，cpu 厂商同时也在不断地努力，给我们提供更加强大的工具。就这个问题，在 intel 近年发

布的 45 纳米 Penryn 处理器中，支持了 SSE4 指令集，其中有一条 pmaxsd 指令，可直接取代前面的位运算步骤，那么可以写出这样一个程序：

```
//第六次优化
int get_max(int* a,int l){
    assert(l%4==0);
    assert(sse4);
    int ret,tmp[4];
    __asm__ __volatile__ (
        "\txorps   %%xmm0,     %%xmm0\n"
        "LP4:\n"
        "\tpmaxsd      (%1),     %%xmm0\n"
        "\taddl     $16, %1\n"
        "\tsubl     $4,   %2\n"
        "\tjnz      LP4\n"
        "\tmovdqu      %%xmm0,     (%3)\n"
        "\tmovl    (%3),     %%eax\n"
        "\tcmpl    4(%3),    %%eax\n"
        "\tcmovll 4(%3),    %%eax\n"
        "\tcmpl    8(%3),    %%eax\n"
        "\tcmovll 8(%3),    %%eax\n"
        "\tcmpl    12(%3),   %%eax\n"
        "\tcmovll 12(%3),   %%eax\n"
        "\tmovl    %%eax,   %0\n"
        :"=m"(ret)
        :"r"(a),"r"(l),"r"(tmp)
        :"%eax");
    return ret;
}
```

这个程序在我的处理器上尚无法运行，故无法测量其优化效果，不过我估计，在未来支持 SSE4 的处理器上，这个程序可以跑到处理每个数据一个时钟周期的极致。

这个程序的优化就做到这里了，确实，这个例子显得过于简单而没有实用性，但它所折射出来的一些思想和方法是共通的，后文将分别具体讨论各种方法与技巧，它们将为我们的优化实战提供详细的参考。

关于各 CPU 指令的运行效率

在后文的内容中我们时常要与 CPU 指令打交道，要做出有效的优化，对这些指令的效率有基本的了解是前提。因此在开始之前，我们先来研究一下这些指令运行的快慢。

在 CISC 架构处理器上，一条特定的指令具体要占用多少个时钟周期是很难有一个确切的答案的。因为这条指令不是独立存在于此的，在实际的运行中，它的微指令可能会被打散并和其它指令并发执行，这样就不再具有一个完整且独立的运行周期。另一方面，指令本

身和指令所使用的数据是否 cache，是否对齐，对效率也有着致命的影响。总的来说，一条指令的运行效率，和代码中的上下文环境有着密不可分的联系。这当然不是说我们就不需要关心各种指令的快慢了，例如一条除法指令无论在什么情况下也是会比一条加法指令要慢的，对指令集中各条指令的效率有个大致的了解，是正确地书写汇编代码的基础，甚至对于编写高级语言代码也很有好处。

诚然 intel 官方的开发手册上有提供各条指令平均占用时钟周期数的一个参考数值，不过我还是更喜欢在自己的平台上进行实际测量，印象也会更深刻一些。

我的测量方式是，将待测指令放在一个有意义的上下文环境中，在指令的两测各插入一条 cpuid 指令以将被测指令隔离开来，并循环 1000000 次，测量消耗的时钟周期，再将被测指令抽去，再次循环同样的次数并测量，两者相减并除以循环次数，就得到了一个近似的时钟周期数。当然，这样的测量有许多可能导致误差之处，不过没有关系，因为我们关注的重点并不是它实际需要占用时钟周期数的绝对数值，这个数值在不同的 cpu 上是有差异的，但了解一个大体的情况始终是有意义的。下面给出我所测量的一些结果，它们将会在后文的论证中被引用。

指令名称	操作数类型	平均时钟周期数
add	r32/r32	1
shr	imm8/r32	1
bsr	r32/r32	1
mul	r32	2
div	r32	21
fadd	r	3
fdiv	r	35
fsqrt	n/a	60
padd	xmm/xmm	1

关于数值运算的优化

一．除法

在现代 cpu 中，乘法指令的速度基本已经被优化到足够快了，但除法指令，由于其逻辑的过于复杂，一直以来都需要消耗庞大数目的时钟周期，比其它简单指令要慢上数十倍。从前文中所做的测量就可以看出，即使使用 20 条位移指令，如果能取代一条除法指令的话，也是十分值得的。

可是只要稍加思考就会想到，假如存在一种通用的除法算法，其效率能够比 CPU 内部硬件实现的除法指令还要高的话，CPU 厂商一定会使用这种算法来重写除法电路。也就是说，对于一次普遍性的除法任务，固然 div 指令较为缓慢，但是也没有什么可优化的余地。从逻辑上讲是这样，事实也就是这样。

消除除法

当然，这并不意味着对于除法我们就束手无策了。在各种特殊的场合，时常有专门的方法可以进行一些针对性的优化。最简单有效的一个方法无疑就是：避免使用除法，这时常需要在逻辑上进行一些调整，例如最简单的一句条件判断语句：if(a/b<c)，我们可以将其转换为 if(a<b*c)，要注意这并不总是成立，要考虑到乘法可能溢出还有 b 为负数时不等式需要变号等问题。在实际的算法设计中，有时也有机会进行一些类似的变换，不过需要非常小心。

减少取模运算

在竞赛题目中，我们时常需要解决这样的问题：要求对某种对象进行计数，不过由于数目太大，仅仅需要我们给出最后答案 mod M 的值即可。在这类问题的算法设计中，滥用%M 操作对效率的影响可能是致命的，而取模操作，在计算机底层和除法操作是同一条指令所完成，速度极其缓慢，应该尽可能消除。这个工作有时只是举手之劳，例如对于一个可能为负的整数 a 进行取模，惯常的写法是(a%M+M)%M，之所以有两次 mod，是为了解决底层除法指令规定余数的符号和被除数相同的问题，但是第二次 mod 是相当浪费的，一个简单的 if 就可以解决这个问题：

```
inline int mod(int a){a%=M;if(a<0)a+=M;return a;};
```

又比如说，在许多线性递推的模型中，将其抽象为矩阵乘积的形式可以高效地解决。在求矩阵乘积的内层循环中，频繁地调用除法指令会使程序运行效率不堪忍受。为了解决这个问题，可以在计算结果中某个元素的值时先不进行取模，仅仅简单地进行乘法和累加，在计算完后再来取模。实践表明，此方法可以极大的提高程序的运行效率。

减少浮点除法

如果把浮点数也加入进来考虑，那优化的方法就更加灵活多样。一方面浮点数的各种运算之间的速度差距更为悬殊，除法指令更是慢到难以忍受；而另一方面，浮点数模拟的是数学中实数的概念，运算规律为我们所熟悉，不需要像对待整数运算一样去推导数论方面的结论。例如，在浮点数中，我们可以认为 a÷b 等价于 a×(1÷b)，而在整数运算中是不行的。相比在整数领域，对于浮点我们可以更加不遗余力地将除法指令替换为多条更加快速的指令：最简单的例子是，如果需要多次进行除以 b 的操作，我们事先计算 b 的倒数再来进行乘法是很划算的。如果要计算 a÷b＋c÷d，我们可以先通分变为(a×d＋c×b)÷(b×d)再进行计算，虽然多了 3 次乘法，但是消除了一次除法，仍然是值得的。对此我进行了实际的测量，用通分与不通分的方式分别计算 10000000 次，在我的机器上，通分的方法，整整快了一倍还要多。这种优化甚至可以扩展到不相关的代数式上，例如我们需要计算：

```
x=a/b;
y=c/d;
z=e/f;
```

我们可以写为：

```
t=1/(b*d*f);
x=a*d*f*t;
y=c*b*f*t;
z=e*b*d*t;
```

在我的机器上，第二段代码占用的时间，仅仅是第一段的 1/3！至此我们已经能够看出，浮点数乘法指令在除法的面前，几乎是可以忽略不计的。其实不仅仅是乘除法，浮点指令集中不同指令的效率差距拉得很开，这给了我们的优化无限的可能。就曾经有人自己实现了一个非常高效的基于牛顿迭代的开根算法，比 FPU 内部硬件实现的还整整快了 6 倍。不过，这些技巧更像是某种智力游戏，就不再在这里展开讨论了，我们只需要对各种浮点指令的速度有一个大致的了解，在编码时再即兴发挥即可。不过凡事不能过度，如果需要使用过多的简单指令来取代复杂指令，不但效率是否能有优化值得商榷，还会显著地导致代码易错，难

于调试和维护，在精度方面的损失，也不可忽略。

特殊除法运算的加速

上面讨论的技巧，主要的思想都是通过避开除法或者减少除法指令的数量来进行优化。那当我们必须直面一条除法指令时，有没有办法让它可以运行得更快呢？对于通用的目的，可以说是没有。但我们程序中对除法的使用，常常有某种特殊性，最简单而且有效的一个例子是：我们日常使用的是有符号整数，但是在进行除法时操作数常常可以保证都是非负的，这时我们应当先将操作数转换为无符号类型再做除法：

```
a=(unsigned int)b/123;
```

