论现代硬件上的常数优化

成都外国语学校 宋佳兴

摘 要

本文深入探讨了单线程算法的优化思想和方法。鉴于上一篇关于同一主题的集训队论文已经是十年前的了，本文的内容与当前的硬件环境更为贴近。本文涵盖了 x86 架构与 Coffee Lake 微架构简介、通用优化方法、计算密集型算法、数据结构等多个板块，从汇编代码、硬件特性等底层概念出发，对常数优化进行了定性和定量的研究。

前言

随着计算机硬件的发展，渐近复杂性已经不再占据绝对的地位，相同的时间复杂度可能会产生很大的速度差异。此外，硬件的并行能力在不断增强，高速缓存和主存之间的速度差距也在扩大，因此底层优化的提升空间越来越大。很多时候，底层优化可以弥补算法上的先天不足，从而实际提升竞赛成绩。在竞赛之外，我们可以通过这种方式更深入地理解计算机，提高代码质量。

在竞赛中 SIMD 的使用是不允许的，所以本文并未涉及到 SIMD。然而，这些优化思想和方法仍然可以应用于 SIMD 的环境中。

我的电脑与 OI 赛场的评测机 i7-8700k 具有相同的微架构，因此在相同的主频下，它们的速度接近。程序的运行环境如下：

CPU 微架构	Coffee Lake
时钟频率	4.4GHZ 不变
主存	32GB 2933MHz
操作系统	Linux
编译器	GCC 9.5
编译选项	-O2

1 x86 架构与 Coffee Lake 微架构

指令集架构 (Instruction Set Architecture) 定义了机器语言的含义，它是硬件执行机器代码的基础。其中，x86-64 是最广泛使用的指令集架构，它的存在使得各种不同的硬件设备能够理解和执行相同的机器代码。

另一方面，微架构 (Microarchitecture) 定义了硬件执行机器指令的具体细节，它决定了硬件的性能。每一种微架构都有其独特的设计和优化，这些因素共同决定了处理器的性能。

Coffee Lake 是 Intel i7-8700k 处理器的微架构，我们选择它作为研究对象。

1.1 汇编语言

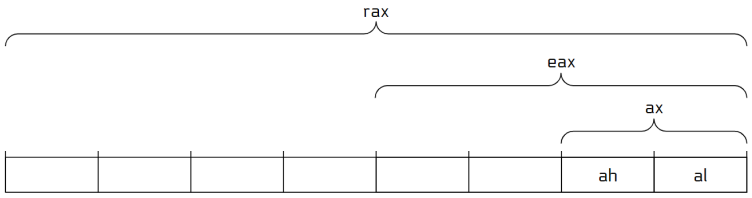
学习汇编的目的并不是为了手动编写它，因为在 O2 优化下，手写汇编的提升空间相当有限。我们的真正目标是能够理解编译器生成的汇编代码，知道程序运行的原始操作是什么，以及优化的可能性在哪里。一旦熟悉了汇编，我们就可以预想出 C++ 代码在编译后的样子。

汇编语言 (Assembly Language) 和机器代码几乎是一一对应的，可以被视为程序运行的原始操作。现在，让我们进入正题。

1.1.1 寄存器

寄存器 (Register) 是计算机中的一种重要存储单元，它们用于存储和操作数据。在汇编语言中，寄存器可以被视为具有名称的变量。通用寄存器有 16 个：rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15。

通用寄存器都是 64 位无符号整数，为了支持 32,16,8 位的运算，每个寄存器的低 32,16,8 位和 8~15 位都可以当成一个完整寄存器用，如图所示。



`mov` 指令用于将一个值（可以是常量、寄存器中的值或内存中的值）复制到寄存器中。这是汇编语言中最常用的指令之一：

```
mov   rax, 0×3f3f3f3f
mov   rbx, 114514
mov   al, 114
mov   rcx, rax # let rcx = rax
mov   edx, -1 # let edx = 0×ffffffff
```

大多数算术运算的指令都很简单，如下所示。

汇编	C++
`add rax, rbx`	`rax += rbx`
`sub ebx, ecx`	`ebx -= ecx`
`imul rsi, rdi`	`rsi *= rdi`
`neg rax`	`rax = -rax`
`not rax`	`rax = ~rax`
`and rax, rbx`	`rax &= rbx`
`or rax, rbx`	`rax |= rbx`
`xor rax, rbx`	`rax ^= rbx`
`shl rax, 10`	`rax <<= 10`
`shr rax, 10`	`rax = (unsigned long long)rax >> 10`
`sar rax, 10`	`rax = (long long)rax >> 10`

1.1.2 地址解引用

解引用是一种操作，它允许我们访问存储在特定地址上的值。在 C++ 中，数组访问就是一种解引用的例子。

最简单的解引用 `rbx = *(int*)rax` 所对应的汇编代码为：

```
mov rbx, DWORD PTR [rax]
```

解引用地址的一般的用法为：

```
SIZE PTR [base + index * scale + displacement]
```

base 和 index 都是 64 位寄存器，scale 和 displacement 都是常量，其中 scale 只能为 1,2,4,8 之一，每个部分都可以省略。这种格式的设计是为了提供灵活性，减少计算地址的开销。

`SIZE PTR` 描述解引用的数据大小，有以下选择：

`BYTE PTR`	8 位
`WORD PTR`	16 位
`DWORD PTR`	32 位
`QWORD PTR`	64 位

解引用不是只能用在 mov 指令中，请看下面的例子：有一个 int 数组首地址为 rax，把下标为 rbx 的元素加上 ecx，汇编如下：

```
add DWORD PTR [rax + rbx * 4], ecx
```

1.1.3 跳转

在汇编语言中，我们使用跳转指令来实现条件分支和循环。这些指令允许我们根据特定的条件改变程序的执行流程。来看一下这段 C++ 求和代码编译后的结果：

```
int sum(int* begin, int* end) {
    int res = 0;
    while (begin != end) res += *begin++;
    return res;
}
```

汇编中函数有一个规定：每个参数依次存放在 rdi, rsi, rdx, rcx, ... 中，返回值存进 rax，使用 ret 指令返回[1]。以下为 sum 函数的汇编：

```
sum:
    xor   eax, eax
    cmp   rdi, rsi
    je    end
loop:
    add   eax, DWORD PTR [rdi]
    add   rdi, 4
    cmp   rsi, rdi
    jne   loop
    ret
end:
    ret
```

在这段汇编中，`je end` 是条件跳转指令，它允许我们在两个值相等时跳转到程序的另一部分。cmp 指令用于比较两个值，并将结果存储在 FLAGS 寄存器中，然后，我们可以使用条件跳转指令（如 je）来根据比较的结果改变程序的执行流程。

1.1.4 获得编译结果

工具 compiler explorer[2] 可以人性化地显示编译器生成的汇编。不了解的汇编指令可以在参考资料 [6] 中查询。

1.2 指令流水线

指令流水线 (Instruction Pipelining) 是一种技术，它使 CPU 能够同时执行多条指令。这是通过将每条指令的执行过程分解为几个阶段来实现的，每个阶段都在同时处理多条指令。

执行任何机器代码都需要经过以下阶段：

[1]函数调用约定 https://stackoverflow.com/questions/18024672

[2]https://godbolt.org

1. 从内存中获取一大块机器代码。
2. 在大块代码中分离出每条指令，并将其转换为微指令 (Micro-operation)。
3. 微指令进行寄存器重命名 (Register Renaming / Allocation)。
4. 微指令进入调度器 (Scheduler, Reservation Station), 输入操作数已经可用的微指令会被优先发送到执行端口。
5. 微指令进入执行单元进行执行。
6. 结果被临时存储，后续的微指令可以直接获取所需的数据。
7. 所有微指令按照最初的顺序，将结果写回寄存器。

前三个阶段被称为 CPU 的前端，后四个阶段被称为 CPU 的后端。CPU 前端的速度相对固定，而后端通常有提升空间，只有当指令以合适的顺序到达时，CPU 后端才能忙碌起来。

影响流水线速度的限制因素主要有三类：

- 结构风险 (Structural Hazards)：同一类型的指令过多，相应的执行单元不足。
- 数据风险 (Data Hazards)：当前指令需要等待上一条指令的结果才能开始执行。
- 控制流风险 (Control Hazards)：遇到跳转指令时，CPU 不能暂停等待结果，否则会导致流水线停顿。因此 CPU 会使用分支预测器 (Branch Predictor) 预测接下来的走向，并提前将指令放入流水线。如果分支预测未命中，就需要清空流水线。

控制流风险是最严重的，我们用一个例子来说明：

```
int foobar(bool a[100000]) {
    int cnt0 = 0, cnt1 = 0;
    for (int i = 0; i < 100000; i++)
        if (a[i] == 0) cnt0++;
        else cnt1++;
    return cnt0 ^ cnt1;
}
```

- 如果 $a_i = i \bmod 2$，那么分支预测器可以发现 if 语句是 01 交替的，此时不会发生控制流风险，foobar 函数会运行 34 µs。
- 如果 a_i 从 $\{0, 1\}$ 中随机选取，那么分支预测器就无能为力了，此时会发生控制流风险，foobar 函数会运行 310 µs。

数据风险通常发生在指令形成依赖链时，例如，一条指令的结果被用作下一条指令的输入：

```
int v0 = 1, v1 = 2, v2 = 3, v3 = 4;
for (int i = 0; i < 1000000000; i++)
    v1 += v0, v2 += v1, v3 += v2, v0 += v3;
```

这里第 3 行的四条指令就必须严格按顺序执行，而且 v0 += v3 执行完了下一轮的 v1 += v0 才能开始，因此 CPU 根本无法并行。如果把第 3 行改为：

```
v0 += v0, v1 += v1, v2 += v2, v3 += v3;
```

那么数据风险就几乎没有了，速度是原来的 2.9 倍，此时问题变成了结构风险，可以通过循环展开解决，详见 2.1。

1.3 Coffee Lake 微架构

x86 指令集极其庞大，导致制作工艺复杂。相比之下，RISC（精简指令集计算机）采用了一个更小但功能齐全的指令集，这大大简化了处理器的设计，从而提高了性能。然而，x86 指令集不能舍弃对旧软件的兼容性。为了在保证兼容性的同时提高性能，处理器内部采用了一个精简的微指令集。CPU 的前端部分负责将内存中的机器代码翻译成微指令 (Micro-operation, 简称 µop)，然后提供给后端执行。这样，处理器制造商就可以通过优化 CPU 的前后端来提高性能。

微架构定义了 x86 指令如何被翻译成微指令（以下简称 µop），这些 µop 如何使用执行端口，以及执行所需的时间。OI 赛场的评测机 i7-8700k 采用了 Coffee Lake 微架构，因此，本文的所有性能分析都将基于 Coffee Lake 进行。

1.3.1 延迟与吞吐量

首先，我们需要理解 x86 指令的速度并不是一个可以用单一数字表示的概念。在流水线环境中，速度与并行程度有关。然而，我们可以通过考虑两个极端情况来理解一条指令的速度：

- 延迟 (Latency)：这是一条指令从开始执行到得到结果所需的时间。在完全串行的情况下，它可以被理解为每条指令的执行时间。
- 吞吐量倒数 (Reciprocal Throughput)：在同一类型的指令完全并行执行的情况下，它是每条指令的平均执行时间。

这里的时间是以时钟周期 (clock cycle) 数量来衡量的，时钟周期是 CPU 活动的一个周期。通常所说的主频就是每秒能进行多少个时钟周期。在后文中，我们将使用平均每次操作的时钟周期数 cycles 来描述一种操作的常数因子。例如，平均每次操作 0.5 cycle 意味着这种操作每秒大约能执行 9×10^9 次。测量时钟周期的方法在 6 中。

在 Coffee Lake 微架构上，一些指令的延迟和吞吐量倒数信息如下：

指令	操作数	延迟	吞吐量倒数	备注
`add, sub`	`r, r/i`	1	0.25	r 表示 8~64 bit 的任何寄存器，i 表示常量 (immediate value)
`and, or, xor`	`r, r/i`	1	0.25	加法和减法与位运算相比没有差距，都是每个周期可以并行执行 4 条指令
`shl`	`r, i`	1	0.5	

1.3.2 微指令与执行端口

每条 x86 指令等价于一条或多条微指令 (μop)，每条 μop 会进入一个特定的执行端口 (Execution Port)。执行端口和执行单元的区别在于，执行端口内部包含了一个小型流水线，这个流水线又包含了若干个执行单元。因此，即使上一个 μop 还在执行，每个周期都可以有一个新的 μop 进入同一个执行端口。

在 Coffee Lake 微架构上，共有 8 个执行端口，分别命名为 p0, p1, …, p7。其中，p0156 都包含算术逻辑单元 (ALU)，都可以完成 add 和 sub 指令。但是，它们的其他功能却略有不同。p23 是加载端口，用于将内存中的值加载到寄存器中；p4 是存储端口，用于将值存储到内存中；而 p7 则是一个功能较弱的辅助端口。

在 x86 指令到 μop 的翻译过程中，还存在一种称为融合微指令（fused-μop）的概念。fused-μop 和 μop 的区别在于，某些 fused-μop 实际上是 μop 的二元组。x86 指令首先会被翻译成 fused-μop，在确定了执行端口后，融合的 μop 二元组才会被分开，分别进入不同的执行端口。fused-μop 的存在使得 CPU 的前端可以更快地向后端提供 μop。

