打印算法优化

作者：向俊波，[1302035400@qq.com](mailto:1302035400@qq.com)，China，18683124260/18608146082

摘要：本文提出了整数打印算法和浮点数打印算法，打印结果为最常用的十进制。整数打印算法包含了打印32位整数（unsigned int）和64位整数(unsigned long long)；浮点数打印算法包含了两类算法：（1）第一类算法d2sci（双精度浮点数）和f2sci（单精度浮点数）注重性能，保证结果无损的情况下最大化性能；（2）第二类算法xjb32（单精度浮点数）和xjb64（双精度浮点数）注重打印结果可读性，本算法具有与ryu、dragonbox、schubfach等算法相同的输出结果。

关键词：整数、浮点数、打印算法、无损、性能

注：本文属于初期草稿。

# 引言

计算机内部采用二进制存储数据，通常采用整数和浮点数两种方式存储数据。整数包含有符号整数和无符号整数，浮点数通常有双精度浮点数和单精度浮点数（本文暂不考虑f16，bf16，f8等特殊浮点类型）。在将数据存为文本文件时需执行打印操作，或等效为将二进制数据转换为十进制ASCII，或称为数据可视化、渲染。打印操作广泛存在于各大工业软件、金融软件、后台监控软件、数据分析等领域，提高打印算法的性能是具有很重要的工程意义。本文算法可用于提高json库性能。本文将介绍如何优化打印算法。第二章为整数打印算法，第三章为浮点数打印算法，第四章为实验评估，评估本文算法与其他算法在相同软硬件条件下的性能差异。

（注意：由于内容较长，公式较多，如有疑问，请参考源码。时间精力有限，排版格式上可能会有很多问题，也可能会有错别字，或者证明过程错误，如果发现可以使用邮箱联系讨论。部分变量名可能会在不同场景重复使用多次，请根据上下文理解。由于相关证明过程可能具有相似繁琐性，故省略了部分内容，跨度可能较大。）。

## 整数打印

当前计算机内部整数包含有符号整数和无符号整数，所有类型如下表1-1所示。

表1-1

|  |  |  |  |
| --- | --- | --- | --- |
| 类型（有符号） | 值域 | 类型（无符号） | 值域 |
| i8 | [-128, 127] | u8 | [0, 255] |
| i16 | [-32768, 32767] | u16 | [0, 65535] |
| i32 | [-2^31, 2^31-1] | u32 | [0, 2^32-1] |
| i64 | [-2^63, 2^63-1] | u64 | [0, 2^64-1] |

有符号整数类型包含4种，其中i64类型包含其他3种类型（即其他3种类型可无损转换为i64）。无符号整数类型包含4种，其中u64包含其他3种类型（即其他3种类型可无损转换为u64）。最常用的是i32、u32、i64、u64这四种类型，其中打印i32可转换为u32，打印i64可转换为u64。故本文只介绍打印u32和u64类型（等效为C语言中的unsigned int、unsigned long long类型）的算法优化。

（1）u32打印算法

u32类型的值域为[0, 4294967295]，打印结果长度范围为[1, 10]。将长度结果范围分为4种情况。

表1-2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 长度范围 | 输入值值域 | 输入值等效形式 | example | 查表次数 |
| [1, 3] | [0, 10^3-1] | aaa | 12 | 1 |
| [4, 6] | [10^3, 10^6-1] | aaabbb | 12345 | 2 |
| [7, 9] | [10^6, 10^9-1] | aaabbbccc | 12345678 | 3 |
| 10 | [10^9, 2^32-1] | aaabbbcccd | 123456789 | 3 |

预计算0到999的包含前缀0的ASCII值并存放至常量表中，每个值采用4个字节存储，总空间为4000byte，即3.90625KB。当前大多数CPU的一级缓存大小为32KB，在打印大规模整数时不会超过一级缓存大小。

根据长度范围的不同，共有以下4种情况：

1.长度范围为[1, 3]的整数x时，等效形式为aaa即等于x，需要消除的前缀0的长度计算方式为(x<10)+(x<100)，知道前缀0的长度即可查表知道该值的ASCII值。

2.长度范围为[4, 6]的整数x的等效形式为aaabbb，先计算aaa和bbb值，需要消除的前缀0的长度计算方式为(x<10\*1000)+(x<100\*1000)，后查表得到最终打印值。

3.长度范围为[7, 9]的整数x的等效形式为aaabbbccc，计算aaa、bbb、ccc值，需要消除的前缀0的长度计算方式为(x<10\*1000\*1000)+ (x<100\*1000\*1000)，后查表得到最终打印值。