无符号类型的除法比有符号类型进行得更快，所以这样确实可以起到优化作用。

关于其它的技巧，我选择了除以二的方幂、除以常数、商值较小 这三种情况进行研究。需要说明的是，我们这里所涉及的，全部都是整数除法。要手动实现浮点除法，过于复杂也没有多少提升空间，不做讨论。

除以二的方幂

首先要讲的是除以二的方幂，这个技巧已经为大家所熟知：除以 2 的 k 次方，等价于将被除数向右算术位移 k 位。在 CTSC2008 的 totem 一题中，要求我们将答案对 16777216 取模，而这个数字，恰好就是 2^{24}，所以在计算过程中我们可以照常计算任其溢出，而不会影响到数字末 24 个 bit 的值，这样会比使用了取模运算的程序快很多倍，这一点也会部分影响到题目最后的得分。

怎样找到位串中 1 的位置

不过我在这里主要想讨论的问题是，给出一个数 b，且已知 $b=2^k$，我们怎样最快地得到那个 k 值。

既然 $b=2^k$，那么在 b 的二进制表示中，一定是有且只有一个 1 的，我们只要找到这个 1 的位置，就得到了 k。线性地位扫描固然可行，但是速度无法忍受，使用对数函数虽然是理论 O(1)，但显然不切实际。还有一种容易想到的方法是将 b 进行截断，然后构造一张查找表，从高到低对每段判断是否为 0，如果不为 0 就在查找表中得到 1 的位置。这种方法在只分为 2 段时相当高效，可是需要一张 64KB 的查找表，虽然在今天的计算机中 64KB 可以忽略不计，但此方法终归不够优美。

我首先给出我所想到的在 C 语言下最好的方法：

```
char tb[32]={31,0,27,1,28,18,23,2,
             29,21,19,12,24,9,14,3,
             30,26,17,22,20,11,8,13,
             25,16,10,7,15,6,5,4};
//x 需要为 unsigned int
#define LOG(x) tb[((x)*263572066)>>27]
```

一看便知，我是构造了一张无冲突无浪费哈希表来做查询。它是怎样构造出来的呢？

当 2^k 乘以一个数时，其实就是将这个数左移 k 位，利用这一点，我构造出来一个数 T，使得对 T 分别左位移 0~31 位时，结果的高 5 位会取遍 0~31 的所有数。相信说到这里，大家都会想起那个基于欧拉回路的经典算法，这里不再赘述。

CPU 的位扫描指令

但是前面所说的这些方法，都不是我想讲的重点。换个角度思考，处理器能够在一个时钟周期内完成 add，mov 等指令，这些指令都需要访问全部的 32 个 bit，按理来说，处理器要想快速得到一个 1 的位置，是非常容易的，幸运地是，IA-32 处理器确实也提供了这样的功能，被称为位扫描（bit scan），位扫描指令分为 bsf（bit scan forward）和 bsr（bit scan reversed），前者从最低位向最高位进行扫描直到找到第一个 1 为止，而后者相反。有了 bsr 指令，我们就可以轻松快速地完成刚才的任务：

```
inline unsigned int LOG2(unsigned int x){
    unsigned int ret;
    __asm__ __volatile__ ("bsrl  %1, %%eax":"=a"(ret):"m"(x));
    return ret;
}
```

无疑 LOG2 可以比 LOG 宏更快速的运行，但优势决不仅仅在于速度。刚才的那个 LOG 宏其实并不能真正计算对数，而只能处理 x 为 2^k 的情况，但是 LOG2 就不同了。由于使用了 bsr 指令，它在找到最高位的 1 时就会停止，所以其返回值，确实就是 log2(x)取整后的值。如果你有一定算法编程经验，马上就会发现这个函数是多么的有用，我们求 RMQ 所用的 ST 表、求 LCA 所用的 log 在线算法、以及许多按位二分的算法，都需要使用这样一个功能，而刚才实现的 LOG2 函数，速度达到极致，又不需要多余的空间占用，堪称完美。

这里多提一句 bsf 指令，这条指令和 bsr 相反，是从低位往高位扫描找第一个 1，这首先让人联想到树状数组中使用的 LOWBIT 函数，不过使用 bsf 和 shl 实现 LOWBIT，并不见得会比 x&-x 来得快。我认为 bsf 指令最大的用处在于，可以用来得到一个二进制数末尾有多少个 0，并且一次性除去这些 0，在 Miller Rubbin、二进制 gcd 等算法中，都能发挥极大的作用。

编译器对除以常数的优化

接下来讨论除以常数的优化，在浮点领域，要计算除以常数 b 的商是很容易的，因为只要事先计算出常数 br＝1÷b，那么除以 b 就是乘以 br。到了整数领域，此方法显然就不再适用了，不过首先会想到的是，能不能采用类似的方法，因为虽然在整数域，1÷b 的值并不存在，但不要忘记一件事：计算机中的数域并不是真正的整数域，而是模 2^{32} 的剩余系，因此，看起来我们只需找到一个数 t，使得 $t\times b\equiv 1 \pmod{2^{32}}$ 就可以了，然后除以 b 就可以转化为乘以 t 了。真的是这样吗？

首先一点，t 并不一定是找得到的，如果熟悉基本的数论定理，容易推出当 b 为奇数时，一定找得到这个 t，当 b 为偶数时，一定找不到，证明很简单，不再赘述。于是就能想到一个改进方法，基于这样两个事实：除以 2 的方幂可以用位运算快速进行；a÷(b1×b2)＝a÷b1÷b2。第一个结论前文已经说过，这里简要证明一下第二个结论：

a÷(b1×b2)＝d → d×(b1×b2)＋q＝a (d,q 为非负整数,0≤q＜b1×b2)
a÷b1＝d1 → d1×b1＋q1＝a (d1,q1 为非负整数,0≤q1＜b1)
d1÷b2＝d2 → d2×b2＋q2＝d1 (d2,q2 为非负整数,0≤q2＜b2)
联立三式
(d2×b2＋q2)×b1＋q1＝d×(b1×b2)＋q
→ d2×(b1×b2)＋(q2×b1＋q1)＝d×(b1×b2)＋q
∵ 0≤q2×b1＋q1≤(b2－1)×b1＋(b1－1)＝b2×b1－1

$0 \leqslant q \leqslant b2 \times b1 - 1$ 且 $q \equiv (q2 \times b1 + q1) \pmod{b1 \times b2}$

∴ $q = q2 \times b1 + q1$

∴ $d = d2$

∴ $a \div (b1 \times b2) = a \div b1 \div b2$

有了这个结论，我们就可以把 b 分为两部分，令 $b = b1 \times b2$，且 $b1 = 2^k$，b2 为奇数，则计算 $a \div b$ 可以先将 a 右移 k 位得到 $a \div b1 = d1$，再计算 $d1 \div b2$，由于此时 b2 已是奇数，则可以使用刚才的算法。问题就这样解决了吗？

其实这个算法中有个漏洞，在 $a \times b = c$ 的时候，两边同乘以 $t(t \times b \equiv 1 \pmod{2^{32}})$，的确可以得到 $a = c \times t$，但是别忘了我们常常需要计算的是带余除法，也就是说 $a \times b + q = c (0 \leqslant q < b)$，这时再将两边同乘 t，得到 $a + q \times t = c \times t$，q 的值我们不知道，也没有办法绕过除法求得，也就没法得到 a 的值了。但无论如何我们已经得到了一个可以在 $b|a$ 时计算 $a \div b$ 的快速常数除法了。

现在我们主要的困难在于，整数除法的定义式中有两个未知数 a(商)和 q(余数)，虽然我们只想要知道商，但是无论如何恒等变形，都难以摆脱 q。那么这样的算法真正存在吗？

这时编译器是我们最好的老师，我使用 gcc -O2 编译如下的 C 代码：

```
#include<stdio.h>
int main(){
    unsigned int a;
    scanf("%u",&a);
    a/=7;
    printf("%u",a);
    return 0;
}
```

查看编译器生成的汇编代码，可以看到 a/=7 一句被如此编译：

```
movl    -8(%ebp), %ecx
movl    $613566757, %edx
movl    %ecx, %eax
mull %edx
subl %edx, %ecx
shrl  %ecx
addl %ecx, %edx
shrl  $2, %edx
movl    %edx, -8(%ebp)
```

-8(%ebp)是堆栈上储存变量 a 的位置。从这段代码我们可以看到，编译器确实仅仅使用了 add, sub, shr, mul 指令就完成了除法的工作，那么它是怎么做到的呢？

让我们从那个奇怪的数字 613566757 入手，这个数字是什么呢？受我们前一个方法的启发，将它和 7 相乘，并查看它的二进制表示。发现它恰好是 $2^{32}+3$。令 $t = 613566757$，我们将它乘入整除的定义式：

∵ $a \times 7 + q = c$

→ $a \times 7 \times t + q \times t = c \times t$

→ $a \times (2^{32} + 3) + q \times t = c \times t$

→ $c \times t = 2^{32} \times a + (3 \times a + q \times t)$

两边同时加上 $2^{32}\times c$

→ $c\times t+2^{32}\times c=2^{32}\times a+2^{32}\times(a\times 7+q)+(3\times a+q\times t)$