一些指令的 fused-μop、μop 和端口需求如下：

指令	操作数	吞吐量倒数	**fused-μop/μop**	端口需求
`add, sub`	`r, r/i`	0.25	1/1	p0156（任选一个）
`add`	`r, [address]`	0.5	1/2	p23 + p0156
`mov`	`[address], r`	1	1/2	p237 + p4

可以发现吞吐量倒数可以由端口需求算出来，比如端口需求为 p23 + p0156，那么一个周期最多能执行两条指令，不然 p23 就不够用了。因此通过端口需求可以定量地分析指令。

CPU 前端每个周期最多提供 4 条 fused-μop，因此 fused-μop 的数量除以 4 是一个必须的时间。CPU 前端想要跑满 4 fused-μops/cycle （4 fused-μops 每周期）很简单，fused-μops 是有缓存单元的，名叫 DSB (Decode Stream Buffer)，可以记忆最近执行过的 fused-μops，只要它能工作，就不需要从内存中读取 x86 指令并翻译成 fused-μop 了。一个不长也不短的循

环 (8~1000 fused-µops) 迭代很多次大约就是 4 fused-µops/cycle，里面不能有一些极其复杂的指令（比如除法），也不能发生跳转（可以包含条件跳转但不能真的跳转）。

最后，x86 指令信息可以在参考资料 [13] 和 [14] 中查询。查询和流水线模拟工具的详细信息，请参见6。

1.3.3 寄存器重命名

在 Coffee Lake 微架构中，有 180 个物理通用寄存器，调度器 (Scheduler, Reservation Station) 的容量为 97，调度器会把输入操作数已经可用的 fused-µops 发送到执行端口。然而，在汇编中只有 16 个通用寄存器，这似乎使得大量的物理寄存器和调度器容量被浪费。

实际上，处理器会通过寄存器重命名 (Register Renaming / Allocation) 来充分利用物理寄存器。每个汇编中的逻辑寄存器都对应一个物理寄存器，这种对应关系可以动态变化。请看下面的例子：

```
mov eax, DWORD PTR [rdi]
imul eax, 2023
mov DWORD PTR [rdi], eax
mov eax, DWORD PTR [rsi]
imul eax, 2023
mov DWORD PTR [rsi], eax
```

这段代码在做两件无关的事情，但是因为它们共用 eax，似乎就只能顺序执行。然而，实际上，eax 所指代的物理寄存器发生了变化，这两件事情使用了不同的物理寄存器，因此可以并行。

寄存器重命名的工作原理非常简单：每当逻辑寄存器被写入或修改时，它就会指代一个新的物理寄存器，这有点像可持久化数据结构。因此，`mov eax, DWORD PTR [rsi]` 的执行并不会依赖于前面的任何指令。

除了上面的例子，寄存器重命名还有其他的好处：

- ```
  add DWORD PTR [rdi], eax
  add eax, 1
  ```

  在这里，`add eax, 1` 不必等待上一条指令执行完毕，因为 add 只会把结果写入一个新的物理寄存器，不会修改原来的物理寄存器。

- 一个每次迭代都无关的循环可以具有并行多轮迭代的能力，因为使用同一个寄存器不会产生依赖，而且调度器容量很大。一个特别的例子是线段树中一次操作还没有进行完就可以同时开始下一次操作。

- 可以利用寄存器重命名把 `mov rax, rbx` 消除掉，只需要让 rax 指代 rbx 所指代的物理寄存器。但是，这仍然会占用 CPU 前端的带宽。
  ```

指令	操作数	延迟	吞吐量倒数	fused-µop/µop	端口需求	备注
mov	r32/64, r32/64	0	0.25	1/0		不指代同一物理寄存器
mov	r32/64, r32/64	1	0.25	1/1	p0156	已经是同一物理寄存器

1.4 高速缓存与内存

程序使用的临时存储包含若干层，每一层有不同的大小和速度。越往上，访问速度越快，但大小也越小。我的电脑上各层的大小如下（和 OI 赛场的评测机是一样的）：

- L1 缓存 (L1 cache) 由指令缓存 (L1i) 和数据缓存 (L1d) 组成，每核的 L1i 和 L1d 都是 32KB。
- L2 缓存 (L2 cache)，每核 256 KB，不分数据和指令。
- L3 缓存 (L3 cache)，6 核共享 12 MB。
- 主存 (RAM) 32 GB。

当数据被访问时，它所在的缓存行会被移入 L1 缓存，在长时间不访问后依次退回到 L2, L3 和主存。这种设计使得访问越频繁的数据平均开销越小，因为越频繁访问的数据更可能在高速缓存中。

1.4.1 缓存行

缓存与主存间数据传输的基本单位不是字节，而是缓存行 (Cache Line)。在 x86 架构下，缓存行是首地址为 64 倍数的 64 字节段，缓存里存的都是若干个缓存行。这意味着，如果请求的数据不在 L1 缓存时，就需要将整个缓存行移入 L1 缓存。

在瓶颈为内存的程序中，缓存行未命中的数量是重要的信息，如何测量它请参见6。

1.4.2 延迟和吞吐量

和 x86 指令的延迟和吞吐量倒数一样，延迟是一个请求从开始到完成的时间，吞吐量倒数是并行情况下的一个请求的平均时间。

在我的电脑上测量结果如下（仅供参考）：

时钟周期	延迟	吞吐量倒数（读）	吞吐量倒数（写）
L1	5	0.5	1
L2	12	1.7	3.6
L3	41	3.95	6.55
主存	240	12.4	20.8

可以发现，写入的吞吐量几乎总是读入的两倍。在 L1 上，这是执行端口的缘故。除此之外，都是缓存的写入策略导致的：如果写入请求未命中缓存，那么先将数据移动到缓存，然后仅修改缓存。这会导致各级缓存和主存中的数据不一致，因此缓存中每个缓存行都带有一个 dirty bit 记录它是否修改过，当一个 dirty bit 为 1 的缓存行被替换出去的时候，需要将它写回比它大一级的缓存或者主存中去，写入请求比读取请求会多做一件事：把 dirty bit 设置为 1，这在将来会触发一次写入。

写入策略带来的好处是：无论先读再写还是先写再读，都只有第一次未命中。

1.4.3 替换策略

请求在缓存未命中时，需要选择一个原来的缓存行，用新的缓存行替换它，这里怎么选择决定了缓存的性能。

最流行的选择是最近最少使用策略 (Least recently used, 简称 LRU)，它相当于可用缓存减半时的最优策略，有如下定理[3]：

定理. 设 OPT_M 表示可用缓存为 M 个缓存行时，理论最优解会有多少次缓存行未命中，LRU_M 表示 LRU 会有多少次缓存行未命中，那么：

$$LRU_M \leq 2\dot{O}PT_{M/2} \tag{1}$$

但事实上，硬件并没有真正的实现 LRU，而是分组关联 + PLRU，以 L1 缓存为例：

- L1 数据缓存有 32 KB，因此有 512 个缓存行。
- 这 512 个缓存行被分成 64 组，每组包含 8 个缓存行。
- 每组的 8 个缓存行使用 Tree-PLRU，一种近似的 LRU，见参考资料 [9]

如何确定一个地址属于哪个组呢？理想的方式是用哈希函数把地址映射到缓存组，但是计算哈希太慢了，硬件中无法接受。64 位地址被分成 3 个部分，tag, index 和 offset，见下图，index 决定了这个缓存应该放在哪一个组中，tag 则用于在同一个组中区分地址。

[3]来源 https://en.algorithmica.org/hpc/external-memory/policies/

tag	index	offset
[12,63]	[6,11]	[0,5]

没有哈希函数就容易受到不良访问模式的影响：当地址以 2^k 为步长递增时，只有一小部分的组可用，比如访问的地址为 $0, 256, 512, \cdots$ 那么它们只会映射到 $\frac{1}{4}$ 的组。

在 OI 中会遇到的情况是：二维数组第二维是 2^k，比如状压 DP，固定第二维下标不变，第一维下标变化时，地址就是以 2^k 为步长递增，导致缓存不能有效工作。解决的办法是把 2^k 这一维开得大一点，比如多开 256 Byte，这样可以避免地址冲突，使得缓存可以更有效地工作。

1.4.4 虚拟内存

虚拟内存 (Virtual memory) 是一种内存管理技术，它使得程序中的一片连续地址在物理内存上无需连续，从而更有效地利用物理内存中的碎片空间。然而，这种技术也会对性能产生一定的影响。具体来说，内存被划分为大小为 4KB 的页 (Page)，每一个虚拟地址中的页都会映射到物理内存中的一个页。因此，在解引用 *address* 时，我们需要先将 $\lfloor address/4096 \rfloor$ 映射到物理地址，然后再在缓存和 RAM 中进行查找。转译后备缓冲器 (TLB) 是一个用于加速这个映射过程的缓存系统。如果 TLB 未命中，那么就需要从主存中查找对应的物理地址，这个过程的速度很慢。

有时地址转换可能会成为瓶颈：例如，存在一些分散的缓存行，它们的总量适合放入高速缓存，但是由于 $\lfloor address/4096 \rfloor$ 的值各不相同，导致 TLB 无法存储所有的映射关系。

1.4.5 内存优化原则

- 空间局部性：把一起使用的数据在内存中安排在一起，使得它们位于一块或者两块缓存行中。因此，数据结构结点信息的首选是结构体，而不是多个数组。
- 时间局部性：让数据在较小时间段内重复访问。典型的例子是滚动数组和矩阵分块乘法。
- 内存级并行：如果提前知道所有的内存地址，内存请求就可以并行。不能内存级并行的例子是链表。
- 读取快于写入：例子是 DP 的填表法比刷表法快。
- 避免有害现象：地址以 2^k 为步长递增导致缓存大部分不可用，以及访问太分散导致 TLB 未命中。

2 通用优化方法

2.1 循环展开技术

循环展开 (Loop Unrolling) 是一种提升程序性能的简单而有效的方法。

2.1.1 int 类型求和

考虑以下的 C++ 代码：

```
for (int i = 0; i < 2000; i++) sum += a[i];
```

编译后的汇编代码如下：

```
loop:
    add   eax, DWORD PTR [rdi]
    add   rdi, 4
    cmp   rdi, rdx
    jne   loop
```

这个循环每次迭代至少需要 1 个 cycle，主要有三个瓶颈：

- 前两条 add 指令都依赖于上一轮的同一条语句。
- cmp 和 jne 两条语句只生成了一条 µop（因为这个组合太常见了，所以它被设计出来提高性能），但是这个 µop 只能进入 p6 端口，因此每周期只能执行一条。
- 因为每个周期都要跳转，所以 CPU 前端也只能达到 3 fused-µops/cycle 的速度。

虽然这三个瓶颈都使得每次迭代至少需要 1 cycle，但实际情况是 CPU 无法实现最优的调度，所以每次迭代实际上需要 1.4 cycles。

理想的循环长度为 8 到 1000 个 fused-µops，这样 CPU 前端可以达到 4 fused-µops/cycle 的速度，同时也减少了循环变量和跳转的开销。我们可以通过循环展开来实现这个目标：

```
for (int i = 0; i < 2000; i += 8)
    sum += a[i], sum += a[i + 1], ... , sum += a[i + 7];
```

展开后重新编译，得到的汇编代码如下：

```
loop:
    add   eax, DWORD PTR [rdx]
    add   rdx, 32
    add   eax, DWORD PTR [rdx-28]
    add   eax, DWORD PTR [rdx-24]
     ...
    add   eax, DWORD PTR [rdx-4]
    cmp   rcx, rdx
    jne   loop
```

这时，只剩下一个瓶颈：`add eax, [ ... ]` 指令构成了依赖链。实际的平均时间达到了 1 cycle，刚好卡在这个瓶颈上。

最后一步是消除这个依赖链：

```
int s0 = 0, s1 = 0, s2 = 0, s3 = 0;
for (int i = 0; i < 2000; i += 8) {
    s0 += a[i + 0], s1 += a[i + 1], s2 += a[i + 2], s3 += a[i + 3];
    s0 += a[i + 4], s1 += a[i + 5], s2 += a[i + 6], s3 += a[i + 7];
}
int sum = s0 + s1 + s2 + s3;
```

现在，依赖链被打破了，唯一的瓶颈在 p23 端口，实际平均时间达到了 0.5 cycles。虽然只使用两个求和变量也能达到这个速度，但依赖链刚好也成了一个 0.5 cycles 的瓶颈，同时存在两个瓶颈会造成性能损失。

2.1.2 对 int 类型整体加上一个数

考虑以下的 C++ 代码：

```
for (int i = 0; i < 2000; i++) a[i] += x;
```

编译后的汇编代码如下：

```
loop:
    add   DWORD PTR [rdi], esi
    add   rdi, 4
    cmp   rdi, rax
    jne   loop
```

每次迭代的平均时间为 1.34 cycles，因为它有多个 1 cycle 的瓶颈，无法平衡这些瓶颈就会 > 1 cycle。具体瓶颈如下：

- 超短循环本身的缺点：循环变量加法的依赖、跳转、CPU 前端。
- `add DWORD PTR [rdi], esi` 每周期只能执行一条，因为它有一个 μop 只能进入 p4 端口。

指令	操作数	吞吐量倒数	**fused-μop/μop**	端口需求
`add`	`[address], r32/i32`	1	2/4	p23 + p0156 + p237 + p4

循环展开 4 步就足够了，因为 fused-μops 达到了 8。然后就只受限于 `add [address], r32` 的吞吐量，从而达到 1 cycle 的平均时间。

```
for (int i = 0; i < 2000; i += 4)
    a[i] += x, a[i + 1] += x, ... , a[i + 3] += x;
```

2.1.3 计算 int 类型的前缀和

考虑以下的 C++ 代码：