4.长度为10的整数x的等效形式为aaabbbcccd，先计算得到aaabbb、cccd，再计算得到aaa、bbb、ccc、d值，查表得到aaa、bbb、ccc的打印值，d的打印值为d+‘0’；

举例说明：12345，其长度范围位于[4, 6]。计算得出的aaa和bbb为12和345，前缀0的长度为1，通过查表得到“012”和“345”，消除最高1位0即可得到最终打印值“12345”。最多需要3次查表即可得到打印值。

（2）u64打印算法

u64类型的值域为[0, 18446744073709551615]，打印结果的长度范围为[1, 20]，其中0到9时的打印结果长度最短只有1位ASCII值，最大值18446744073709551615有20位ASCII值。本文算法将打印结果的长度范围分为7种情况，如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 长度范围 | 输入值值域 | 输入值等效形式 | example | 查表次数 |
| [1, 3] | [0, 10^3-1] | aaa | 12 | 1 |
| [4, 6] | [10^3, 10^6-1] | aaabbb | 12345 | 2 |
| [7, 9] | [10^6, 10^9-1] | aaabbbccc | 12345678 | 3 |
| [10, 12] | [10^9, 10^12-1] | aaabbbcccddd | … | 4 |
| [13, 15] | [10^12, 10^15-1] | aaabbbcccdddeee | … | 5 |
| [16, 18] | [10^15, 10^18-1] | aaabbbcccdddeeefff | … | 6 |
| [19, 20] | [10^18, 2^64-1] | aabbbcccdddeeefffggg | … | 7 |

与u32打印算法一样，共用相同的查找表。

根据不同的长度范围执行不同代码，计算输入值的等效形式，通过计算得出的等效形式查表得到其ASCII，其中等效形式中最高位aaa或aa（长度为19或20）需要消除前缀0。举例说明：（1）12：等效形式为aaa，aaa为12，查表得到“012”，前缀0长度计算方式为（aaa<10）+（aaa<100），计算得到前缀0的长度为1，消除1位‘0’得到“12”即为最终输出结果。（2）12345：等效形式为aaabbb，计算得到aaa为12，bbb为345，查表得到“012”和“345”，前缀0长度为1，消除最高1位得到“12345”。

本章整数打印算法实现在itoa\_xjb.c。

## 浮点数打印

由于负浮点数的打印结果只比其绝对值打印结果多一个负号，所以本文只讨论正浮点数，不包含特殊值如0、Nan、Inf。且本文只讨论最常用的单/双精度浮点数(即c语言中的float和double类型)，不讨论其他特殊浮点数。

IEEE754双精度浮点数包含64位，由1位符号位（）、11位指数位（）、52位尾数位（）构成，的范围为0或1，的范围为，的范围为。IEEE754单精度浮点数包含32位，由1位符号位、8位指数位、23位尾数位构成，的范围为0或1，的范围为，的范围为。当为0为irregular浮点数。单/双精度浮点数各部分长度如表1-3所示。

表1-3 单/双精度浮点数各部分长度

|  |  |  |  |
| --- | --- | --- | --- |
|  | s（符号位长度） | e（指数位长度） | f（尾数位长度） |
| Float | 1 | 8 | 23 |
| Double | 1 | 11 | 52 |

正浮点数的实数值可表示为以下表达式：



共有两种情况，当exp等于0时（称为subnormal浮点数）有：



当exp不等于0时（称为normal浮点数）有：



在浮点数的舍入区间内，解析所有实数都会舍入到该浮点数，为：



当浮点数为非irregular浮点数时称为舍入半径。

1990年，Steele和White发表论文《how to print floating-point numbers Accurately》提出浮点数打印算法的最优原则（以下简称为S&W原则）：

1. **信息保留**：打印结果可解析为原浮点数
2. **最小长度**：打印结果长度尽可能短
3. **正确舍入**：在满足1和2的基础上如果有两个候选值，应正确舍入（即选择偶数值）
4. **从左到右生成**：打印结果从左生成

满足S&W原则的浮点数打印算法将浮点数转换为结果唯一并确定的实数值。目前，ryu、dragonbox、schubfach、grisu-exact等算法已满足S&W原则。假设只满足S&W原则中第一个条件的算法归为SW1算法。满足所有条件的归为SW算法。本文将提出一个满足SW1的快速算法d2sci（用于双精度浮点数）和f2sci（用于单精度浮点数），和另外一个在schubfach算法基础上改进后满足SW的算法xjb64(用于double)，xjb32(用于float)。