→ $c\times t+2^{32}\times c=2^{35}\times a+(2^{32}\times q+3\times a+q\times t)$

令 $C=c\times t+2^{32}\times c$，$A=2^{35}\times a$，$Q=(2^{32}\times q+3\times a+q\times t)$

我们假设 a 和 q 均取可能得到的最大值，即 $a=(2^{32}-1)\div 7=613566756$，$q=6$。

计算得 Q 的最大值为 31291904586，其二进制表示只有 35 位。

而 $A=2^{35}\times a$，其低 35 位均为 0，36 位开始为商的二进制表示。故只要计算出 C，将其右移 35 位，Q 中混乱且无法预测的结果全部会被屏蔽掉，而 a 的正确值就得到了。

编译器对这个算法的实现也是十分精巧的，本来 C 的最大值甚至会超过 64 位整数(仅超出一位)，但可以看到 gcc 仅用几条简单的 32 位指令就完成了任务。其中有一个小技巧值得一提，例如我们想计算 a 和 b 的平均数，可以写成(a+b)/2，但是 a+b 如果超出了当前可容纳数据范围，进位会被舍弃，除以 2 的答案也就不再正确了，正确的写法是(a-b)/2+b，这个式子可以对所有的 a,b 正确工作。上面的汇编代码中就使用了这个技巧来处理可能爆掉的 1 个 bit。

总结一下这个算法，它的巧妙之处在于突破了恒等变形的枷锁，式子变得混乱无关紧要，最后都会被一起丢弃，只剩下我们想要的东西。上面展示的这一种方法是编译器使用的几种方法中最为复杂的一种，编译器会根据常数的性质来选择不同的算法，剩下几种情况都更加简单，这里不再罗列，我自己编写了一个微型编译器，可以将 a/=b(b 为常数)编译为优化后的指令，我进行了充分的测试，对于所有输入我的程序均能与 gcc 产生一致的结果。

除常数的优化是如此有用，以至于就算不开优化开关，gcc 也会默认给我们做这样的优化，那学习它的原理有何用处呢？其实它可以扩展到非编译期常数的除法，意思是说，如果代码中时常需要除以一个同样的数字，但这个数字是在编译期无法确定的，那么我们就可以自己内嵌一个微型编译器，将优化后的代码编译为机器码存在内存中，需要做除法时跳转过去执行即可，不过这种优化实现代价太高，远远超出了竞赛的适用范围，这里不再展开讨论了。

用减法取代除法？

我要讲的第三点，其实并不是一个优化的技巧，仅仅是澄清一个误会。

我在 ACM 选手的代码中看到过一个 gcd 程序，它使用了一个很不优美的方式：首先手动从 a 中减去 b 数次，如果已经成功取模就可以直接递归，否则才调用除法指令。我刚看到时也深以为然，因为每次递归后的 b 实际上是模 a 的余数，将其看成是均匀分布的话，a÷b 的期望值应该非常小，用几次减法来取代它，想来应该会非常高效。可是我自己实现了代码进行测量，速度反而变慢了一倍，我继续采用更加强力的汇编优化，把简单的连续减法也换成了模拟的除法，效率仍然超不过最朴素的实现。经过了很多的实验，终于发现问题出在除法指令上：CPU 的除法指令效率并不稳定，在 a 和 b 非常接近时，速度也是可以达到非常快的。而手工的模拟，会导致一大堆的条件跳转，使 CPU 不能顺畅地执行代码。这个例子再一次告诉我们：优化决不是想当然的，不成熟的优化是效率低下的根源，只有具备丰富的经验，扎实的基础知识，实证的精神，才能做出真正精巧的优化。

二．乘法

相比除法，乘法的可研究之处就少了许多，从前面的指令时钟周期表中我们可以看出，现代 cpu 上的乘法指令已经足够快速了，所以一些老式的优化也已经过时了：例如将 a×10 写为(a<<3)+(a<<1)，这条语句包含了 2 条位移指令和一条加法指令，还隐藏有一些不可避

免的 mov 指令，其速度已经比不上直接让 cpu 做乘法了。

用 lea 指令优化常数乘法

但另一方面，(a<<3)+(a<<1)其实并不是最好的优化，让我们来看看 gcc 是怎么做的吧：

```
lea  (%eax,%eax,4),%eax
add  %eax,%eax
```

lea 指令本身是一条用来寻址的指令，不过被编程人员广泛使用于计算工作，其格式是 lea offset(reg1,reg2,c),dest，作用是将寄存器 reg1 中的值加上寄存器 reg2 中的值乘以 c 的乘积，再加上常量偏移 offset，将最终得到的结果存到 dest 中。c 是一个常数，取值只能为 1,2,4,8。那么 lea (%eax,%eax,4),%eax 的意义就是：eax=eax+eax*4，即 eax*=5。

在第一条指令将 eax 乘以了 5 之后，再自加一次得到 eax*10，这样简单的两条指令，又确实可以比乘法跑得更快了。不过显然并不是对所有的常数都有如此简单的优化方式，当我输入 a*14，gcc 就只好老老实实地调用 imul 指令了。所以说在乘以常数这个方面，基本上是没有多大的优化余地的，除非是乘以二的方幂，这里我们都很熟悉了：乘以 2^k 等价于将被乘数左位移 k 位。还有一些乘数很小的情况，也有一些巧妙的优化，这里展示几段 gcc 编译出的代码：

```
乘以 2：  addl %eax,%eax
乘以 9：  leal (%eax,%eax,8),%eax
乘以 13：leal (%eax,%eax,2),%edx
          leal (%eax,%edx,4),%edx
乘以 17：movl %eax,%edx
          sall 4,%edx
          addl %edx,%eax
乘以 68：movl %eax,%edx
          sall 6,%edx
          leal (%edx,%eax,4),%eax
```

优化 mul_mod 函数

关于乘法有一个容易被忽略的事实是，mul 指令其实是不会爆掉的，它将两个 32bit 整数相乘，结果保存为一个 64bit 整数，可惜高级语言隐藏了这一事实，也给我们造成了一些不必要的麻烦，我们竞赛中最常遇到的是，题目要求我们将答案对某个常数取模，那么我们就可以在中间过程中随时只保留取模后的结果，但在计算的中间过程，可能会因为乘法而溢出，无法得到正确的答案，在 C 语言层面我们的解决方式是，利用编译器提供的扩展 64 位整数来储存中间结果，就像这样：

```
inline int mul_mod_slow(int a,int b){
    return (long long)a*b%MO;
}
```

这样在速度上会比较慢，最快速且优美的解决方式其实处理器已经为我们提供了：mul 指令将结果储存在 edx:eax 这个 64 位整数中，而 div 指令同样从 edx:eax 中获取被除数，这样我们的汇编代码将会非常简便：

```
#define MO 123456789
inline int mul_mod(int a,int b){
    int ret;
    __asm__ __volatile__ ("\tmull %%ebx\n\tdivl %%ecx\n"
            :"=d"(ret):"a"(a),"b"(b),"c"(MO));
    return ret;
}
```

这个函数是简洁而且高效的，在我的机器上，mul_mod 函数比 mul_mod_slow 函数快了一倍还要多。

既然说到了这里，就再顺便提及一个技巧，它用来解决这样一个问题，如果要求我们模的常数是一个 64bit 整数，那在做乘法时就没有扩展类型给我们使用了，这时看上去必须要我们手动编写高精度整数运算的代码了。

幸运的是，有一个巧妙的方法可以简单解决这个问题：

```
typedef long long ll;
#define MOL 123456789012345LL
inline ll mul_mod_ll(ll a,ll b){
    ll d=(ll)floor(a*(double)b/MOL+0.5);
    ll ret=a*b-d*MOL;
    if(ret<0)ret+=MOL;
    return ret;
}
```

首先它使用浮点运算来得到 a*b/MOL 的值，关键在于第二句，显然 a*b-d*MOL 中的两个乘法都是可能会溢出的，不过没关系，因为我们可以预见其差是一个 64bit 可容纳的正整数，那么溢出部分的差仅可能为 0 或者 1，最后一句符号的特判用来处理溢出部分差为 1 的情况。容易注意到，我们计算 a*b/MOL 时使用了浮点运算，误差是不可避免的，故建议不要对太大的 MOL 使用这个算法。（感谢 crazyb0y 无私地给我提供这个技巧）

三．高精度运算

在我看来，编写大整数运算的代码本来就应该是属于汇编语言的工作，在 IA-32 指令集中，有一大套的指令几乎就是为了大整数运算而存在的。在汇编语言层次，我们可以采用 2^{32} 进制来存储整数，以获得最高的运算效率和指令支持，其速度是非常惊人的。我用内嵌汇编实现了这样一套函数库，在我的机器上，它比我们传统使用的方法要快上数倍。在附件中的 bigint.c 文件中可以找到这个实现。