```
for (int i = 0; i < 2000 - 1; i++) a[i + 1] += a[i];
```

这里有一个冷知识，（当 a 是数组名而非指针名时）前缀和写 `a[i + 1] += a[i]` 比 `a[i] += a[i - 1]` 更好，因为 O2 对后者的优化是不到位的。这个问题在 GCC 12 中得到了修复，但在 NOI-Linux 中的 GCC 9 是未修复的。另外，O2 优化对 `partial_sum` 的优化是到位的，因此可以放心使用。

对于 fused-µops 不足 8 的超短循环，循环展开肯定有效果，但实际上没有，原来是编译器没有生成预期的汇编代码。

换了一种写法后，就达到了 1 cycle 的平均时间。

```
int sum = 0;
for (int i = 0; i < 2000; i += 4) {
    sum += a[i + 0], a[i + 0] = sum;
    ...
    sum += a[i + 3], a[i + 3] = sum;
}
```

解释一下为什么速度能达到 1 cycle，`sum += a[ ... ], a[ ... ] = sum` 会生成如下汇编：

指令	fused-µop/µop	端口需求
`add eax, DWORD PTR [ ... ]`	1/2	p23 + p0156
`mov DWORD PTR [ ... ], eax`	1/2	p237 + p4

这两条语句可以分成三个部分：加载值 (p2)，加法 (p0156)，存储值 (p237 + p4)，只有加法的阶段构成了依赖链，但是执行加法只需要一个周期，而另外两个阶段都是完全并行的，执行端口刚好够用。

2.1.4 自动展开功能

手动循环展开可能会比较麻烦，因为复制粘贴后还需要修改下标。这里，我将介绍两种循环展开的简写方法。

需要注意的是，循环展开有一定的副作用，它会增大机器代码的长度，从而影响 CPU 前端获取机器代码的速度。因此，我们不建议使用编译选项进行全局展开，而是只对程序中的瓶颈循环进行展开。

2.1.5 使用 pragma 命令

在想要展开的 for 循环前面加上一句 `#pragma GCC unroll n`，其中 n 是一个数字，表示展开循环的步长。示例如下：

```
#pragma GCC unroll 4
for (int i = 0; i < 2000; i++) a[i] += x;
```

这段代码相当于：

```
for (int i = 0; i < 2000; i += 4)
    a[i] += x, ..., a[i + 3] += x;
```

特别地，当循环次数为常数时，可以彻底消除循环：

```
#pragma GCC unroll 10
for (int i = 0; i < 10; i++) a[i] += x;
```

2.1.6 使用 C++ 模板函数

由于竞赛通常会禁用 pragma 命令，所以我们还有第二种方法，即定义一个模板函数来辅助完全展开：

```
template <int n, int k = 0, class F>
void UL(int i, F lambda) {
    if (k < n) lambda(i + k), UL<n, k + (k < n)>(i, lambda);
}
```

使用方法如下：

```
for (int i = 0; i < 2000; i += 4)
    UL<4>(i, [&](int i) { a[i] += x; });
```

不必担心 UL 函数和匿名函数的调用开销，因为编译器足够聪明。

2.2 循环融合

将需要访问同一数组的循环融合在一起是一种简单而有效的优化方法，其目标是减少内存读写次数。

例如，如果我们需要对数组 a 进行两次前缀和计算，常规的写法可能如下：

```
int sum0 = 0;
for (int i = 0; i < 2000; i++)
    sum0 += a[i], a[i] = sum0;
int sum1 = 0;
for (int i = 0; i < 2000; i++)
    sum1 += a[i], a[i] = sum1;
```

根据上一节的内容，如果我们进行循环展开，`sum0 += a[i], a[i] = sum0` 的平均时间为 1 cycle，瓶颈在依赖性和内存写入操作。

然而，如果我们将这两个循环融合在一起，写法如下：

```
int sum0 = 0, sum1 = 0;
for (int i = 0; i < 2000; i++)
    sum0 += a[i], sum1 += sum0, a[i] = sum1;
```

这样就减少了一半的内存读写次数。如果再加上循环展开，平均每次 `sum0 += a[i], sum1 += sum0, a[i] = sum1` 就只需要 1.06 cycles，比融合前快了近一倍。

```
int sum0 = 0, sum1 = 0;
#pragma GCC unroll 32
for (int i = 0; i < 2000; i++)
    sum0 += a[i], sum1 += sum0, a[i] = sum1;
```

这里我们要展开 32 步，因为 CPU 在实现最优调度时可能会遇到困难，通过增大步数可以使平均时间更接近 1 cycle。

在 OI 中，循环融合可以应用于动态规划和数据结构。例如，当我们有以下 DP 转移方程时：

$$dp_{i,j} = \min\{dp_{i-1,j-1} + w_1, dp_{i-1,j-2} + w_2, dp_{i-1,j-3} + w_3\} \tag{2}$$

一个较快的写法是单独处理 $dp_{i,0}, dp_{i,1}, dp_{i,2}$，然后使用一个循环处理后面的。在这种情况下，循环展开可以带来好处，因为每个 $dp_{i-1,j}$ 都会被连续使用 3 次，没有必要多次读取内存，循环展开可以帮助编译器更好地做这一优化。

2.3 去除分支

分支是指跳转指令。分支预测机制相当智能，当分支模式的循环节不超过 200 时，预测成功率几乎可以达到 100%。此外，嵌套循环和 switch 语句也能被成功预测。在预测成功的情况下，如果跳转条件为真，需要 1~2 cycles，而如果为假，平均需要 0.5 cycles。

对于总是执行或者不执行的 if 语句，我们可以大胆地使用它们。由于跳转比不跳转需要更多的时间，我们可以使用 `__builtin_expect` 函数来告诉编译器 if 语句的执行倾向，以便进行优化：

```
if (__builtin_expect(x < y, 0)) x = y; // 几乎总是不执行
if (__builtin_expect(x > y, 1)) x = y; // 几乎总是执行
```

分支预测器发现循环节的能力是可以被利用的，比如非递归线段树每次操作的循环次数都是固定的，就可以精准预测。

然而，分支预测未命中的情况也很常见，大部分算法中的 if 语句都是不可预测的，此时需要花费 15~20 cycles。因此，如果 if 语句后面的语句很少，执行的 cycles 不多，那么我们应该考虑去除这些分支。

2.3.1 cmov 指令

请看下面的代码：

```
for (int i = 0; i < 100; i++) a[i] = max(a[i], b[i]);
```

这段代码在编译后并没有生成跳转指令，而是使用了 cmovge 指令，含义是只有在大于等于的情况下才会进行赋值。对应的汇编代码如下：

```
mov     edx, [address of ai]
cmp     [address of bi], edx
cmovge  edx, [address of bi]
mov     [address of ai], edx
```

如果我们换一种写法：

```
void checkMax(int& a, int b) { if (a < b) a = b; }
for (int i = 0; i < 100; i++) checkMax(a[i], b[i]);
```

那么编译器就会生成跳转指令。因此，如果我们想让编译器生成 cmov 指令，就需要注意代码的编写方式。以下是几种编译器通常会生成 cmov 指令的写法：

- 定义两个变量 a, b，计算好值后接一个和 cmov 功能相同的 if 语句：

  ```
  int a = ..., b = ...;
  if (a < b) a = b;
  ```

- 定义两个变量 a, b，计算好值后使用三目运算符：

  ```
  int a = ..., b = ...;
  int c = a > b ? a : b;
  ```

- 使用 min/max 函数。

cmov 系列指令有很多，大多数的性能相同，但有两个例外：

指令	操作数	延迟	吞吐量倒数	端口需求
大部分 cmov	r32/64, r32/64	1	0.5	p06
cmova, cmovbe	r32/64, r32/64	1	1	2*p06

在 C++ 中，无符号的比较可能会生成较慢的 cmova, cmovbe，也可能生成较快的 cmovae, cmovb，这取决于写法，可以通过检查汇编代码来确认。从 GCC 12 开始，就不会再生成较慢的指令了。

2.3.2 位运算

- && 和 || 会生成分支，如果这个分支是不可预测的，可以替换为位运算 &, |。
- 将一个数和 0/-1 进行 and 运算可以实现筛选，例如：

```
int mask = ... ? -1 : 0;
a[0] = max(a[0], b[0] & mask); // mask = -1 时才会 checkMax
a[1] = max(a[1], b[1] & mask);
```

- 使用 `__builtin_ctz` 遍历 mask 中的所有 1，可以用来优化状压 DP。

```
while (mask) {
    int i = __builtin_ctz(mask);
    ...
    mask &= mask - 1;
}
```

- 另一个位运算替代分支的技巧，见参考资料 [7]。

2.3.3 比较运算符

仅仅使用比较运算符而没有 if 是不会生成分支的，比如 `a = x < y` 会生成 set 系列指令。set 指令信息和 cmov 完全相同，也是 seta 和 setbe 较慢。

特别地，无符号的 `a + (c < d)` 和 `a - (c < d)` 还可以生成较快的 adc 和 sbb 指令。

2.4 使用 long long 替代 pair

由于 64 位的指令通常和 32 位一样快，因此使用 `long long` 替代 `pair<int, int>` 来执行某些操作会更快：

- 加载或存储一个 `long long` 比加载或存储两个 `int` 快。
- `long long` 的比较等价于 `pair<int, uint32_t>` 的比较，而 `uint64_t` 则等价于 `pair<uint32_t, uint32_t>`，这样可以去除分支。另外，`__int128` 的比较也是无分支的。

  这在数据结构维护最小值的位置（如 dijkstra）和 DP 记录方案时非常有用。
- `long long` 的加法可以模拟两次 `int` 的加法，前提是没有负数且没有溢出。例如：

```
c[i][0] = c[i * 2][0] + c[i * 2 + 1][0];
c[i][1] = c[i * 2][1] + c[i * 2 + 1][1];
```

  可以替换为

```
*((long long*)c[i]) = *((long long*)c[i * 2]) + *((long long*)c[i * 2 + 1]);
```

  这还可以用于优化无负数无溢出的求和场景，从而获得两倍的速度。

在强制以 `long long` 解引用时，地址最好是 8 的倍数，否则数据可能跨越两个缓存行，这样会使速度慢一倍。可以使用 alignas 说明符[4]定义对齐的数据。

2.5 提高内存访问效率

2.5.1 空间局部性

空间局部性是一种定性的优化方法，其目标是让访问的数据尽可能在空间上连续，从而减少缓存行未命中的次数。

为了实现空间局部性，有两个需要做出选择的地方：

- 内存布局：对于高维数组，哪一维应该在前面？一组相关的数据应该用多个数组存储，还是用一个结构体数组存储？
- 访问顺序：当有多个 for 循环时，需要确定它们的顺序，以使地址尽可能连续。

在必要时，使用 `unsigned char` 或 `unsigned short` 来压缩空间也是一种方法，但这并不总是有效，因为 8 和 16 位的运算可能会带来性能损失，见参考资料 [8]。

例如，ST 表的预处理适合以下的内存布局和访问顺序：

```
int t[20][1 << 20];

for (int k = 0; k < log_n; k++)
for (int i = 2 << k; i <= n; i++)
    t[k + 1][i] = min(t[k][i - (1 << k)], t[k][i]);
```

区间 DP 的常规的写法可能如下：

```
for (int len = 1; len < n; len++)
    for (int l = 1; l <= n - len; l++) {
        int r = l + len;
        for (int k = l; k < r; k++)
            dp[l][r] <- dp[l][k] + dp[k + 1][r] + ...
    }
```

交换 for 循环的顺序可以使访问更连续：

```
for (int l = n; l >= 1; l--)
    for (int k = l; k <= n; k++)
        for (int r = k + 1; r <= n; r++)
            dp[l][r] <- dp[l][k] + dp[k + 1][r] + ...
```

[4] https://zh.cppreference.com/w/cpp/language/alignas

2.5.2 时间局部性

时间局部性是让适合缓存的数据在一段时间内被反复访问。

最常见的例子是滚动数组，而滚动数组有两种写法：开两倍，开一倍（需要正确的转移顺序）。通常，开一倍会更快。

另一种方法是分块卡进缓存。例如，下面一段代码对 a 数组进行若干次区间求和：

```
int sum(int* a, int l, int r); // 对 a[l..r] 求和（循环展开）

int a[1000000], L[1000], R[1000], ans[1000];
memset(a, -1, sizeof a);
for (int i = 0; i < 1000; i++) ans[i] = sum(a, L[i], R[i])
```

a 数组太大了，只适合 L3 缓存。如果把 a 数组分割为若干适合 L1 的块，对每块扫 1000 遍再处理下一块，就达到了时间局部性的目的：

```
for (int k = 0; k < 1000000; k += 4000) // 处理块 [k, k + 3999]
    for (int i = 0; i < 1000; i++)
        ans[i] += sum(a, max(L[i], k), min(R[i], k + 3999));
```

在 OI 中，bitset 题[5]经常需要这样的分块操作（建议的每块的大小为 256/512 bit），一方面节省了空间，另一方面让数组适合 L3 缓存。

正确地选择块大小才能有效地利用缓存，对于 L1, L2 缓存，数据总量在缓存大小的 50% ~ 80% 比较合适，而 L3 缓存是共享的，最好不要超过 $\frac{1}{3}$。

还有一个方法是复用寄存器，减少内存读写的次数，比如 1000×1000 的矩阵乘法。