CPU 优化特性

CPU 是计算机的心脏，我们的程序代码就会在这里被执行，所以要研究优化，深入到 CPU 的内部是很有意义的。

CPU 执行效率≠主频

可能有的人会粗浅地认为，CPU 执行程序的速度，主要是由 CPU 的主频所决定的，CPU

的主频代表 CPU 在一秒钟内可以运行多少个时钟周期(clock cycles)。这在十多年前可能还是相当正确的，那时 CPU 的主频提升很快，基本符合摩尔定律所预计的，十多个月就能够提升一倍，所以可以寄希望于 CPU 主频地不断翻倍来提升我们电脑的性能。但是近几年，受制造工艺和功耗的限制，CPU 的主频已经不再能够提升了，那么，就是说 CPU 的性能也不再提升了吗？当然不是，想想 8088 的时代，那时执行一条简单的 16-bit 整数乘法指令，都要消耗掉上百个时钟周期，而现在呢，平均只需要使用 1~2 个周期就够了，之所以能有这样的飞跃，是因为现代 CPU 不再局限于将一条条指令简单地执行，而引入了许多复杂的优化机制。例如 Intel Pentium 4 处理器中采用了名为 NetBurst 的控制单元技术，它内部集合了各种高效的优化策略，例如指令预取、分支预测、乱序执行等等，使我们的代码可以更加快速地执行。但有一个现实是，特定的优化技术对不同形式的代码的优化效果是不同的，也就是说，我们想实现同样一个功能，一些实现上的细微差别，可能在底层会影响到 cpu 的优化，效率也会天差地别。所以，如果我们能够对 CPU 所使用的优化技术有所了解，就可以有意识地在编码时去配合 CPU 工作，而达到更高的效率。接下来，我们就一个一个地来看看 CPU 内部的优化机制，它是什么，怎么工作，怎样去配合它。

注：这一部分本来理应完全在汇编语言层面进行研究，可以讲述的规则与技巧也太多，但出于更贴合我们竞赛的一些考虑，我大量删减了这一部分的篇幅，只留下了一些在高级语言层面也有意义的内容。

一. CPU 高速缓存机制

内存的访问是缓慢的

我们使用高级语言编写程序，变量(variable)是一个非常重要且常用的概念，我们整个程序中都使用变量来存储数据，保存程序运行状态，传递参数，在汇编语言中的核心概念“寄存器”到了高级语言中已经被隐藏掉。但有一个事实是，变量是储存在内存中的，而内存的访问速度是远远跟不上 CPU 的运行速度的，在 CPU 需要访问内存时，它需要停下来，将请求通过控制总线和地址总线发送给内存，然后等待着内存把数据准备好，再去提取数据。这个过程是相当漫长的，需要占用几十上百个时钟周期，严重影响了 CPU 整体的执行效率，现代 CPU 从两个角度来解决了这个问题，其一是让 CPU 在等待内存准备数据时可以抽身去做后面的事情，其二是使用缓存(cache)的思路。第一点在后面乱序执行部分会讲到，现在我们来研究 CPU 的缓存机制。

高速缓存的工作原理

缓存基于这样一个事实，我们的程序总是倾向于执行相临近的指令码，并使用相临近的内存数据，一个内存位置在访问后有很高的几率会在短时间内被再次被访问。于是 CPU 在内部加入了多级高速缓存，级别越高，离 CPU 越近，访问速度越快，但相应的造价也越高，容量也越小。例如在我的电脑上有两级缓存，一级缓存大小为 64KB，二级为 2MB，CPU 访问一级缓存的速度几乎可以看作是在访问寄存器。

CPU 缓存具体是怎样工作的呢，首先，当一个变量需要被加载到缓存中时，并非简单地将这个单一的内存位置加载，而是一次性加载一行，在我电脑上一行为 64 字节，也就是说地址的低 6 位不同，而高位相同的这一段连续的内存位置会被一次性加载进来。我的 1 级缓存大小为 64KB，算下来一共可以加载 1024 个不同的行，但一个特定的内存位置并不能随意加载到其中任意一行，而只能加载到它所在的组(set)中，每个组有 8 个行，那么就有 1024 ÷8＝128 个不同的组，128＝2^7，于是就通过地址的 6~12 位来决定当前行该加载到哪一个组

中，如果当前组中已经加载满了 8 个行，显然就只有将其中一行从缓存中删除，下次再访问到此行就需要重新加载了。当需要访问某一个内存位置时，CPU 会首先检查它是否被加载到缓存中，如果已经加载，就直接在缓存中读写之，这称之为缓存命中(hit)，仅仅需要相当少的延迟时间。但是如果没有的话，就需要重新从内存中进行加载，称之为 cache miss，这个代价是十分巨大的，我们写程序时需要尽可能地去避免它，这也是我们需要了解高速缓存工作机制的原因。

实验观察高速缓存的存在性

为了可以直观地看到高速缓存的存在，我们先来看如下一个程序。

```
int sum=0;
for(i=0;i<N;i++)sum+=a[i];
```

这个程序简单地将 a 数组的所有元素求和，为了更加明显地看出效果，这里使用了-O2 进行编译。N 的值为 10000000，运行十次并测量其占用的时钟周期数，以下是测量结果。

```
TIME0:  167150043
TIME1:  24793497
TIME2:  27947781
TIME3:  25602750
TIME4:  27184617
TIME5:  18583821
TIME6:  15880428
TIME7:  15828174
TIME8:  15944328
TIME9:  16141635
```

可以很明显的看出，第一次求和占用的时间是后面的十倍左右。这主要是因为 a 数组是没有经过初始化的，所以整个都不在缓存中，使得第一次遍历需要极大的代价，所以说，缓存是切实存在而且不可忽视的。

怎样使高速缓存更好地工作

那么，我们该怎样编写代码来配合 CPU 缓存的工作呢？

站在高级语言的层面，我们主要从不同的变量类型的角度来看。

尽量使用局部变量

最重要且有用的一点就是：尽量使用局部变量。局部变量所处的位置是程序的运行时堆栈，这里的数据访问非常频繁，且数据很集中，在大部分时候，堆栈顶部的数据都会处于 CPU 的一级缓存中，访问速度非常快。而相对的，全局变量需要使用一个 32 位的地址来进行寻址，使得指令码非常长，不利于 CPU 的各种机制的工作，而且缓存命中率也远远不如局部变量来得要高。

使用结构体组织相关联的数据

另一方面，尽量使用结构体来组织相关联的数据，例如：

```
int x[MAXN],y[MAXN];
```

在普遍情况下就不如使用 struct 的效率来得高：

```
struct point{int x,y;};
point p[MAXN];
```

道理也很简单，因为 x 和 y 是相关联的数据，所以可以期望，当访问到 x 时，我们一般马上还会同时用到 y，如果把他们放在一个 struct 中，任意一个元素被访问时，整个结构都会进入缓存，速度会很快。

数据的对齐

说到这里，很容易发现数据对齐也是很重要的，假设我们的 point 数据类型在内存中的放置跨越了缓存行的边界，每次访问都需要加载两个缓存行，效率之低下是可想而知的。数据对齐的一般原则是，普通变量应该按照它自身的大小对齐，较大的 struct 应该按照一个 2 的方幂的边界对齐。C 语言标准中并没有提供对于对齐的支持，但是各个编译器都有提供自己的语法，在我们竞赛使用的 gcc 中，要求编译器辅助对齐的格式是：

```
int array[1024] __attribute__((aligned(64)));
```

联合数据类型

C 语言中还有提供一种叫做联合(union)的数据类型，往往被人们所忽视了，其实它也有一定的实用价值。

```
union{
    A a;
    B b;
};
```

在上述代码中,a 和 b 是不同的数据类型,但是由于使用它们的代码不会同时使用两者，所以可以让他们的内存位置相重叠。这样做不单单是可以节约一点内存，而且使用完 a 再来使用 b 时，由于 a、b 的内存位置相同，所以 b 会也存在于高速缓存中，从而获得较高的访问效率。

避免动态内存分配

最后一点，动态内存分配是应该尽量避免的，虽然它有众多的好处，但是不可回避的是其极低的效率，分配和释放时系统会遍历一个内存块链表进行查找，而且其分配出来的内存块可能会相当零散，不利于 CPU 缓存的工作，如果在程序中大量使用动态内存分配来分配和释放小型对象，分配内存本身很有可能成为效率瓶颈。但在竞赛中，我们也时常需要实现各种动态数据结构（例如二叉平衡树），较好的解决方法是先开一个足够大的内存池，以后需要分配节点就手动从内存池中取出。另外，结合前面所讲，在实现这些动态数据结构时，使用 struct 来表示一个节点也比使用各个分离的数组要来得高效。

一些例外

凡事并非绝对，我们在编写程序时还时常需要开一些尺寸很大的数组，这些数组也应该开在堆栈上（局部变量）么？当然不是，如果开在堆栈上，不但很容易造成堆栈空间溢出，而且如果堆栈指针 esp 常常很大地变动,会使得堆栈顶部的数据常在高速缓存中这一优势荡然无存，对效率有害无益。另外，gcc 支持运行时提供局部数组尺寸的功能，看起来很实用，