```
for (int i = 0; i < 1000; i += 4)
    for (int k = 0; k < 1000; k += 2)
        for (int j = 0; j < 1000; j++)
            #pragma GCC unroll 4
            for (int ii = 0; ii < 4; ii++) {
                c[i + ii][j] += a[i + ii][k + 0] * b[k + 0][j];
                c[i + ii][j] += a[i + ii][k + 1] * b[k + 1][j];
            }
```

前两个 for 枚举了 a 矩阵中一个 4×2 的小块，这 8 个元素会持久地保留在寄存器中。b 矩阵读取的次数只有 $\frac{1000^3}{4}$，c 也只有 $\frac{1000^3}{4}$，而且 c 满足了时间局部性，一段时间内频繁访问的数据量适合 L1 缓存。这样就极大地缓解了内存的压力，平均每次 `c += a * b` 只有 1.15 cycle，而乘法的吞吐量倒数是 1 cycle。

2.5.3 内存级并行

内存请求的并行处理通常比串行处理要快得多，因此优先选择可以并行的算法。遍历链表和平衡树是不能并行的，而树状数组和非递归线段树可以并行，内存可以并行的条件是所有的地址都可以直接通过简单算术运算得到，不需要跟随指针。

[5]例如「JSOI2019」精准预测 https://loj.ac/p/3101

2.5.4 树重新编号

如果需要在树上进行多次动态规划或查询，重新编号树可以提高访问速度。不同的编号方式有不同的效果：

- 按 BFS 序重新编号后，记 fa_u 表示 u 的父亲，那么 fa_u 是一个单调不减的数组，所以在树上进行动态规划时访问连续，使得它比 DFS 序更适合动态规划。

  自下往上的例子：

  ```
  for (int u = n; u >= 2; u--) dp[fa[u]] = max(dp[fa[u]], dp[u] + 1);
  ```

  自上往下的例子：

  ```
  for (int u = 2; u <= n; u++) {
      int A = dp2[fa[u]], B = dp[fa[u]]; // 去除分支
      dp3[u] = max(dp3[fa[u]], dp[u] + 1 == B ? A : B) + 1;
  }
  ```

  如果需要从父亲枚举所有儿子，可以预处理 R_u 表示 u 的最后一个儿子的编号。

  ```
  for (int i = 1; i <= n; i++) R[fa[i]] = i;
  for (int i = 1; i < n; i++) R[i + 1] = max(R[i + 1], R[i]);
  for (int u = n; u >= 1; u--) {
      sz[u] = 1;
      for (int v = R[u]; v != R[u - 1]; v--)
          sz[u] += sz[v];
  }
  ```

  如果树是通过 prufer 序随机生成的，枚举儿子的内层循环的分支预测未命中率会比较高，所以枚举儿子的写法效率并不高。

- 按 DFS 序重新编号可以支持快速遍历子树。

- 按重链剖分 DFS 序重新编号后，任意一条链可以拆成 $O(\log n)$ 个区间，把这些区间的左右端点分别求出来，分别用 for 循环遍历，这样 $O(n^2)$ 暴力可以处理更大的数据范围，可能能得到更高的分数。

2.6 高效使用 STL

使用 STL 时，有时会遇到 TLE 的问题。然而，STL 中有许多函数是为了提高性能而设计的。在 cppreference[6] 上，你可以了解到许多有用的函数。本节将总结这些函数的使用技巧，而不会详细介绍每个函数。对于不熟悉的函数，请参阅 cppreference 进行查询。

[6] https://zh.cppreference.com/

2.6.1 vector

- `reserve(n)`: 为 `vector` 预分配至少 n 的容量。当大小还不到 n 时，`push_back` 就不会重新分配内存。
- `emplace_back( ... )`: 经过测试，其速度不会慢于 `push_back`，但是通常不会比 `push_back` 快，只有像 `vector` 套 `vector` 这种复杂类型上 `emplace_back` 才有优势。
- 非成员函数 `move`: 如果你想将 `vector` A 赋值为 B，并且 B 不再需要时，可以使用 `A = move(B)`。这对其他 STL 容器也适用。

2.6.2 set

- `insert(x)`: 这个函数有返回值，可以利用起来。
- `erase(it)`: 这个函数返回 `next(it)`,并且复杂度为 $O(1)$。因此,将 `s.erase(it++)` 改写为 `it = s.erase(it)` 会更好。当元素值域很小时，可以开一个迭代器数组记录每个值的迭代器，这样就可以通过迭代器删除元素，复杂度为 $O(1)$。
- `insert(it, x)`: 当我们想插入 x，并且知道插入的位置恰好在迭代器 it 之前时，就可以使用这个函数。复杂度为 $O(1)$，返回值为插入元素的迭代器。如果 x 不应该插入在 it 之前，复杂度退化到 $O(\log n)$。它的 `emplace` 版本为 `emplace_hint`。

  两种典型的用法为：

  - 从一些元素初始化 `set` 时，可以先排序，再连续调用 `s.insert(s.end(), x)`。
  - 同时插入 x 和 x-1。

    ```
    auto it = s.insert(x).first;
    s.insert(it, x - 1);
    ```

- `it++, it--`: 常数很小的 $O(\log n)$，应当积极使用。

上述方法对 `map` 也适用。

2.6.3 priority_queue

- `priority_queue` 的构造函数是 $O(n)$ 的，即线性构造堆算法。
- 两个 `priority_queue` 可以实现可删除堆，避免使用 `multiset`。

2.6.4 unordered_map

`__gnu_pbds::gp_hash_table` 的速度远快于 `unordered_map`，它唯一的不足是哈希函数不合理，在非随机数据下没有复杂度保证。因此，可以使用自定义哈希函数[7]来弥补这一点。

2.6.5 sort

在使用 `sort` 函数时，使用传统函数作为比较方法会导致效率低下，我们应该使用小于符号的重载或匿名函数。原因如下：

- 传统函数：由于函数地址作为实参传递，无法避免函数调用的开销。
- 小于符号重载：当我们使用自定义类型时，编译器会为 `sort` 生成一个特定的实例。在这个实例中，小于符号被内联优化。
- 匿名函数：匿名函数本质上是定义了一个类，并重载了该类的 `()` 运算符。匿名函数本身是该类型的一个实例，因此当它作为 `sort` 的比较函数时，编译器也会为 `sort` 生成一个特定的实例。

[7]推 荐 https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structures/HashMap.h

3 计算密集型算法

3.1 计数题通用优化

当模数为常量时，编译器会自动使用取模算法进行优化，因此这种情况下的程序会比需要输入模数的程序运行得更快。无论模数是否固定，减少取模次数都是优化的首选方法。

3.1.1 首选优化

以下是一些简单但普遍需要的优化方法：

- 当内层循环有公因式时，应将其提取出来。
- 只要在 uint64_t 的范围内，就不需要频繁地取模，例如 `a * b + c * d` 只需要在最后取模。
- 在模意义下进行加减运算时，不需要取模，只需要用 if 进行判断。值得一提的是，加法有一种替代写法。这种写法通常会稍微快一点，但有较低的概率会被编译成分支而不是 cmov，这时就会慢很多倍。

```
void inc(int& a, int b) {
    if (int(unsigned(a += b) - P) >= 0) a -= P;
}
```

3.1.2 使用无符号整数

使用 uint64_t 取模通常比使用 long long 快，因为处理负数取模相对复杂。以下是有符号取模和无符号取模的性能信息对比：

类型	延迟	吞吐量倒数	fused-uops/uops	端口需求
有符号（常量模数）	10	2.5	10/8	2*p1 + p5 + 2*p06 + 3*p0156
无符号（常量模数）	9	2	7/6	2*p1 + p5 + p06 + 2*p0156
有符号（变量模数）	可变	24.38	57/56	很多
无符号（变量模数）	可变	21.04	36/33	很多

有符号取模（常量模数）的吞吐量倒数受到前端限制，因为 fused-uops 太多。

无符号整数还有另一个优点：unsigned -> uint64_t 的转换通常是免费的，而 int -> long long 的转换则不是，原因如下：

- int -> long long 的规则是非负数高 32 位补 0，负数补 1。这需要 movsxd 指令（性能信息和 add 一样）。

- unsigned -> uint64_t 的规则是高 32 位补 0，由于每次修改 32 位寄存器时都会自动把高 32 位清 0，所以通常不需要用额外的指令清 0。

但是，uint64_t -> unsigned -> uint64_t 的转换并不是免费的，因为这需要把高 32 位清 0。为了避免这种转换，临时变量应始终使用 uint64_t。

在下文中，我们约定 `u32 = uint32_t, u64 = uint64_t, u128 = __uint128_t`。

3.1.3 使用较大数据类型进行暂存

请看下面的代码：

```
const int P = 998244353;
u64 foobar(u32 a[2000], u32 b[2000]) {
    u64 res = 0;
    for (int i = 0; i < 2000; i++) res = (res + (u64)a[i] * b[i]) % P;
    return res % P;
}
```

一个简单的优化是使用 u128 来暂存结果。

```
u128 tmp = 0;
for (int i = 0; i < 2000; i++) tmp += (u64)a[i] * b[i];
return tmp % P;
```

u128 是用两个 64 位寄存器模拟出来的。加法需要 add 和 adc 指令，速度还是很快的。而取模则是调用库函数，虽然相对较慢，但是取一次模的代价还是值得的。

如果要计算 4 个数组的内积 $(\sum a_i b_i c_i d_i) \bmod P$，u128 就可能存不下，解决方案是每 128 次中断一次，对 $128P^4$ 取一次模：

```
u128 tmp = 0;
for (int k = 0; k < 2000; k += 128) {
    for (int i = k; i < min(2000, k + 128); i++) {
        auto A = (u64)a[i] * b[i];
        auto B = (u64)c[i] * d[i];
        tmp += (u128)A * B;
    }
    const u128 _128P4 = (u128)4 * P * P * P * P;
    auto t = tmp - _128P4;
    if ((__int128)t >= 0) tmp = t;
}
return tmp % P;
```

这里的 `(u128)A * B` 看起来是非常慢的，其实不然。因为这个表达式会被编译成 `mul r64` 指令，该指令计算 r64 与 rax 的完整 128 位乘积，低 64 和高 64 分别放进 rax, rdx。

指令	延迟	吞吐量倒数	fused-uops/uops	端口需求
`mul r64`	4	1	2/2	p1 + p5

另一个例子是小型矩阵乘法，如果用 u128 的话就会进行 n^2 次 u128 取模，这样很慢，所以应该使用 u64 暂存结果，当最外层循环每迭代 8 次的时候就全体对 $8P^2$ 取模。

3.1.4 打破取模链

取模操作的延迟很高，因此我们应该避免这样的取模链：

```
(u64)a * b % P * c % P * d % P * ...
```

我们可以将其替换为分治型合并：

```
// a * b * c * d
auto L = (u64)a * b % P;
auto R = (u64)c * d % P;
auto res = L * R % P;
```

对于超长的乘法链，我们可以使用循环展开来打破它。例如，计算阶乘可以这样写：

```
u64 tmp[8];
for (int i = 0; i < 8; i++) tmp[i] = 1;
for (u64 k = 1; k <= 1000000000; k += 8)
    #pragma GCC unroll 8
    for (int i = 0; i < 8; i++)
        tmp[i] = tmp[i] * (k + i) % P;
u64 ans = 1;
for (int i = 0; i < 8; i++) ans = ans * tmp[i] % P;
```

计算 $N!$ 只需要 $3N$ cycles，瓶颈在端口 p1。在下一节 3.2 中，我们可以将其优化到 N cycles。

3.1.5 求和与前缀和

在模意义下，求和与前缀和的操作都可以做得非常快，长度为 N 的数组求和用时 $0.26N$ cycles，前缀用时 $1.44N$ cycles。

求和可以使用 u64 来实现双倍的加载速度，从而突破速度瓶颈。而且，一次 u64 加法相当于两次 u32 加法，前提是结果不超过 $2^{32}-1$。因此，我们可以每次加载 4 个 u64 并求和，然后再把高低 32 位分离开来。以下代码用时 $0.26N$ cycles。

```
u64 sum(u32 a[2000]) {
    assert((size_t)a % 8 == 0); // 确保定义时使用 alignas(8)
    u64 s0 = 0, s1 = 0;
    #pragma GCC unroll 32
    for (int i = 0; i < 2000; i += 8) {
        #define load(p) *(u64*)&p
        u64 t = load(a[i]) + load(a[i + 2]) + load(a[i + 4]) + load(a[i + 6]);
        s0 += (u32)t, s1 += t >> 32;
    }
    return (s0 + s1) % P;
}
```

直接计算前缀和会很慢，因为模意义下的加法延迟较高而无法并行。为了实现并行，我们需要将数组等分成三个部分，通过提前对前两个部分求和，使得三个部分可以同时开始计算前缀和。下面的代码用时 1.44N cycles（包含提前求和）。

```
void inc(u32& a, u32 b) {
    if (int((a += b) - P) >= 0) a -= P;
}
void psum(u32 a[2000]) {
    u32 s0 = 0, s1 = sum(a, 0, 666), s2 = sum(a, 666, 666*2);
    inc(s2, s1);
    #pragma GCC unroll 16
    for (int i = 0; i < 666; i++) {
        inc(s0, a[i]), a[i] = s0;
        inc(s1, a[i + 666]), a[i + 666] = s1;
        inc(s2, a[i + 666*2]), a[i + 666*2] = s2;
    }
    inc(s2, a[1998]), a[1998] = s2;
    inc(s2, a[1999]), a[1999] = s2;
}
```

3.2 内积优化

在模意义下，内积可以达到相当理想的速度（平均每次操作 1 cycle）。使用 SIMD，甚至可以达到 1/6 cycles。卷积和向量乘矩阵都是内积的形式，因此在许多情况下，可以替代 FFT 类算法骗分。

计算内积的最优实现方法是将每 16 个数分为一块，然后在块内使用 u64 暂存结果，最后再用 u128 求和。以下是相关的代码：