但实际上完全不建议使用，不但具有前面所述的所有缺点，而且使得最朴素的堆栈操作也变得很复杂，效率会大大受到影响。

代码缓存

以上提到的都是 CPU 的数据缓存，其优化的是程序代码访问内存数据的效率，不过不要忘记了，程序代码本身也是存储在内存中的，由 eip 指针所指涉，每当需要执行一条指令时，CPU 会取出 eip 所指向的指令，解码并执行，按理来说，这里访问内存的频率比代码本身访问数据的频率还要高，自然也是 CPU 优化的要点，而优化的方式，同样也是使用高速缓存。CPU 的代码缓存和数据缓存工作方式基本一致，这里只简单来看看在高级语言层面怎样去配合代码缓存工作。

相邻放置相关代码

第一个原则是，将功能上有关联的函数在源代码中的位置放得尽量地近，例如一个函数 f 中要调用函数 g，就可以将 f 的源代码和 g 的源代码相邻放置，这样在第一次执行 f 函数时，就有机会将 g 的代码一起加载到缓存中。类似地，在整个程序中可以将调用频率最高的一些函数相邻放置，这样它们就有机会随时保持在代码缓存中，而那些很少调用的函数，可以放得远一点，它们被加载到缓存中只是浪费空间。

减小循环体尺寸

毫无疑问，程序中的时间瓶颈所在一般都在一个循环中，但要注意，真正占用时间的一般是循环体，而不是循环语句本身。循环体需要被反复地执行，第一次进入循环体时，循环体代码不可避免地会被从内存加载到代码缓存中，但是到第二次进入循环体时，我们期望这时循环体的代码仍然在代码缓存中，如果每次循环都需要重新从内存中加载代码，效率将会受到影响。为了避免这种不利情形的发生，保持循环体内容尽量简介很有帮助。例如有如下的代码：

```
for(int i=0;i<1000000;i++){
    proc1();
    proc2();
}
```

如果 proc1 和 proc2 干的事情毫无关联，同样可以写成这样：

```
for(int i=0;i<1000000;i++)proc1();
for(int i=0;i<1000000;i++)proc2();
```

那么哪个更加高效呢？直观地看来，第二段代码比第一段干了更多的事情：计数器变量 i 的比较和累加需要进行两次，浪费了时间，确实，如果 proc1 和 proc2 都非常的简单，都只有很少的几条指令，第一段代码或许会更有效率。但是，如果 proc1 和 proc2 都是很复杂的过程，那么循环语句本身所占用的时间几乎可以忽略不计，另一方面，在单个循环体中运行的代码更短了，一来代码本身有更大的可能全部存在于高速缓存中，更重要的是，这段代码所访问的数据也会更多地保存在一级缓存中，还有即将要提到的分支预测引擎，也会更好的工作。

总结起来，对于 CPU 缓存类型的优化，我们编写代码的大体原则就是：本来就有关联的东西（变量、代码），尽量在代码组织上让它们紧密地联系起来。本来毫无关联的东西，

就应该把它们相分隔开。这不单单是可以让我们的代码更加有效率，同样也是编写更优美的代码的基本原则。

二. 分支预测

仅仅简单地顺序执行一条一条的指令并不能满足程序设计的需要，为了描述程序的逻辑，我们至少需要一种条件性的改变程序执行流程的方式，对于高级语言，我们有各种 if 语句，形式复杂的循环语句，而翻译到汇编语言，就只有一组简单的条件跳转指令了。这些条件跳转的形式是，当某个 CPU 标志位被置位(set)时，跳转到某个内存地址执行，否则顺序执行，从而将程序分成两路。

条件跳转是低效的

总的来说，条件跳转指令是低效的，它会影响到代码缓存、踪迹缓存(trace cache)、CPU 流水线等机构的工作。而对 CPU 流水线的影响是最大的。

现代 CPU 执行一条指令，需要经过十多个复杂的步骤，例如指令预取、指令解码、寄存器重命名、执行指令、指令退役等等，如果每处理一个指令它都需要独占 CPU，那定然是会非常低效的。所以现代 CPU 都引入了流水线机制，每一条指令都好像流水线上的一个产品，经过一道道工序最终完成，一条指令还没有完成时，已经有许多条后续指令已经进入流水线开始处理了。这对于指令顺序执行时是很有效的，但是一旦出现一条条件分支指令，CPU 就会不知道该把哪一路指令放入流水线，现在它可以选择暂停流水线，当条件分支指令的走向确定了以后，再重新启动流水线，但这样会有 10 来个时钟周期的代价，是十分巨大的。为此 CPU 使用了分支预测(branch prediction)技术，它通过一系列算法来估计当前分支的走向，然后直接将这个分支的指令进入流水线开始处理，这样流水线就不需要停工，但是一旦误测(misprediction)，cpu 需要倒回去修正它犯下的错误，然后重新执行正确的分支，这个代价极其巨大，会高达几十个时钟周期。CPU 使用了复杂的算法来做这个预测，它对于在实际应用中可能会遇到的许多模式都能很好的工作。

实验观察分支预测的存在性

我们还是先通过一个实际的例子来感受它的存在性：

```
int main(){
    for(int i=0;i<N;i++)a[i]=i%2;
    ull start=get_clock();
    for(int i=0;i<N;i++){
        if(a[i])a[i]=123123123;
        else a[i]=1234123;
    }
    printf("%llu\n",get_clock()-start);
    return 0;
}
```

在我的电脑上，输出为 32889996。

现在我们将第二行改为：

```
    for(int i=0;i<N;i++)a[i]=rand()%2;
```

输出变成了：102583557。

注意到我们的改变根本没有涉及到被测的代码，在第一个程序中，a 数组中的值为交替的 0 和 1，而在第二个程序中 a 数组中的值以 50%的概率随机为 0 或者 1。CPU 的分支预测技术有能力探寻出第一种模式，从而高效地执行，但是对于纯粹随机的数据它却无能为力。

提高分支预测的成功率

要优化条件分支语句，一方面可以考虑适当地组织代码使得分支预测的成功率尽量高，为了做到这一点，我们需要大致了解条件分支算法的特点。在以下情况下，分支预测的正确率会相当高：

- 如果这个分支总是走向同一方向
- 如果分支按照某种简单的模式重复
- 如果这个分支和前面的某个分支相关联

由此也可以看出，在循环语句中计数器的比较操作所导致的条件分支还是相当安全的，因为它总是走向同一方向（除了最后一次退出循环）。

消除条件分支

和对付除法类似，优化条件分支的另一个办法是消除掉它，下面给出一些比较有启发性的例子：

(1)对 eax 取绝对值：

```
cltd
xorl %edx,    %eax
subl %edx,    %eax
```

来分情况看看为什么这样做是正确的：

如果 eax≥ 0，设 a=eax

```
after line 1:    eax=a     edx=0
        line 2:  eax=a     edx=0
        line 3:  eax=a     edx=0
```

如果 eax<0

```
after line 1:    eax=a          edx=-1
        line 2:  eax=~a         edx=-1
        line 3:  eax=~a+1       edx=-1
```

而~a+1 就是在计算机中-a 的表示法

(2)实现 if(ebx>eax)ebx=eax：

```
subl %ebx,    %eax
sbbl %edx,    %edx
andl %eax,    %edx
addl %edx,    %ebx
```

(3)实现 int cmp(int a)函数，当 a=0 时返回 0，当 a>0 时返回 1，当 a<0 时返回-1：

```
int cmp(int a){return (a>>31)+(-a>>31&1);}
```

需要注意的是，在高级语言中的条件语句并不简单地对应汇编语言中的一条条件跳转指令，而是逻辑表达式中的每一项都会产生一个条件跳转，所以在编写高级语言中的 if 语句时应该尽可能地精简。

三. 乱序执行

CPU 的本职工作，是执行一条条的指令序列，仅仅优化了对内存的访问以及跳转指令的效率并不解决根本问题，最核心的需求是，对于一大段的指令代码，有没有有效的方法可以让它跑得更快。现代 CPU 在这上面已经走得很远，其中最为有效的一个技术被称为乱序执行(out-of-order excution)，乱序执行，正如字面含义，cpu 企图打乱各条指令的顺序，以获得更高的执行效率。但是想想我们平常看到的汇编程序，好像只有很少的指令顺序可以交换，试图去打乱顺序似乎是浪费时间。CPU 的乱序执行确实并不是这样单纯，它主要依赖于两个重要的辅助技术：寄存器重命名(register renaming)和微指令(microcode)。

微指令

微指令简称为 uop，是一套类似于 risc 的指令。现代 cisc 为了弥补和 risc 处理器的差距，将外部送入的 x86 指令先进行解码，分解为更为细小的操作，再送给 cpu 的核心进行处理，当然这个解码过程本身也是需要时间的，这个环节甚至一度成为瓶颈之一，需要程序员在编写代码时注意许多规则来优化解码，不过现在新的处理器中都引入了踪迹缓存(trace cache)技术，这种技术会缓存外部指令解出的 uop，使得解码部分不再成为瓶颈，不过有一个永远不变的注意事项是：指令码的体积越小越好，对解码单元也好、踪迹缓存也好都是有很大帮助的，关于如何缩小指令码的体积，又是汇编语言的智力游戏，不在这里展开讨论。