```
u64 iproduct(u32 a[2000], u64 b[2000]) {
    u128 res = 0;
    for (int i = 0; i < 2000; i += 16) {
        u64 s = 0;
        #pragma GCC unroll 16
        for (int j = 0; j < 16; j++) s += a[j + i] * b[j + i];
        res += s;
    }
    return res % P;
}
```

在 GCC 9 中，长度为 N 的内积只需要 1.02N cycles。特别注意，a 和 b 这两个数组中至少有一个必须是 u64，否则实际性能会远低于理论性能。

从 GCC 11 开始，编译的结果更加优秀，至少有一个 u64 的情况下只需要 1.01N cycles。即使都是 u32，通过对外层循环进行循环展开也可以达到 1.04N cycles。在本节，我们将会暂时使用 GCC 11 进行研究。

例题. （在线卷积）[8] 此题省略推导后，有如下递推式：

[8] https://codeforces.com/contest/1792/problem/F2

$$A_i = \frac{1}{i}\left(2\sum_{j=1}^{i-1} jA_jA_{i-j} - (i-1)A_{i-1}\right) \tag{3}$$

$n \le 5\times 10^4, 5.5s$

令 $B_i = iA_i$，那么只需要对每个 i 计算 $\sum_{j=1}^{i-1} B_jA_{i-j}$，套用内积即可。需要决定一下用不用 u64，u64 的内积速度更快，但也会增加内存压力，经过测试后，不使用 u64 速度会更快。

在本机用时 320ms。

```
a[1] = b[1] = 1;
fo r(int i = 2; i <= n; i++) {
    u128 sum = 0;
    #pragma GCC unroll 4
    for (int j = 0; j < i; j += 16) {
        u64 s = 0;
        #pragma GCC unroll 16
        for (int k = 0; k < 16; k++)
            s += (u64)b[j + k] * a[i - j - k];
            // a 有偏移量，不会越界
        sum += s;
    }
    a[i] = (sum * 2 - b[i - 1]) * inv[i] % P;
    b[i] = (u64)a[i] * i % P;
}
```

例题. （多项式多点求值）[9] 给定 n 次多项式 $f(x)$，对 $i \in [1,m]$ 计算 $f(x_i)$。

$n, m \le 64000, 1.8s$

$f(x) = \sum_{i=0}^{n} a_ix^i$ 可以视作 $\{a_i\}$ 和 $\{x^i\}$ 的内积，可是后者无法直接算出，所以把系数分块：

$$f(x) = \sum_{k=0}^{\lfloor \frac{n}{256} \rfloor} x^{256k} \sum_{i=0}^{255} a_{256k+i}x^i \tag{4}$$

先把 $\{1, x, x^2, \cdots, x^{255}\}$ 算出来用 u64 存，然后把每个块与它做内积再合并。一个实现细节是 u128 的高效取模方法，详见代码。

在本机用时 1.14s。

[9] https://www.luogu.com.cn/problem/P5050

```cpp
// 多项式系数为 a[0..n]
// 求值点 b[0..m-1]
for (int k = 0; k < m; k++) {
    const int B = 256;
    u64 pows[B + 1] = {1};
    for (int i = 0; i < B; i++)
        pows[i + 1] = pows[i] * b[k] % P;
    u64 res = 0;
    for (int i = n & -B; i >= 0; i -= B) {
        u128 s0 = 0;
        #pragma GCC unroll 2
        for (int j0 = 0; j0 < B; j0 += 16) {
            u64 s1 = 0;
            #pragma GCC unroll 16
            for (int j1 = 0; j1 < 16; j1++)
                s1 += a[i + j0 + j1] * pows[j0 + j1];
            s0 += s1;
        }
        const u64 r64 = -(u64)P % P; // 2^64 % P
        u64 low = s0, high = s0 >> 64;
        if (low >= 8ULL * P * P) low -= 8ULL * P * P;
        res = (res * pows[B] + high * r64 + low) % P;
    }
    ans[k] = res;
}
```

这段代码还有两个优化方向：

- 加速预处理 $\{1, x, x^2, \cdots, x^{255}\}$ 的过程，因为这个过程是串行的，速度较为缓慢。
- 复用寄存器：扫描一遍系数的时候同时计算多个答案，这样可以减少加载指令。

例题. （快速阶乘算法）[10] 给定质数 p，计算 $n! \bmod p$，有 T 组数据。

$n, p \le 2^{32} - 1, T \le 5, 2s$

根据 Wilson 定理，我们有

$$\begin{aligned} (p-1)! &\equiv -1 \pmod p \\ \prod_{i=1}^{n} i \prod_{i=1}^{p-1-n} (-i) &\equiv -1 \pmod p \\ n!(p-1-n)!(-1)^{p-1-n} &\equiv -1 \pmod p \end{aligned} \tag{5}$$

因此，我们只需要选择 n 和 $p-1-n$ 中较小的一个来计算阶乘，这样就只有 5×10^9 的计算量。

阶乘可以看作 $(x+1)(x+2)(x+3)\cdots(x+B)$ 的多点求值，套用一下就能过了。

[10] https://www.luogu.com.cn/problem/P5282

值得一提的是，还有一个可以加速 4.5 倍的改进：对于每个 $i \le [1,n]$，我们可以把它的 $2,3,5,7$ 的因子全部提出来，这样每个数都变成了一个和 $2,3,5,7$ 互质的数。满足这个条件的取值只有 $N(1-\frac{1}{2})(1-\frac{1}{3})(1-\frac{1}{5})(1-\frac{1}{7})$ 种。对于一个和 $2,3,5,7$ 互质的数 x，它的出现次数等于 $\lfloor \frac{n}{x} \rfloor$ 以内只含有 $2,3,5,7$ 的数的数量。对于一个 k，出现次数恰好为 k 的数在一个区间中，我们可以把区间的乘积算出来，然后把乘积的 k 次方乘进答案里面。

3.3 整数除法

除法和取模指令一直以来都需要庞大的 CPU 资源，而且会暂停 CPU 的前端。因此有各种各样的算法可以通过对除数预处理来大大加速之后的除法，这些算法可以优化需要输入模数的程序、计算大量除法的数论算法。有的算法比编译器所用的方法还要高效，在模数为常量的程序中也能发挥作用。

3.3.1 Barrett reduction

首先 $\lfloor \frac{a}{m} \rfloor = \lfloor a \times \frac{1}{m} \rfloor$，把 $\frac{1}{m}$ 近似为一个分数 $\frac{p}{2^k}$，为了让二者尽可能的接近，p 应该等于 $\lfloor \frac{1}{m} \times 2^k \rfloor$ 或者 $\lceil \frac{1}{m} \times 2^k \rceil$，选择哪一个之后再决定。

于是 $\lfloor \frac{a}{m} \rfloor \approx \lfloor \frac{a \times p}{2^k} \rfloor$，然后就可以愉快地用乘法和右移了，那么问题来了，它能算对吗？

定理. 设 $\frac{p}{2^k} = \frac{1}{m} + \epsilon$，$\epsilon$ 在不同范围内有不同的效果。

- 当 $|\epsilon| \le \frac{1}{a}$ 时，$\lfloor \frac{a \times p}{2^k} \rfloor$ 和 $\lfloor \frac{a}{m} \rfloor$ 至多差 1，而且 $\frac{p}{2^k} > \frac{1}{m}$ 时答案只可能偏大，$\frac{p}{2^k} < \frac{1}{m}$ 时答案只可能偏小。
- 当 $0 \le \epsilon < \frac{1}{am}$ 时，答案是准确无误的。

证明.

$$\begin{aligned} \lfloor \frac{a \times p}{2^k} \rfloor &= \lfloor a \times \left(\frac{1}{m} + \epsilon \right) \rfloor \\ &= \lfloor \frac{a}{m} + a\epsilon \rfloor \end{aligned} \tag{6}$$

只要 $|a\epsilon| \le 1$，即 $|\epsilon| \le \frac{1}{a}$ 时，答案就至多偏差 1。

什么时候答案是准确的呢，$\frac{a}{m}$ 加上非负的 $a\epsilon$ 后要想改变整数部分，$a\epsilon$ 至少得为 $\frac{1}{m}$，只要让 $0 \le a\epsilon < \frac{1}{m}$，整数部分就肯定不会改变，即 $\lfloor \frac{a}{m} + a\epsilon \rfloor$ 和 $\lfloor \frac{a}{m} \rfloor$ 始终相等。 □

那么定理第一部分怎么应用？它可以用于取模，假如 $\lfloor \frac{a}{m} \rfloor$ 少算了 1，那么 $a - \lfloor \frac{a}{m} \rfloor \times m$ 仍然在 $[0, 2m)$ 的范围内，用一次 if 就可以搞定。在实践中，我们通常会遇到 10^9 级别的模数 m，而 a 则是两个数的乘积，即 m^2 级别，我们令 $k = 64, p = \lfloor \frac{1}{m} \times 2^k \rfloor$，那么 $|\epsilon| < \frac{1}{2^{64}}$，对于 $[0, 2^{64})$ 以内的 a, m 都是有效的。算法的关键步骤 $\lfloor \frac{a \times p}{2^{64}} \rfloor$ 会将两个 u64 乘起来提取高 64 位，

可以通过 x86 的 `mul r64` 指令实现，所以速度很快。而算法的最后一步“对 $[0, 2m)$ 内的结果取模”可以选择去掉，因为我们可以将所有数维持在 $[0, 2m)$ 中，加减法变为对 $2m$ 取模。

接下来实现一个输入模数，然后使用 Barrett reduction 取模的函数：

```
u64 p, m;
void set_mod(u32 mod) {
    assert(mod >= 2);
    m = mod, p = -m / m + 1; // 补码技巧计算 2^64 / m
}
u64 mul(u64 a, u64 b) {
    u64 c = a * b;
    c -= u64((u128)c * p >> 64) * m;
    if (c >= m) c -= m;
    return c;
}
```

定理第二部分就更有用了，只不过条件有点苛刻。这也是编译器的默认取模算法，我们来了解一下编译器的实现。

```
u64 mod(u64 a) { return a % 998244353; }
```

根据定理，我们需要让 $\epsilon < \frac{1}{(2^{64}-1)\times 998244353}$，令 $k = 93, p = \lceil \frac{1}{m} \times 2^k \rceil = 9920937979283557439$，$\epsilon = \frac{p}{2^k} - \frac{1}{m}$ 就足够小了，编译器生成的汇编代码就这样做的！

```
# rdi % 998244353 -> rax
movabs  rax, -8525806094425994177 # =9920937979283557439
mul     rdi
mov     rax, rdx
shr     rax, 29
imul    rdx, rax, 998244353
mov     rax, rdi
sub     rax, rdx
```

在输入模数的情况下要用这个算法只能运行时决定 k，因为 k 过大会导致 p 就不能存进 u64，过小又不能保证正确性。

定理第二部分的主要价值是计算除法，在数论算法中会用到很多除法，而且除数通常都不超过 $\sqrt{n}$，于是可以对 $[2, \sqrt{n}]$ 预处理 p，来实现快速除法。通常选择 $k = 64$，这时的适用条件为 $am \le 2^{64}$。示例代码如下：