微指令的例子

我们先来看看 uop 是什么样的：

```
addl (%esi),  %eax
```

这条指令将 esi 指向的内存数据加到 eax 中，它将会被分为两条微指令，第一条加载 esi 指向的内存，第二条做运算并保存结果到 eax。

而相反的， addl %eax, (%esi) 这条指令会被分解为三条 uop，第一条加载 esi 指向的内存，第二条将其值与 eax 的值相加，第三条将运算结果存回 esi 指向的内存位置。

微指令与乱序执行

代码一旦被分解为 uop，乱序执行就容易进行多了。CPU 内部有着多个不同的通道，有的负责从内存中加载数据，有的负责将数据存储到内存，有的负责执行整数指令，有的负责浮点指令。不同的 uop 在被安排为适当的顺序后，可以在各个通道并行地执行，例如这一条指令正在等待从内存中加载一个不在缓存中的数据，这时其它几个通道可以提前开始进行后面的整数运算，CPU 的工作能力可以被充分的利用起来。

寄存器重命名

但现在还存在一个阻碍乱序执行良好工作的障碍：我们可以使用的寄存器数目是很有限的，我们常常会连续使用同一个寄存器来完成某些工作，而这些工作本身却是不相关的，例如如下代码：

```
a+=b;
c+=d;
```

这两行代码在编译后一般是如下形式：先将 b 内存位置加载到 eax 中，再将 eax 加到 a 内存位置。然后对 c，d 做同样的工作。

假如变量 b 并不存在于高速缓存中，那么在执行到第一条指令的时候就会发生延迟，这时如果 d 在缓存中，就可以同时开始进行 c+=d 的工作了。但问题在于，现在两行代码都使用 eax 作为中间寄存器，看起来 c+=d 必须等待了，除非对于第二行代码使用 ebx 作为中间寄存器。如果每一行代码都换寄存器使用的话，很快就没有寄存器可用了。

什么是寄存器重命名

寄存器重命名技术被发明出来解决这个问题，现代 CPU 内部实际上有大量可用的物理寄存器，我们编程时可见的 eax 等寄存器只是一个虚拟的概念，被称为逻辑寄存器。在代码需要使用某个逻辑寄存器时，CPU 内部的寄存器分配表(register allocation table,RAT)会将某个物理寄存器映射到它上面，到了 uop 的级别，eax,ebx 等名字已经不存在了，取而代之的是其映射的物理寄存器。在上述代码中，虽然两行都会用到 eax 寄存器，但是在经过寄存器重命名之后，它们已经是不同的寄存器，就有机会乱序执行了。

CPU 流水线

现在我们已经可以理清乱序执行以及相关优化机制的大概流程了,CPU 的指令预取单元会提前读取接下来要执行的一定数目的指令，将它们送入解码单元，经过解码和寄存器重命名后，进入重定序缓存(reorder buffer)被适当地打乱顺序，然后再送入执行单元正式执行，执行后结果将被保存起来，最后退役单元(retirement unit)将它们安排为正确的顺序，我们就说这些指令退役了，也就是正式执行完毕了。

从这个流程我们可以看出，如果一开始的“指令预取”一步就出现了问题，那后面做的这么多步骤都是无用功了，全部必须推翻重来，再次印证了条件分支的危险性。

降低语句间的依赖关系

在一条指令被执行的这个复杂过程中，有许多环节都有可能成为瓶颈，如果编写汇编代码，有太多的细节值得注意。但如果是使用高级语言来编写代码，处理器级别的细节基本已经被完全隔离开来,我们不需要也没有办法去干涉背后的行为,编译器会帮我们处理一切。惟独有一条原则仍然可以指导我们编写高级语言代码：降低语句之间的依赖关系。

高级语言语句，经过编译器的编译转换为数条汇编指令，我们没有能力也没有精力在编写高级语言时还在 cpu 级别思考微指令乱序执行的情况,但高级语言中充斥着对内存变量的使用，内存变量不像寄存器，没有重命名单元，如果两句高级语言语句使用了同一个内存变量的话，后一句是一定要等前一句执行完毕后才可以开始执行的，这样就影响了 CPU 流水线的工作，我们来看一个例子：

```
double sum=0;
for(int i=0;i<N;i++)sum+=a[i];
```

a 是一个 double 类型的数组，N 的大小为 10000000。这段代码的运行时间为：88532892 个时钟周期。

简单地将其改为 4 路加法：

```
double s0=0,s1=0,s2=0,s3=0;
for(int i=0;i<N;i+=4){
    s0+=a[i];s1+=a[i+1];
    s2+=a[i+2];s3+=a[i+3];
}
double sum=(s0+s1)+(s2+s3);
```

运行时间成功减少到了 43307199 个时钟周期。稍加分析，在第一段代码中，虽然 cpu 很可能能成功地进行分支预测并进行指令预取，但是在 sum+=a[i]一句中，sum 依赖于上一次累加的结果，而 i 依赖于循环计数器刚进行累加后的值，于是所有指令都只有一个接一个按部就班地执行，效率自然会有问题。而第二段代码中，这一条依赖链被剖分成了互相独立的四路，即使不去探究它们被翻译成了怎样的汇编代码，也可以想象这样的代码是能很好地配合 cpu 内部机制的工作的。

使用 32 位数据类型

这里还有一个基本原则：除非在内存十分紧张的情况下，应该使用 32 位数据类型来存储所有程序需要用到的整数。其实这也能够帮助 cpu 乱序执行引擎的工作，如果使用 16 位数据，那就不可避免地需要使用 ax 等寄存器，但是这些寄存器在 cpu 内部其实是不存在的，而仅仅是 32 位寄存器的一个局部，它们是不支持寄存器重命名的，例如在上一句语句中使用了 eax 寄存器，这一句中又使用 ax 寄存器，就会造成延迟。不使用局部寄存器还有很多理由，例如非 32 位指令的机器码长度会更长而不利于很多 cpu 内部机制的工作。总之，在 32 位计算机上，请使用 32 位数据类型。

位运算技巧

在计算机内部，一切数据都是以二进制进行储存，其支持的最基本的元操作应该是布尔逻辑运算，例如逻辑与(and)、逻辑或(or)、逻辑非(not)、逻辑亦或(xor)等等，这些运算的实现，几乎可以说是直接与电路的基本元件逻辑门相对应起来，是计算机天生所最擅长的操作，其执行速度无疑也是最快的。至于整数的四则运算等操作，是用众多逻辑门所搭建起来的，加减法电路的逻辑门数量比较少，同样可以在较短的时间内进行运算，但如果像除法一般逻辑复杂，电路也相应庞大，速度就很难以接受了。再到更复杂的浮点运算，速度也就更慢。简而言之，在众多的 CPU 指令中，凡是在 bit 级别进行操作的，基本上都能够在一个 clock cycle 中完成，例如位移(bit shift)、位扫描(bit scan)、逻辑运算等等，再加上加减法、取补码等等简单的整数运算，组成了一个非常高速的指令集合。而巧妙地组合这些运算，往往能取代一些原本缓慢的操作，获得速度上的极大提升。

同样，本文不在汇编级别研究这个问题（虽然可能在汇编级别思路更为广阔、优化效果也会更好），这里列出一些我所收集或者原创的 C 代码位运算技巧：

一. 位压缩的技巧

在竞赛中，为了节省内存、加快运算速度或者进行状态压缩 DP，我们时常需要将一组布尔变量压缩到一个打包的 32bit 变量中。要读写其中的单个 bit，使用位运算是最合适不过的了。下面的代码都假设已经声明了一个 32bit 变量 a，它的 32 个 bit 是表示 32 个布尔标志：

读取第 k 位：	a>>k&1
读取第 k 位并取反：	~a>>k&1
将第 k 位清 0：	a&=~(1<<k)
将第 k 位置 1：	a|=1<<k
将第 k 位取反：	a^=1<<k
将第 k1~k2 位反转：	a^=((1<<(k2-k1+1))-1)<<k2
是否恰好只有一个 true：	!(x&(x-1))&&x
判断是否有两个相邻的 true：	x>>1&x
是否有三个相邻的 true：	x>>1&x>>2&x

二. 打包位统计

延续上面的话题，在 bit 被打包以后，我们时常想要快速知道一些关于整个一个包的信息，例如里面有多少个 true 等，固然可以一个 bit 一个 bit 地处理，不过这样就使打包的优势荡然无存了，想要 O(1)又没有特别好的办法，一个比较常见的处理方式是先预处理一张比较小的表，例如 2^{16}，然后将要查询的包分段查询再汇总。这里要介绍的方法技巧性比较强，纯粹使用位运算在对数时间进行处理。

(1)统计 true 的个数的奇偶性：

```
x^=x>>1;x^=x>>2;
x^=x>>4;x^=x>>8;
x^=x>>16;
```

运算结果的第 i 位表示在原始数据中从第 i 位到最高位 true 数目的奇偶性，有了这个结果，我们就可以很方便地得到任意一段的奇偶性：如果想要得到 k1~k2 位中 true 个数的奇偶性，直接计算(x>>k1^x>>(k2+1))&1 即可。

(2)统计 true 的数目：

```
int count(unsigned int x)
{
    x=(x&0x55555555)+(x>>1&0x55555555);
    x=(x&0x33333333)+(x>>2&0x33333333);
    x=(x&0x0F0F0F0F)+(x>>4&0x0F0F0F0F);
    x=(x&0x00FF00FF)+(x>>8&0x00FF00FF);
    x=(x&0x0000FFFF)+(x>>16&0x0000FFFF);
    return x;
}
```

又是一个基于二分思想的算法，需要特别注意，传入的参数 x 的类型是 unsigned int。在位运算中，我们所期望的类型都应该是无符号整数，因为对于有符号的类型，在进行右位移时将采用算术位移(arithmetic shift)的方式，对应于指令集中的 sar 指令，它和 shr 指令的行为是有差异的，如果这里出了错，往往会导致艰苦的调试。有符号数据常出的另一个问题是，在进行类型扩展时，不是填充 0 而是扩展原来的最高位，这里也常常导致非常难以发现的错误。总之，在进行位运算的子程序中，推荐全部使用无符号数据类型。

(3)反转位的顺序：

```
unsigned int rev(unsigned int x){
    x=(x&0x55555555)<<1|(x>>1&0x55555555);
    x=(x&0x33333333)<<2|(x>>2&0x33333333);
    x=(x&0x0F0F0F0F)<<4|(x>>4&0x0F0F0F0F);
    x=(x&0x00FF00FF)<<8|(x>>8&0x00FF00FF);
    x=(x&0x0000FFFF)<<16|(x>>16&0x0000FFFF);
    return x;
}
```

原理和上一个程序基本一样。

三. 消除分支

这个主题在前面讲 CPU 的分支预测机制时已经提过，现在再给出一些 C 语言层面的技巧：

(1)计算绝对值：

```
int abs(int x){
    int y=x>>31;
    return (x+y)^y;
}
```

(2)求较大值：

```
int max(int x,int y){
    int m=(x-y)>>31;
    return y&m|x&~m;
}
```

(3)x 与 a,b 两个变量中的一个相等，现在要切换到另一个：

```
x^=a^b
```

四. 其他

不使用额外空间交换两个变量：

```
void swap(int& x,int& y){x^=y;y^=x;x^=y;};
```

计算两个整数的平均数，不会溢出：

```
int ave(int x,int y){return (x&y)+((x^y)>>1);};
```

使用高维数组的注意事项

我们日常使用的计算机，内存均为线性结构，一个一个的字节依序排列起来，就和我们在 C 语言中使用的一维数组的结构基本一致。但是在实际编写程序解决问题时，简单的线性结构往往是不够的。例如在进行几何变换、线性递推等工作时，矩阵是一个强有力的工具，但是矩阵本身就是一个 2 维的数据结构；又比如在使用动态规划算法编写程序时，高达 3、

4 维的数组结构也非常常见。

高维数组在内存中的安放

显然高维度的数组本身是不能直接放置在一维的内存中的，为了达到这个目的，需要进行一次映射，这个过程也很容易想象：对于二维数组，就先将第一行顺次放入，紧接着放入第二行……一直到整个数组都被放入为止。对于更高维的数组，可以将其内存布局想象成一个递归的过程：为了将一个 d 维数组放置在内存中，假设其第一维的尺寸为 x，那么就递归地将 x 个 d-1 维的数组放置在内存中，当 d 下降到 1 时就可以直接线性安放了。幸运的是，在 C 语言中我们不用手动去实现这样一个映射，C 语言本身提供了对高维数组的支持，但千万不要忘记的是，C 语言在编译器级别同样做了那样一个复杂而且耗时的映射，我们来看一个例子：

```
int a[10][10][10][10][10];
int main(){
    int x=3,y=7,z=4,w=2,l=1;
    a[x][y][z][w][l]=5;
    return 0;
}
```

这是一个再简单不过的程序，我们声明了一个 5 维数组 a，并去写入其中一个位置，那么来看看编译器生成了怎样的汇编代码：

```
movl    -20(%ebp), %ecx
movl    -24(%ebp), %ebx
movl    -28(%ebp), %esi
movl    -32(%ebp), %edx
movl    -36(%ebp), %edi
movl    %edx, %eax
sall  $2, %eax
addl %edx, %eax
leal  (%eax,%eax), %edx
imull    $10000, %ecx, %eax
addl %eax, %edx
imull    $1000, %ebx, %eax
addl %eax, %edx
imull    $100, %esi, %eax
leal  (%edx,%eax), %eax
addl %edi, %eax
movl    $5, a(,%eax,4)
```

无需去细看这些代码做了什么，这里有 17 条指令，其中还包括 3 条相对耗时的 imul 操作，可以预见这段代码会消耗 20 多个时钟周期，但它所做的工作，仅仅是 a[x][y][z][w][l]=5 这简单的一句话。

减少高维数组寻址

有了这样一个印象，对我们编写代码会有很大的帮助，例如在常见的动态规划程序中

可以看到类似这样一句话：

```
if(a[x-1][yy][z-3]<a[x][y][z])a[x][y][z]=a[x-1][yy][z-3];
```

重复对高维数组进行寻址的代价将是不可忽视的，这样写效率就会高上许多：

```
int &mn=a[x][y][z],v=a[x-1][yy][z-3];
if(v<mn)mn=v;
```

再比如，我们对一个多维数组的读写时常并不是随机顺序的，在许多情况下，我们访问了一个多维数组中的元素，接下来要访问的元素就在它的附近。

例如当我们遍历一个高维数组时，访问了 a[x][y][z]之后就将访问 a[x][y][z+1]，而其实 a[x][y][z+1]的地址可以直接通过 a[x][y][z]的地址递增得到。又比如，在动态规划程序中，我们可能总是会从 a[x-1][y-2]去更新 a[x][y]，在这种情况下，预先算出一个偏移量，然后就可以通过简单的指针加减运算来访问两个内存位置了。

在编写和数组相关的代码时，强烈建议尽量使用指针来进行所有操作，并且尽量不去依赖于编译器实现的多维到一维的映射。就算是进行简单的数组遍历，也能取得很可观的优化效果，这一个技巧在周以苏前辈的论文中有详尽的讲解，此处不再赘述。

高维数组访问的底层效率

以上的内容都停留在汇编层面，更具体地说，就是“读写高维数组需要经过一个复杂而耗时的映射过程”这一事实。其实我们可以在更加底层来看待问题，先来看这样一个例子：

```
int a[2048][2048];
int main(){
    int i,x,y;
    for(i=1;i<=100;i++){
        for(x=0;x<2048;x++)
            for(y=0;y<2048;y++)
                a[x][y]=x+y;
    }
    return 0;
}
```

这个程序对一个 2048×2048 的二维数组进行遍历并赋值，外层再循环 100 次来放大效果，我简单地使用 linux 的 time 命令来测量时间，此程序运行的 user time 为：1.760s。

现在我交换 x 与 y 循环的顺序，即改为按照列顺序来访问，按理来说，这个改动在 C 语言级别，甚至在汇编语言级别都应该是无伤大雅的，我们再次运行程序并测量时间。

令人大跌眼镜地是，改动后的程序运行时间为 18.757s，慢了整整十倍还要多。

现在，我将数组的尺寸扩大 1，即变为 2049×2049，现在需要处理的数据更多，想象起来程序应该继续变慢，但是测量的结果是：4.644s，比上一个程序却快了 4 倍。

现在我们可以来分析一下这三个测量结果是怎么回事了，首先，按照行顺序来访问数组是一定比按照列顺序要快的，因为按照行顺序访问，实际上就是按照数据在内存中排列的顺序来访问，这是非常有利于 CPU cache 的工作的，而按照列顺序恰恰相反，每次回到同一行时本来应该期望元素存在于一级缓存中，但是由于访问了许多其它行的数据，当前元素可能已经被从缓存中清除掉了，这样效率自然会低下。

但是，这个因素事实上只会导致程序从 1.7s 慢到 4.6s，那么导致第二个测量结果 18s

的罪魁祸首是什么呢？答案是：数组的尺寸。我们本来使用的数组尺寸是 2048，它是一个 2 的方幂，我们从来就被教导说：使用 2 的方幂的数据更利于计算机工作，因为计算机是基于二进制工作的，想象起来在这里也应该如此，因为编译器为了做二维到一维的映射，需要做乘法，而数组的尺寸正是一个乘数，它是二的方幂，编译器就知道用位移运算来代替乘法操作，程序也理应更加高效。遗憾的是，这里的一点点效率提升在我们要面临的另一个巨大问题面前是可以忽略不计的。

回顾前面讲到 cpu cache 机制时的内容，当我们访问一个地址时，只有这个地址的末 10 来个 bit 被用来确定它在缓存中的位置，也就是说，如果地址递增的步长是一个二的方幂，那么就会造成频繁的 cache 冲突，在最严重的情况下，接连访问的不同位置的数据在 cache 中的组都是一样的，也就是说每次访问都会 cache miss，程序自然会像蜗牛爬一般运行了。