```
u64 inv[32000];

void init(int n) {
    for (int i = 2; i <= n; i++) inv[i] = ~0ULL / i + 1; // ceil(2^64 / i)
}
u64 idiv(u64 a, u64 b) {
    // 确保 b ≠ 1
    return (u128)a * inv[b] >> 64;
}
```

3.3.2 快速模乘算法

Barrett reduction 计算取模是间接的，意思是它是通过计算除法来得到余数的。还有直接计算余数的算法[11]，它比 Barrett reduction 更高效，可惜的是它不适合 OI 中 10^9 级别的模数，所以我提出了一个修改版本，让它 10^9 级别的模数上发挥作用。

首先

$$\begin{aligned} a \times z \bmod m &= \{a \times \frac{z}{m}\} \times m \\ \{x\} &= x - \lfloor x \rfloor \end{aligned} \tag{7}$$

原理是 $a \times z \bmod m$ 只和 $a \times \frac{z}{m}$ 的小数部分有关。

和 Barrett reduction 一样，把 $\frac{z}{m}$ 近似为分数 $\frac{p}{2^k}$，其中 $p = \lceil \frac{z}{m} \times 2^k \rceil$，那么

$$\begin{aligned} a \times z \bmod m &\approx \{a \times \frac{p}{2^k}\} \times m \\ &= \frac{ap \bmod 2^k}{2^k} \times m \\ &= \frac{ap \bmod 2^k \times m}{2^k} \end{aligned} \tag{8}$$

最终结果不一定是整数，因为之前 p 已经上取整了，这里就选择下取整。我们有一个与 Barrett reduction 类似的定理。

定理. 设 $\frac{p}{2^k} = \frac{1}{m} + \epsilon$，当 $\epsilon < \frac{1}{am}$ 时，答案是准确无误的。

证明.

$$\begin{aligned} \lfloor \{a \times \frac{p}{2^k}\} \times m \rfloor &= \lfloor \{a \times (\frac{z}{m} + \epsilon)\} \times m \rfloor \\ &= \lfloor \{a \times \frac{z}{m} + a\epsilon\} \times m \rfloor \end{aligned} \tag{9}$$

如果 $a \times \frac{z}{m}$ 加上 $a\epsilon$ 后整数部分都改变了，那么小数部分也就大相径庭了，为了让整数部分不改变，需要保证 $a\epsilon < \frac{1}{m}$，然后继续推导：

$$\begin{aligned} &= \lfloor \{a \times \frac{z}{m} + a\epsilon\} \times m \rfloor \\ &= \lfloor \left(\{a \times \frac{z}{m}\} + a\epsilon \right) m \rfloor \\ &= \lfloor \{a \times \frac{z}{m}\} \times m + am\epsilon \rfloor \end{aligned} \tag{10}$$

因为 $\{a \times \frac{z}{m}\} \times m$ 是整数，只要后一个部分 $a_i m\epsilon < 1$ 答案就是准确的，注意到 $a_i m\epsilon < 1$ 其实和之前的必要条件 $a\epsilon < \frac{1}{m}$ 等价。 □

[11] https://arxiv.org/pdf/1902.01961.pdf

在实践中一般取 $k = 64$，因为 $k > 64$ 的时候计算 $ap \bmod 2^k$ 会遇到麻烦。根据定理，$k = 64$ 适合 $2^{32} - 1$ 以内的 a, z, m。

这个算法需要对 z 预处理出 p，由于 z 通常不是固定不变的，我们需要一个更快的预处理方法。本来 $p = \lceil \frac{z}{m} \times 2^{64} \rceil$，我们把它改成 $\lfloor \frac{z}{m} \times 2^{64} \rfloor + 1$ 也无伤大雅，唯一的区别是 ϵ 的范围从 $[0, \frac{1}{2^{64}})$ 变成了 $[0, \frac{1}{2^{64}}]$。

$$
\begin{aligned}
p &= \lfloor \frac{z}{m} \times 2^{64} \rfloor + 1 \\
\text{let } 2^{64} &= Am + B \\
p &= \lfloor \frac{z}{m} \times (Am + B) \rfloor + 1 \\
p &= \lfloor \frac{z}{m} \times (Am + B) \rfloor + 1 \\
p &= \lfloor z \times \frac{B}{m} \rfloor + zA + 1
\end{aligned}
\tag{11}
$$

注意到 $\lfloor z \times \frac{B}{m} \rfloor$ 可以用 Barrett reduction 算，把 $\frac{B}{m}$ 近似为一个分数 $\frac{q}{2^{64}}$，其中 $q = \lceil \frac{B}{m} \times 2^{64} \rceil$，即：

$$
\lfloor z \times \frac{B}{m} \rfloor = \lfloor \frac{zq}{2^{64}} \rfloor \tag{12}
$$

实现如下：

```
const int P = 998244353;

u64 trans(u64 x) {
    constexpr u64 A = -(u64)P / P + 1;
    constexpr u64 q = ((u128(-(u64)P % P) << 64) + P - 1) / P;
    return x * A + u64((u128)x * q >> 64) + 1;
}
u64 mul(u64 a, u64 z) {
    return a * z * (u128)P >> 64;
}
```

计算等比数列 $1, x, x^2, \cdots$ 时就可以把 x 转化成 p，从而更快地完成 $\cdot x \bmod P$ 的操作。

```
void foobar(int n, u32 x, u32* pows) {
    u64 p = trans(x), now = 1;
    for (int i = 0; i < n; i++) pows[i] = now, now = mul(now, p);
}
```

trans 和 mul 的性能信息如下：

表达式	延迟	吞吐量倒数	fused-uops/uops	端口需求
`mul(a, p)`	6	2	4/3	2*p1 + p5
`trans(z)`	7	3	7/6	3*p1 + p5
`a * z % P`	12	3	7/6	3*p1 + p5 + p06 + 2*p0156

3.3.3 更快的小型内积

快速模乘算法还可以推广到小型内积 $(a_1b_1 + a_2b_2 + \cdots + a_nb_n) \bmod m$，只需要令 $p_i = \lceil \frac{b_i}{m} \times 2^{64} \rceil$，那么：

$$\left(\sum a_i b_i\right) \bmod m = \lfloor \frac{(\sum a_i p_i) \bmod 2^{64} \times m}{2^{64}} \rfloor \tag{13}$$

它适用的条件是 $\sum a_i \le \frac{2^{64}}{m}$。

这个方法可以广泛地优化计数类 DP，因为 DP 的转移通常都是内积形式，比如这种形式就很好用。

$$dp_{i,j} = \frac{i}{n} \times dp_{i-1,j-1} + \frac{n-i}{n} \times dp_{i-1,j} \tag{14}$$

3.3.4 整除判定

我们可以使用快速模乘算法来解决整除判定问题。例如，如果我们想要判断一个数 m 是否能被另一个数 a 整除，我们可以首先写出余数公式：

$$\begin{aligned} a \times z \bmod m &= \lfloor \{a \times \frac{p}{2^{64}}\} \times m \rfloor \\ &= \lfloor \frac{ap \bmod 2^{64} \times m}{2^{64}} \rfloor \end{aligned} \tag{15}$$

在这里，我们令 $z = 1$，那么 $p = \lceil \frac{2^{64}}{m} \rceil$，我们可以得到一个余数恰好等于 0 的充要条件：

$$ap \bmod 2^{64} < p \tag{16}$$

这是一个非常简洁的条件，实际效率也非常高。它的适用条件是 $am \le 2^{64}$。

此外，还有另一个方法，它能处理 10^{18} 级别的 a, m，但是要求 m 是奇数。我们先假设 m 能整除 a，那么存在整数 k 使得 $a = mk$，然后我们可以通过下面的方式计算出这个 k：

$$\begin{aligned} a &\equiv mk \pmod{2^{64}} \\ k &\equiv am^{-1} \pmod{2^{64}} \end{aligned} \tag{17}$$

我们可以得出结论：$mk < 2^{64}$ 是 m 整除 a 的充要条件。

证明. 必要性：如果 m 整除 a 那么 $mk = a < 2^{64}$。充分性：$mk < 2^{64}$ 和 $mk \equiv a \pmod{2^{64}}$ 可以推出 $mk = a$。 □

因此，我们的最终判定条件为：

$$a \times (m^{-1} \bmod 2^{64}) \bmod 2^{64} < \lceil \frac{2^{64}}{m} \rceil \tag{18}$$

计算 m 模 2^{64} 的逆元最快的方法是使用牛顿迭代法，具体方法见参考资料 [10]。

3.3.5 Montgomery 乘法

Montgomery 乘法也是应用广泛的取模算法，要求模数必须是奇数，见参考资料 [11]。适合 10^{18} 级别的模数，因此可以加速 Pollard's rho。向量化后效率高也是它的亮点。

3.3.6 浮点数辅助取模技巧

使用浮点数也可以方便地实现取模。

假如有一个模数 $p \le 10^{15}$，可以用下面的代码计算 `a * b % p`：

```
long long mul(long long a, long long b, long long p) {
    auto c = a * b - (long long)(1.0 * a * b / p + 0.5) * p;
    if (c < 0) c += p;
    return c;
}
```

原理是这样的：

- 先算出 $\lfloor \frac{ab}{p} \rfloor$ 的近似值，可以证明 `1.0 * a * b / p` 的误差小于 $\frac{p}{4\times 10^{15}}$。
- 然后计算 $ab - \lfloor \frac{ab}{p} \rfloor \times p$，这里让 long long 溢出了，所以结果会对 2^{64} 取模，但是不会影响答案。
- 最后再用 if 解决 $\lfloor \frac{ab}{p} \rfloor$ 算出来偏大的情况。

为了提升速度，预处理 `rp = 1.0 / p`，然后把 `1.0 * a * b / p` 替换成 `rp * a * b`，这样会稍微增大误差，不过 $p \le 10^{15}$ 时仍然能保证正确性。

```
long long mul(long long a, long long b, long long p, double rp) {
    auto c = a * b - (long long)(rp * a * b + 0.5) * p;
    if (c < 0) c += p;
    return c;
}
```

如果 b 也是定值的话，还可以进一步优化，预处理 `b_p = 1.0 * b / p`，然后就把浮点数乘法减少到了 1 次。

有几个注意事项：

- 不要把 long long 换成 u64，因为无符号整数和浮点数的相互转化较慢。
- 10^{18} 级别的模数需要切换成 long double。

即使是 10^9 级别的输入模数，这个算法也是比直接取模快不少的。以下是性能信息的对比，由于所有的延迟都较大，表格中并未列出具体的延迟数据。

表达式	吞吐量倒数	fused-uops/uops
`(u64)a * b % p`	21	39/34
预处理 `1.0 / p`	5.05	19/16
预处理 `1.0 * b / p`	4	15/13

3.4 整除分块

整除分块的一个主要瓶颈在于除法运算，因此我提出了一个改进的方法。

通常，整除分块的目的是计算以下这种式子：

$$\sum_{ab\le n} f(a)g(b) \tag{19}$$

一个更好的计算方法是分别计算 $a \le \lfloor \sqrt{n} \rfloor$ 和 $b \le \lfloor \sqrt{n} \rfloor$ 的贡献，然后再减去 $\max(a,b) \le \lfloor \sqrt{n} \rfloor$ 的贡献。用公式表达就是：

$$\sum_{ab\le n} f(a)g(b) = \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} \left(f(i)S_g(\lfloor \frac{n}{i} \rfloor) + g(i)S_f(\lfloor \frac{n}{i} \rfloor) \right) - S_f(\lfloor \sqrt{n} \rfloor)S_g(\lfloor \sqrt{n} \rfloor) \tag{20}$$

在这里，S_f 和 S_g 分别表示 f 和 g 的前缀和。这个公式只需要 $\lfloor \sqrt{n} \rfloor$ 次除法，次数为传统整除分块的 $\frac{1}{4}$，并且打破了除法依赖链。当需要多次运行整除分块时，我们还可以使用 Barrett reduction 来进一步优化除法。

3.5 bitset

手写 bitset 可以轻松比 std::bitset 更快。即使某些字符串问题的数据范围较大[12]，手写 bitset 仍然有机会通过。

std::bitset 可以被轻松超越是因为它有以下不足：

- 不灵活，只能操作整体，不支持操作区间。
- 每完成一次运算都会把结果写入内存，比如 `a | (b & c)` 和 `a |= a << shift` 都会在内存中转一次。
- 部分函数本身速度较慢，比如 `count`, `any`, `_Find_next`。

现在我们来一一讨论每个问题该怎么用手写 bitset 优化。

[12]例如「NOI2023」字符串 https://loj.ac/p/3983

首先我们应该用 u64 数组来存 bitset，AND, OR, XOR 运算都是简单的，需要提速的时候就循环展开。

不平凡的操作是位移。

3.6 位移

比起整体左移和右移，区间复制是更有用操作，我们实现一个类似于 memcpy 的 `bitcpy(d, di, s, si, len)` 函数，表示把 $d_{di..di+len-1}$ 赋值为 $s_{si..si+len-1}$，实现流程如下：

1. 开始时每次复制 1 bit，直到 $di \bmod 64 = 0$ 或者已经复制了 len 个 bit。
2. 从现在开始每次复制 64 bit，即 $d_{di..di+63} \leftarrow s_{si..si+63}$，直到剩余 bit 数不足 64。
3. 最后每次复制 1 bit，直到已经复制了 len 个 bit。

最复杂的是第二个阶段，$d_{di..di+63}$ 刚好是一个 u64，而 $s_{si..si+63}$ 可能跨越两个 u64，需要用位移把两个部分拼起来。注意特判 $si \bmod 64 = 0$ 的情况，不然会发生位移量为 64 的位移，这是未定义行为。

代码如下，我们需要使用 `__restrict__` 关键字来提示编译器两个范围不相交，不然优化效果不好：

```
void bitcpy(u64* __restrict__ d, u64 di, u64* __restrict__ s, u64 si, u64 len) {
    while (len && di % 64) assign(d, di++, s, si++), len--; // assign 函数复制单个 bit
    auto dst = d + di / 64, src = s + si / 64;
    auto end = dst + len / 64;
    auto shift = si % 64;
    if (!shift) memcpy(dst, src, (end - dst) * 8);
    else while (dst != end) *dst++ = src[0] >> shift | src[1] << 64 - shift, src++;
    di += len & -64, si += len & -64, len %= 64;
    while (len--) assign(d, di++, s, si++);
}
```

3.7 运算融合

运算融合是将表达式中的多个运算合并到一个循环中，从而避免了中间结果的内存读写操作。例如，对于没有位移操作的表达式，如 `a = (a & b) | c`，运算融合相对简单，如果表达式中包含一个位移操作，可以把表达式嵌入 bitcpy 中。然而，当表达式中包含多个位移操作时，完全融合就变得相当复杂。考虑到 bitcpy 的性能瓶颈并非内存读写，因此在每次 bitcpy 后进行内存中转对效率的影响并不大。

3.8 popcount

如果能用 pragma 的话 popcount 根本不是问题。在代码中加入 `#pragma target("popcnt")` 后 `__builtin_popcountll` 就会编译成 x86 指令 popcnt，并行情况下每周期可以处理 64 bits。

不能用 pragma 的话，`__builtin_popcountll` 的速度就比较慢了，这是因为它会调用二进制库中的函数 `__popcountdi2`，这个函数本身实现很好，但是调用开销会让它的性能减半，所以最好不要使用它，以下是一些替代方案：

- 对 0~65535 预处理 popcount，然后每次读取一个 `uint16_t` 再查表，速度大概 10.23 bits/cycle。

```
auto ptr = (uint16_t*)bits;
u64 res = 0;
for (int i = 0; i < N / 16; i++) res += lookUp[ptr[i]];
```

- 手写一个和 `__popcountdi2` 实现一样的函数，这个代码的原理见参考资料 [12]。速度大概 14.17 bits/cycle。

```
u64 popcnt(u64 n) {
    n -= n >> 1 & 0x5555555555555555;
    n = (n & 0x3333333333333333) + (n >> 2 & 0x3333333333333333);
    n = (n + (n >> 4)) & 0x0F0F0F0F0F0F0F0F;
    return n * 0x101010101010101 >> 56;
}
    u64 res = 0;
    #pragma GCC unroll 8
    for (int i = 0; i < N / 64; i++) res += popcnt(bits[i]);
```

- 继续优化上一个方法。我们的目的是求出很多 u64 的 popcount 之和，所以在完成前两步迭代后，进行若干次合并，把 u64 的数量减下去再完成后两步迭代。速度大概 19.00 bits/cycle。

```
u64 popcnt(u64 n) {
    n = (n & 0x0F0F0F0F0F0F0F0F) + (n >> 4 & 0x0F0F0F0F0F0F0F0F);
    return n * 0x101010101010101 >> 56;
}
u64 res = 0;
for (int i = 0; i < N / 64; i += 7) {
    u64 low = 0, high = 0;
    #pragma GCC unroll 7
    for (int j = 0; j < 7; j++) {
        u64 n = bits[i + j];
        n -= n >> 1 & 0x5555555555555555;
        low += n & 0x3333333333333333;
        high += n >> 2 & 0x3333333333333333;
    }
    res += popcnt(low) + popcnt(high);
}
```

这个合并的技巧还可以发挥到极致，见参考资料 [16]

3.9 判断 bitset 是否全 0

较优的实现是使用循环展开求出所有 u64 的 OR。

3.10 遍历所有的 1

较优的实现是枚举每个 u64，然后用如下方式遍历当前 u64 中的所有 1。

```
while (n) {
    int i = __builtin_ctzll(n);
    ...
    n &= n - 1;
}
```

3.11 快速傅里叶变换

快速傅里叶变换的常规写法每次蝴蝶变换的乘数都会切换，这个切换是可以消除的。有一种优秀写法只会切换 $O(n)$ 次，它既不是 DIT 也不是 DIF，但和 DIT 和 DIF 都有一些相似之处，接下来我以 DIF 的方式来解释它：

记 $DFT(a,n,z)_k$ 表示 $\sum_{i=0}^{n-1}(z^i a_i)\omega_n^{ik}$，相当于先给 a 点乘上等比数列 $\{1,z,z^2,\cdots\}$ 再做 DFT。考虑将多项式 a 的系数分为前半和后半：

$$\begin{aligned} DFT(a,n,z)_k &= \sum_{i=0}^{n-1}(z^i a_i)\omega_n^{ik} \\ &= \sum_{i=0}^{n/2-1}\left((z^i a_i^{(left)}) + (z^i a_i^{(right)})z^{n/2}\omega_n^{n/2\cdot k}\right)\omega_n^{ik} \\ &= \sum_{i=0}^{n/2-1}\left((z^i a_i^{(left)}) + (z^i a_i^{(right)})z^{n/2}(-1)^k\right)\omega_n^{ik} \end{aligned} \tag{21}$$

通过讨论 k 的奇偶性，可以把 $(-1)^k$ 去掉：

$$\begin{aligned} DFT(a,n,z)_{2k} &= \sum_{i=0}^{n/2-1}\left((z^i a_i^{(left)}) + (z^i a_i^{(right)})z^{n/2}\right)\omega_{n/2}^{ik} \\ &= DFT(a^{(left)} + z^{n/2}a^{(right)}, n/2, z)_k \\ DFT(a,n,z)_{2k+1} &= \sum_{i=0}^{n/2-1}\left((z^i a_i^{(left)}) - (z^i a_i^{(right)})z^{n/2}\right)\omega_n^i\omega_{n/2}^{ik} \\ &= DFT(a^{(left)} - z^{n/2}a^{(right)}, n/2, z\cdot\omega_n)_k \end{aligned} \tag{22}$$

由于偶数的答案和奇数的答案分别构成了子问题，实现上可以把偶数的答案放到左边，奇数的答案放到右边，这样会导致最终答案的顺序乱了，我们后面再讨论这个问题。

另一个问题是每次蝴蝶变换的乘数的 $z^{n/2}$ 怎么求出来，有没有什么简洁的形式？结论是递归到区间 $[l,r)$ 时，蝴蝶变换的乘数 $z^{(r-l)/2}=f(N/l)$，其中

$$f(2^{k_1}+2^{k_2}+\cdots+2^{k_n})=\omega_{2^{k_1+2}}\omega_{2^{k_2+2}}\cdots\omega_{2^{k_n+2}}\quad(k_1<k_2<\cdots<k_n)\tag{23}$$

f 的预处理方法是倍增：$f_{i+2^k}=f_i\cdot\omega_{2^{k+2}}$。

最后代码如下：

```
for (int i = n >> 1; i; i >>= 1)
    for (int j = 0; j < n / 2 / i; j++)
        for (int k = j * i * 2; k < j * i * 2 + i; k++) {
            u64 x = a[k], y = (u64)a[k + i] * w[j] % P;
            a[k] = add(x, y), a[k + i] = sub(x, y);
        }
```

这样 DFT 后点值顺序是乱的，但是我们可以不管他：直接把点值乘起来，要 IDFT 回去的话就把这个函数的每一步依次撤销。

其他有效的优化：

- 使用 3.3.2 中的快速模乘算法对 `w` 数组预处理。
- 循环展开，$i\le 8$ 的情况还要单独拎出来。

加上了所有优化的代码见参考资料 [16]。

4 数据结构

数据结构的性能瓶颈通常在于内存请求和分支预测未命中，使用指针的数据结构更加低效，因为内存请求无法并发。以下是一些参考优化方向：

- 使用结构体捆绑结点信息，不重要的信息放在外面。另外，成员类型要按大小从大到小排列，这样才能最小化结构体的大小。
- 尝试离散化、离线算法，选择简单的数据结构。
- 采用更好的实现方式，这也是本章的主要内容。

4.1 非递归线段树的优化

非递归线段树是一种效率高且易于编写的数据结构，其基本原理可以参考资料 [18]。本节将介绍一些可以优化的细节。首先，我们约定数组长度为 $M = 2^{20}$，数据类型为 int。

4.1.1 修改

在进行修改操作时，如果可以避免使用 pushUp，那就尽量避免，因为 pushUp 会产生内存依赖链，并增加内存请求。

- 单点加区间和的最优写法。这种写法的所有内存请求都是独立的。

```
void add(int i, int v) {
    for (i += M; i; i >>= 1) c[i] += v;
}
```

- 单点增大区间 Max 的最优写法。实验证明，无论在哪种数据下，这种写法都比 `c[i] = max(c[i], v)` 更快。

```
// let c[0] = inf
void checkMax(int i, int v) {
    for (i += M; v > c[i]; i >>= 1) c[i] = v;
}
```

- 单点修改维护 Max 的最优写法。虽然需要先读后写，但是读和写都是可以并行的。

```
void modify(int i, int v) {
    for (i += M; i; i >>= 1) c[i] = v, v = max(v, c[i ^ 1]);
}
```

这种方法本质上也是 pushUp，但是它利用了信息的交换律，从而减少了内存请求。这个方法还可以用在区间修改中，例如下面的区间加和区间 Max

```
int c[M << 1][2]; // c[i][0] is max, c[i][1] is tag;
void add(int l, int r, int v) {
    l += M - 1, r += M + 1;
    int L = c[l][0], R = c[r][0];
    while (l ^ r ^ 1) {
        int vl = ~l & 1 ? v : 0, vr = r & 1 ? v : 0; // 去除分支
        L = max(L, c[l ^ 1][0] += vl), c[l ^ 1][1] += vl;
        R = max(R, c[r ^ 1][0] += vr), c[r ^ 1][1] += vr;
        l >>= 1, c[l][0] = L += c[l][1];
        r >>= 1, c[r][0] = R += c[r][1];
    }
    L = max(L, R);
    while (l >>= 1) c[l][0] = L += c[l][1], L = max(L, c[l ^ 1][0]);
}
```

Dijkstra 算法的堆选择非递归线段树会更快。每个结点维护 min 和它的下标，尽量将值和下标压缩到一个 u64 中，这样就可以只维护 u64 的最小值。

4.1.2 区间查询

如果没有维护任何标记，区间查询可以直接定位出所有需要的结点，而无需逐层跳跃。下面是单点修改和区间 Max 的示例：

```
int lg(int n) { return 31 ^ __builtin_clz(n); }
int query(int l, int r) {
    l += M - 1, r += M + 1;
    int o = l >> lg(l ^ r); // lca 的左儿子
    int L = -1e9, R = -1e9; // 解除两路的依赖关系
    while ((l >>= __builtin_ctz(~l)) != o) L = max(L, c[++l]);
    o++; // lca 的右儿子
    while ((r >>= __builtin_ctz(r)) != o) R = max(R, c[--r]);
    return max(L, R);
}
```

4.1.3 懒标记下放

虽然没有标记永久化快，懒标记下放也是可行的。下面的代码可以避免某些结点被 pushDown 两次。

```
int lg(int n) { return 31 ^ __builtin_clz(n); }
int query(int l, int r) {
    l += M - 1, r += M + 1;
    for (int i = 20; i >= 1; i--) pushDown(l >> i);
    for (int i = lg(l ^ r); i >= 1; i--) pushDown(r >> i);
    ...
}
```

4.2 线段树上二分的优化

线段树上二分是一个常见的操作，有时它用来将 $O(\log^2 n)$ 的复杂度降低到 $O(\log n)$，有时在 pushUp 的时候也需要它。

线段树的结构非常适合进行二分操作。通过本节的优化，线段树上二分将比树状数组的二分快 4 倍。

4.2.1 集合第 k 小

例题. 维护一个可重集，支持加入/删除元素，查询第 k 小，值域 10^6。

树状数组可以简洁地查询第 k 小，原理见参考资料 [15]。

```
// M = 1 << 20
int kth(int k) {
    int p = 0;
    for (int i = M >> 1; i; i >>= 1)
        if (k > c[p + i]) p += i, k -= c[p];
    return p + 1;
}
```

为了压缩空间并提升内存访问效率，非递归线段树也可以采用树状数组的策略，每个结点存储左半段的和，叶子结点都被删除，第 k 小的查询非常好写：

```
int c[M];
int kth(int k) {
    int i = 1;
    while (i < M) {
        if (k > c[i]) k -= c[i], i = i * 2 + 1;
        else i = i * 2;
    }
    return i - M;
}
```

由于存储方式的问题，add 和 sum 都需要重写，见参考资料 [16]。重写后的 add 的速度已经超过了树状数组的 add，但是 sum 还没有。

下面是速度测试环节，将 10^6 个均匀随机数插入可重集中，然后每次查询的 k 在 $[1, 10^6]$ 均匀随机。

类型	单次查询周期数
树状数组	639
非递归线段树	356

为什么非递归线段树比树状数组快了整整一倍，这得谈到内存：

- 树形数据结构都有一个特点：离根越近的元素访问频率越高，最终形成的局面是前几层在缓存中，后几层不在缓存中。位于缓存的层数越多整体访问速度就越快。

- 线段树的前几层空间是连续的，每 16 个元素才占用一个缓存行，而树状数组前几层是分散的，每个元素都会占用一个缓存行。

- 树状数组的前几层地址构成了一个公差为 2^k 的等差数列。根据缓存分组关联的特性 (1.4.3)，只有常数个元素能够进入 L1 缓存。幸运的是，由于虚拟地址到物理地址的转换，等差数列的公差不会超过 4096，因此一部分元素可以被保留在 L2 缓存中。而对于 L3 缓存，其容量足够大，可以存储整个树状数组。

综上，线段树有前若干层访问很快的优点，但是树状数组没有。

我们接下来继续优化线段树上二分的操作[13]。目前的首要瓶颈是分支预测，这是因为随机数据导致预测的命中率只有 50%。我们首先将分支替换为 cmov。

```
int kth(int k) {
    int i = 1;
    while (i < M) {
        bool flag = k > c[i];
        if (flag) k -= c[i];
        i = i * 2 + flag;
    }
    return i - M;
}
```

再次测试后，结果令人大跌眼镜，每次查询需要 336 cycles，并没有显著的进步。通过 Profiler 测得的分治预测未命中率 $< 10^{-6}$，这说明问题出在了其他地方。

这里就要介绍一下分支预测的正面作用了。分支预测器预测了接下来的走向后，就会沿着这一路推测执行，还会进行预取 (Prefetch)，意思是提前将数据移入 L1 缓存。由于线段树的结构原因，当前结点的 2 个儿子、4 个孙子等都位于同一缓存行，无论预测是否命中，都获取了正确的缓存行。没有分支预测的话，处理器就完全不知道接下来要访问的数据，所以不会进行预取，然后每次都需要完整地等待读取结果。

预取也可以手动执行，使用 `__builtin_prefetch(address)` 函数可以预取 address 所在的缓存行，如果 address 非法只会忽略该请求，不会引发运行时错误。由于 i 的 16 个 4 级后代刚好适合一个缓存行，我们就预取所有四级后代，然后正式访问数据的时候延迟就很小了。