通过这个例子我们可以看出，尽管使用 2 的方幂数据在许多方面都会使程序更加高效，但在做多维数组的尺寸时恰恰相反，在定义多维数组时，我们不应该让任意一维的尺寸是 2 的方幂，最好所有的尺寸都是奇数。在编写基于状态压缩的动态规划程序时我们常常会违背这个原则，因为 dp 的状态中有一维总会是 2 的方幂，在这种时候我们可以适当把数组开大一点点，虽然不保证一定会取得优化，但养成良好的习惯终会受益。

应用举例

在竞赛中，往往我们只要设计出正确的算法，就能得到全部的分数，那这些常数优化的技巧有什么用武之地呢？仍然是有的，许多时候，我们并不能顺利找到最优的算法，只能退而求其次使用朴素的解决方案，这时，程序的运行效率就和细节上的实现密切相关了。我对众多 NOIP、NOI、IOI 的试题进行了广泛的测试，对于许多题目，使用论文中介绍的各种方法，可以比随意的实现多过 1 ～ 4 个测试点，几道题目积累起来，效果还是非常可观的，下面展示两个比较突出的实例。

【例一. 麦森数】

题目来源：NOIP2003 普及组

题目大意：求出 2^p-1 十进制值的末 k 位（p<=3100000,k=500）。

朴素算法：循环 p 次，每次向高精度数值上乘以 2，复杂度为 O(k×p)。

测试结果：通过 7 个测试数据。

优化方式：优化高精度乘法，消除其中的除法运算和条件分支。

优化效果：通过所有 10 个测试数据。

【例二. 瑰丽的华尔兹】

题目来源：NOI2005

题目大意：有一个 N×M（N,M<=200）的网格，某物体初始时处于网格中的(X,Y)位置，有一个长度为 T(T<=40000)的行动序列，序列中的每一个元素为东南西北中的一个方向，行动序列分为 S(S<=200)段，在每一段内的元素都是相同的。物体将会依照着这个序列每次向特定的方向移动一个单位。网格中某些位置有障碍物，物体在移动过程中不能碰到它们，且物体不能移出网格之外，问最少需要从序列中去掉多少个元素，物体的行动才能顺利进行。

朴素算法：朴素的动态规划，F[i][j][k]表示当前处于位置(j,k)，对于从第 i 段开始的行动序列，最少要去掉多少个元素才能让移动顺利进行。为了进行转移，需要枚举在第 i 段中删除的元素数 t，故总复杂度为 O(N×M×T)。

测试结果：通过 6 个测试数据。

优化方式：优化高维数组的寻址。

优化效果：通过全部 10 个测试数据。

【例三. 翻译玛雅著作】

题目来源：IOI2006

题目大意：给出两个串 W(长度 G<=3000)和 S(长度 N<=3000000)，求 W 的排列在 S 中出现了多少次，字符集大小为 A=52。

朴素算法：枚举 W 在 S 中的起始位置，现在希望知道在 S 中的这一段中各个字符各出现了多少次，如果和 W 中的情况一样则视为匹配，直接实现这个步骤需要 O(A)，总复杂度为 O(A*N)。

测试结果：通过 12 个测试数据。

优化方式：使用汇编优化循环与数组寻址。

优化效果：通过全部 16 个测试数据。

总结

讲了这么多细小的优化原则与技巧，看起来就像一盘散沙，毫无规律可言，那么在做优化工作时，我们有没有什么一般方法可以遵循呢？如果是说具体的某种“万金油”优化技术，答案是，没有。但在做优化时，有一些普适性的真理是值得我们去实践的，在本文的结尾，我与大家分享一下我所奉行的三句业内前辈留下的金玉良言，它们为我们的优化工作指明了正确的方向：

“过早的优化是万恶之源”

这句话因为 Donald Knuth(《The art of computer programming》的作者)而广为人知，而它所传达的精神在我们的竞赛中更是极为实用。从前文的优化实例中也很容易看出，优化是有代价的，而且代价还相当巨大，在程序设计方面，它很可能会使我们的程序的可读性下降、难于调试、难于维护，对于竞赛来说，就很可能会导致我们原本有能力实现或者能够调通的程序现在不再能够驾驭。另一方面，它会浪费编写代码者宝贵的时间，在业界甚至有一种观点，CPU 不要钱，程序员的时间宝贵，花费大量精力去做优化是得不偿失的。而在竞赛中，如果投入太多精力去做优化，可能会使我们没有时间去关注一些更有意义的事情。所以，在编写程序时，特别是在竞赛中，我们需要遵守一个非常重要的大原则：在编写最初版本的程序时，不要做任何会增加编码难度或思维难度的优化，在成功的编码并调试之后，我们拥有了一个可以正确运行的程序，这时在来着手考察是否需要做优化。之所以优化要到此时才可以开始进行，另外一个重要的原因是，在最初编写代码时，我们并不知道程序的瓶颈(bottleneck)在什么地方。如果一个函数执行总共占用的时间都只有 0.01 秒，我们即使将其优化了 100 倍，对整个程序运行时间的优化也是不可察觉的。事实上，在许多情况下，程序 99%的运行时间都在执行 1%的代码，这些代码往往就处于程序的某个最内层循环体中，去优化这个循环体以外的任何代码都会是徒劳的。在另一些情况下，也是在我们竞赛中经常遇到的，70%左右的时间都花在了 IO(输入输出)上，这时无论怎样去优化程序的核心算法都不会有多明显的效果。这些就是所谓的瓶颈，做优化，一定要先找到程序的瓶颈所在，在实际的程序项目中，往往需要使用一个被称为 profiler 的外部工具来辅助我们找到瓶颈所在，例如在 GNU 的开发套件中就有 gprof 工具可供使用，它可以帮我们分析程序中的各个函数在运行时占用了多少时间。在竞赛中，由于程序的规模较小，要找到瓶颈往往不是难事，主要是要有去寻找瓶颈并进行优化的意识，对于不是瓶颈的部分，大可以较为随意地实现。

“程序的优化是无止境的”

在找到程序的瓶颈所在之后，就可以开始着手进行优化了，永远不要对自己说“这段代码不能够再快了”。高级语言层面的逻辑调整、在高级语言层面配合编译器优化，或是在汇编语言层面配合 CPU 优化，在这个组合松散的体系结构下，到处都有深入挖掘的机会。就算在当前思路的基础上已经做到了极致，常常会发现只要换一个角度，又会海阔天空。

为了做出真正有效的优化，对编码者的知识与经验提出了较高的要求，需要我们对 CPU、汇编语言、编译器特性等都有一些广泛的了解。前文所讲的内容虽然粗浅，但从中也可以看出，做优化，到处都是规律，又到处都是特例，为了真正的掌握它们，需要时常亲自实践，成功地优化固然让人喜悦，失败的优化则能让人学到更多的东西。

这里需要再次强调的是，程序的优化决不是想当然的。诚然，程序的效率表现应该是合乎道理的，甚至是可计算的，但里面的机理过于复杂，再加上在不同的计算机平台上的行为也可能会大相径庭，所以唯一可以判断我们的优化有效的方式，就是亲自实现它，并获得效率上的提升。

“Keep it simple and stupid”

最后一句话，是著名的 KISS 原则，原本说来，做底层优化是最没有办法 KISS 的。做底层优化，我们在干的事情往往就是 make it more complicated and more confusing，和 KISS 原则背道而驰，也常常让编码者自己陷入泥潭之中无法脱身，要简洁还是要效率，这似乎是永远无法调和的矛盾。对此我的观点是，当我们把一个优化做得过于复杂时，往往已经走上了不正确的道路了，或许应该停下来想一想，或许还存在一种简洁得多而且更加高效的方法可以更好地解决这个问题。有时迫不得已，为了效率不得不将一些代码做得很不优雅，不过需要做这样极致优化的代码往往是很少的，我们只要精准地发现了效率的热点，只去集中优化这很少数的代码，剩下的大部分代码我们仍然可以以最简洁优雅的方式实现，局面就还能控制。

在这里本文就将要结束了，文中介绍的原则和技巧，多数都只是些皮毛，希望能起到一个抛砖引玉的作用，让大家看到程序编写中还有这样一个有趣的领域，并能在竞赛实战中发掘出它们更多地应用。

祝大家优化愉快。

参考文献

1. 《游戏之旅——我的编程感悟》
2. 《Professional Assembly Language》
3. 《Assembly Language for Intel-Based Computers - Fifth Edition》
4. 《Optimizing software in C++ - An optimization guide for Windows, Linux and Mac platforms》
5. 《Michael Abrash's Graphics Programming Black Book Special Edition》
6. 《Coding ASM - Intel Instruction Set Codes and Cycles》
7. 《Intel® 64 and IA-32 Architectures Software Developer's Manual》

8. 《Optimizing subroutines in assembly language - An optimization guide for x86 platforms》

特别感谢

感谢张君亮老师的辛勤指导
感谢冯国栋(bluemary)学长多年无私的帮助
感谢四川大学左洁教授对本文提出的宝贵意见
感谢所有在竞赛之路上帮助过我的战友与对手
以及无限支持与宽容我的父母及班主任老师