```
alignas(64) int c[M]; // 确保 i 的 4 级后代位于同一个缓存行
int kth(int k) {
    int i = 1;
    while (i < M) {
        __builtin_prefetch(&c[i * 16]);
        bool flag = k > c[i];
        if (flag) k -= c[i];
        i = i * 2 + flag;
    }
    return i - M;
}
```

[13] 灵感来源 https://en.algorithmica.org/hpc/data-structures/binary-search/

现在速度得到了较大提升，每次循环只需要 203 cycles 了。

因为这是一个串行算法，我们在算术上做优化也能提升速度，利用一点奇技淫巧，下面的代码又加速到了 167 cycles。

```
int kth(uint32_t k) {
    k--;
    size_t i = 1;
    while (i < M) {
        __builtin_prefetch(&c[i * 16]);
        int t = k - c[i];
        if (t >= 0) k = t;
        i = i * 2 + (t >= 0);
    }
    return i - M;
}
```

你可能已经认为这段代码优化不动了，那么请看下面的实验：强制一遍 kth 得出结果后才能开始下一次 kth。

```
for (int i = 0; i < N; i++) last = kth(k[i] + (last == 114514));
```

这样会让 kth 变慢到 250 cycles，这是什么原因呢？如果我不强制 kth 顺序执行，即使当前 kth 还没有执行完，分支预测器就已经知道当前循环还有多少次结束，然后将提前下一次 kth 的指令放入流水线，但是调度器的容量也只有 97，这意味着只有当前 kth 执行到最后几十条指令时才能开始并行下一次 kth 调用。我们从中获得启发，将多个操作融合在一起就可以大力度地并行，例如设计 `kth_and_kth`, `kth_and_add` 这种的融合函数。

4.2.2 局部二分

对一个区间进行二分也是非常常见的操作。下面给出一个示例代码，该代码实现了“从 i 开始向后查找第一个大于 v 的位置”：

```
// 确保序列末尾是 inf
int find(size_t i, int v) {
    i += M - 1;
    while (i >>= __builtin_ctz(~i), c[++i] <= v);
    while (i < M) {
        __builtin_prefetch(&c[i * 16]);
        i = i * 2 + (c[i * 2] <= v);
    }
    return i - M;
}
```

4.3 等位信息捆绑

“等位信息”是指每次修改位置和查询位置都相同的信息。最常见的例子是树状数组的区间加区间和，其原理可以参考资料 [17]。实现方法如下：

- 首先，我们需要维护两个数组 A 和 B。
- 当我们要对后缀 $[i,n]$ 加上一个值 v 时，我们可以令 $A_i += v$ 和 $B_i += v \cdot i$。
- 要求前缀 $[1,i]$ 的和，我们可以计算 $(i+1) \cdot S_A(i) - S_B(i)$。

在这里，A 和 B 就是等位信息。一个更优的实现方法是将 A 和 B 用一个结构体捆绑在一起，并且修改和查询都只使用一个循环。

4.3.1 ST 表

ST 表的预处理最优写法见 2.5.1。本节主要讨论以下两个问题：

- 数组的哪一维放在前面会使查询更快。
- 如何最快地计算 $\lfloor \log_2(r-l+1) \rfloor$。

对于第一个问题，答案是将较小的一维放在前面。原因是统计学效应：人为生成的或有意义的数据通常满足 $\log_2(r-l+1)$ 呈正态分布，频率较高的数据会留在缓存中。如果我们特意让 $\log_2(r-l+1)$ 在 $[0, \log_2 n]$ 中均匀生成，那么哪一维放在前面的速度就会一样。

对于计算 $\lfloor \log_2(n) \rfloor$,最常见的方法是预处理一个 `log_2` 数组,或者使用 `std::__lg(n)`。后者的效率很不错，但它并不是最快的。最快的方法是使用 `31 ^ __builtin_clz(n)`。这个原因需要在汇编层面解释：bsr 指令就是我们想要的功能，`std::__lg(n)` 的实现是 `31 - __builtin_clz(n)`，`__builtin_clz(n)` 会被编译为 bsr 后再 xor 31，减法和异或不能抵消（新版本编译器可以抵消），就产生了冗余的操作。如果我们使用 `31 ^ __builtin_clz(n)`，那么两次异或就会被编译器优化掉。

最后，ST 表查询最小值位置可以用 long long 压缩一下，高 32 位存储值，低 32 位存储下标。如果值很大，我们可以分配更多的位给值。

4.3.2 无旋 Treap

见参考资料 [19]。

5 杂项

5.1 vector 替代品

vector 的主要使用场景是使用 vector 数组存储总量 $O(n)$ 的元素，比如存图。这种使用方式的时间常数和空间常数并不优秀。

我们从存图出发，了解如何实现一个优秀的存图方式。最常用的两种存图方式为 vector 数组和邻接表，它们都不是完美的。

速度	vector 数组	邻接表
加边	慢	快
遍历出边	快	慢

完美的存图方式是存在的，它兼具加边速度和遍历出边的速度。首先，统计每个点的度数 d_u，然后对 d 数组做前缀和，最后把 u 的所有出边放进区间 $[d_{u-1}, d_u)$ 中，这一步要按照计数排序的写法。示例如下：

```
// 无向图
int a[E], b[E], d[V + 1], G[E << 1];

void build(int n, int m) {
    for (int i = 0; i < m; i++)
        cin >> a[i] >> b[i], d[a[i]]++, d[b[i]]++;
    for (int i = 0; i < n; i++) d[i + 1] += d[i];
    for (int i = 0; i < m; i++) {
        G[--d[a[i]]] = b[i];
        G[--d[b[i]]] = a[i];
    }
}
void edges(int u) {
    for (int i = d[u]; i < d[u + 1]; i++)
        cout << G[i] << ' ';
}
```

这种写法可以轻易地推广到有向图和任意 vector 数组。

5.2 Binary GCD

Binary GCD 是一种小常数的 gcd 算法，其原理如下：

$$\gcd(a,b)=\begin{cases}2\gcd(a/2,b/2) & (a,b\text{ 均为偶数})\\ \gcd(a/2,b) & (a\text{ 为偶数},b\text{ 为奇数})\\ \gcd(|a-b|,\min(a,b)) & (a,b\text{ 均为奇数})\end{cases}\tag{24}$$

这个算法的关键在于实现：

- 首先，将 a,b 中的 2 除尽。
- 当 a,b 都是奇数时，会递归到 $|a-b|,\min(a,b)$，用 `__builtin_ctz` 将 $|a-b|$ 中的 2 除尽后，又回到了 a,b 都是奇数的情况。这个过程会一直循环，直到 $a=b$。
- 在计算 $a-b$ 的绝对值之前，先计算 $ctz(a-b)$，因为 $a-b$ 和 $b-a$ 的 ctz 是相等的，所以在计算绝对值之前计算 ctz 可以与计算绝对值并行。

下面是一个示例[14]，它的速度足以通过「基于值域预处理的快速 GCD」[15]。

```
int gcd(int a, int b) {
    if (!a | !b) return a + b;
    int az = __builtin_ctz(a);
    int bz = __builtin_ctz(b);
    int z = min(az, bz);
    a >>= az, b >>= bz;
    while (a != b) {
        int diff = b - a;
        az = __builtin_ctz(diff);
        b = min(a, b), a = abs(diff) >> az;
    }
    return a << z;
}
```

5.2.1 常见问题解答

Q. `inline` 加上会变快吗？

内联函数的主要目的是消除函数调用的开销。编译器在这方面足够好了，它能够自动地进行合理的内联处理，因此我们通常无需手动添加 inline 关键字。然而，编译器也不能 100% 准确地判断出哪些函数应该进行内联处理，在确实需要进行内联的情况下，我们可以使用 `__attribute__((always_inline))` 来强制进行内联。

Q. 手写读入中 `x = x * 10 + c - 48` 改成 `x = (x << 1) + (x << 3) + (c ^ 48)` 更快吗？

`(x << 1) + (x << 3)` 是完全没有必要的，因为编译器的实现既没有乘法也没有位运算。

`c - 48` 和 `c ^ 48` 哪个快是微架构相关的，这并不是因为减法和异或的速度差异，而是因为减法被编译成了 `lea eax, [ecx + eax * 2 - 48]`，这在 Coffee Lake 上特别

[14]来源于 https://en.algorithmica.org/hpc/algorithms/gcd/，做了少量修改
[15]https://www.luogu.com.cn/problem/P5435

慢，而在较新的微架构上又特别快。另外，`mov eax, DWORD PTE [ecx + eax * 2 - 48]` 并不慢，这是因为加载端口里有专门计算 lea 的执行单元，它比 ALU 计算 lea 还要快一些。

Q. 数组大小开 2^k 会变慢吗？

可能会，但数组大小本身和速度没有直接关系，数组大小开 2^k 可能是损害性能的间接原因，直接原因见 1.4.3。

6 性能分析

在不借助工具的情况下，唯一能测量的性能信息就是时间。然而，性能分析工具可以提供更多的有用信息，来帮助寻找程序的优化方向。以下是一些我尝试过的工具：

6.1 Profiler

Profiler 可以统计各种事件，比如时钟周期、缓存未命中率、分支预测未命中率、各个函数的开销占比等等。

- perf 是一个命令行工具。获取方法：`apt-get install linux-tools-common linux-tools-generic linux-tools-`uname -r``。文档链接：https://perf.wiki.kernel.org/index.php/Tutorial
- likwid perfctr 也是一个命令行工具，它的功能比 perf 要丰富一些，比如测量代码局部的性能事件。获取链接：https://github.com/RRZE-HPC/likwid。文档链接：https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr。

6.2 缓存系统/分支预测模拟工具

Cachegrind 是一个缓存系统/分支预测模拟器。获取链接：https://valgrind.org/downloads/。文档链接:https://valgrind.org/docs/manual/cg-manual.html

6.3 机器代码分析/流水线模拟

uiCA 是一个流水线模拟器，可以计算吞吐量，还可以提供微架构中的细节。获取链接：https://www.uops.info/uiCA.html。文档链接：[4]，这篇文献也是一份关于微架构的可靠资料。

此外，llvm-mca 和 OSACA 也是流行的机器代码分析工具。

6.4 确保测量结果的准确性

在某些情况下，我们可能会发现测量结果并不准确。这可能是由于编译器对需要测量的操作进行了过度优化，或者是由于我们未能了解的一些硬件特性，例如，未经修改的全局数组不会分配空间，读取时总能命中 L1 缓存。为了确保我们能够获得准确的测量结果，可以参考 [20]。

7 总结

在我们讨论了众多的优化原理和实例之后，相信大家已经理解到，程序优化是一个既复杂又有趣的领域。虽然程序优化本身没有固定的规律，但在大量的实践中，我们可以总结出一些有效的方法和技巧。

最后，程序优化是永无止境的，祝愿大家能够享受优化过程带来的乐趣和满足感。

致谢

感谢中国计算机学会提供学习和交流的平台。
感谢父母对我的培养、关心与支持。
感谢祝敏磊老师和李翔老师对我的培养与教导。
感谢国家集训队杨耀良教练、彭思进教练和张艺缤教练的指导。
感谢李翔老师、何钒佑、崔隽章同学为本文审稿。

参考资料

[1] Sergey Slotin, Algorithms for Modern Hardware, https://en.algorithmica.org/hpc/

[2] Agner Fog, The microarchitecture of Intel, AMD, and VIA CPUs, https://www.agner.org/optimize/microarchitecture.pdf

[3] Agner Fog, Optimizing software in C++, https://www.agner.org/optimize/optimizing_cpp.pdf

[4] Andreas Abel and Jan Reineke, uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures, https://arxiv.org/pdf/2107.14210.pdf

[5] 骆可强《论程序底层优化的一些方法与技巧》，IOI 2009 集训队论文集。

[6] x86 and amd64 instruction reference, https://www.felixcloutier.com/x86/

[7] 快速范围判断，https://zhuanlan.zhihu.com/p/147039093

[8] 8 和 16 位运算的缺点，https://stackoverflow.com/questions/41573502

[9] Wikipedia, Tree-Pseudo-LRU, https://en.wikipedia.org/wiki/Pseudo-LRU#Tree-PLRU

[10] Fast-inverse-trick, https://cp-algorithms.com/algebra/montgomery_multiplication.html#fast-inverse-trick

[11] Montgomery Multiplication, https://cp-algorithms.com/algebra/montgomery_multiplication.html

[12] Algorithms behind Popcount, https://nimrod.blog/posts/algorithms-behind-popcount/

[13] Instruction Table, https://uops.info/table.html, its introduction https://dl.acm.org/doi/pdf/10.1145/3524059.3532396

[14] Instruction tables, https://www.agner.org/optimize/instruction_tables.pdf

[15] 树状数组查询集合第 k 小，https://oi-wiki.org/ds/fenwick/#%E5%8D%95%E7%82%B9%E4%BF%AE%E6%94%B9%E6%9F%A5%E8%AF%A2%E5%85%A8%E5%B1%80%E7%AC%AC-k-%E5%B0%8F

[16] 本文补充材料 https://github.com/platelett/thesis

[17] 树状数组区间加区间和，https://oi-wiki.org/ds/fenwick/#%E5%8C%BA%E9%97%B4%E5%8A%A0%E5%8C%BA%E9%97%B4%E5%92%8C

[18] 非递归线段树，https://zhuanlan.zhihu.com/p/361935620

[19] skip2004, Treap, https://www.cnblogs.com/skip2004/p/16574235.html

[20] Idiomatic way of performance evaluation, https://stackoverflow.com/questions/60291987