### d2sci算法、f2sci算法

本算法只满足最重要的信息保留原则，旨在追求极致的性能优化，打印结果可能包含冗余数字，牺牲了一定可读性，可根据场景需求是否选择该算法。本节d2sci算法实现C语言代码地址在gitee.com/xjb714/dtoa-benchmark。f2sci算法实现在f2sci.cpp文件中。

在C语言中采用打印一个双精度浮点数可满足信息保留原则，也就是最多17位十进制有效数字，以此为灵感，d2sci算法先将输入浮点数转换为一个包含17位十进制整数和十进制指数，后打印结果至缓冲区。

在C语言中采用打印一个单精度浮点数可满足信息保留原则，也就是最多9位十进制有效数字，以此为灵感，f2sci算法先将输入浮点数转换为一个包含9位十进制整数和十进制指数，后打印结果至缓冲区。

设需要打印的实数表示为：



由得：



将输入浮点数转换为十进制科学计数法结果，并且该结果在内。满足，为整数。其中：



* 对于双精度浮点数，当不等于0，由得，有：



其中：



故有：



由于：



有：



故b的计算方式如下：



同理，对于单精度浮点数，b的计算方式如下：



* 对于双精度浮点数，当等于0，由有：



所以（下列clz函数为计算一个值前缀0个数，双精度浮点数时计算64位值，单精度浮点数时计算32位值）：



故*b*的计算方式如下：



同理，对于单精度浮点数，*b*的计算方式如下：



以上为*b*的计算过程。

对于双精度浮点数，计算的过程如下：



将四舍五入到整数即可得到计算结果。

对于单精度浮点数，计算的过程如下：



将四舍五入到整数即可得到计算结果。

采用与grisu算法相类似的自定义浮点数。将或存储在自定义浮点数查找表中。将输入浮点数v和或转换为自定义浮点数后计算得出或后再四舍五入。

其中f2sci算法可通过穷举所有单精度浮点数（正数约21亿个）证明其打印结果具有无损性；d2sci算法结果无损性需要写上证明过程。（完整证明过程在float-printing-v15.pdf。）

省略本节内容。后续有时间再补充完成。

### schubfach\_xjb算法

本算法符合S&W所有原则，在schubfach的基础上优化而来，包含适用于double的schubfach64\_xjb算法和适用于float的schubafch32\_xjb算法。

本节算法为本文3.3节算法的穷举测试提供参照，将按照本节算法作为正确答案对比测试算法输出结果是否正确。

适用于双精度浮点数的schubfach算法如图1所示，源码来自<https://github.com/abolz/Drachennest>。

相关证明过程请参考schubfach的原算法论文。



图1-1 schubfach算法代码（适用于双精度浮点数）

图1-1中算法将浮点数转换为二元组，二元组表示的实数值为:



其中图1第31行可以等效为32行，其表示的实数值相同，具有相同的十进制有效数字。

假设第31行已经替换为第32行，则返回结果二元组中的等于第16行中的变量。

假设等于第24行的变量。则返回结果二元组中的只有四种可能结果，即：



第32行的返回结果为或。第37与第40行返回结果为或。

第30行中的up\_inside与wp\_inside（用up与wp表示）共有3种情况（不存在up与wp均为1）：

表1-4

|  |  |  |  |
| --- | --- | --- | --- |
| up | wp | up!=wp | (sp+wp)\*10 |
| 0 | 0 | 0 | / |
| 0 | 1 | 1 | (s/10+1)\*10 |
| 1 | 0 | 1 | s/10\*10 |

当up为0且wp为1时，(sp+wp)\*10等于s/10\*10+10。

当up为1且wp为0时，(sp+wp)\*10等于s/10\*10。

可以用以下表达式代替：



当up为0且wp为0时，则返回结果中的为或。

第35行到第37行仅对irregular数（即double值低52位为0）有效。

第37行中的u\_inside与w\_inside（用u与w表示）共有4种情况：

表1-5

|  |  |  |  |
| --- | --- | --- | --- |
| u | w | u!=w | s+w |
| 0 | 0 | 0 | / |
| 0 | 1 | 1 | s+1 |
| 1 | 0 | 1 | s |
| 1 | 1 | 0 | / |

当u=0且w=1时结果为，此时且，则最终等价为。等价为vb&-4。

第38行和第39行可以等价为判断vb的最后3bit值是否为3、6、7。其中vb>mid等效为vb最后2bit为3，vb==mid&&(s&1)!=0等效为vb最后3bit值为6。vb的最后2bit为3共有两个情况，即vb的最后3bit值为3或7。故可以用以下表达式代替：



由于小于1e17，可等效为。

优化后的算法代码如下图1-2：

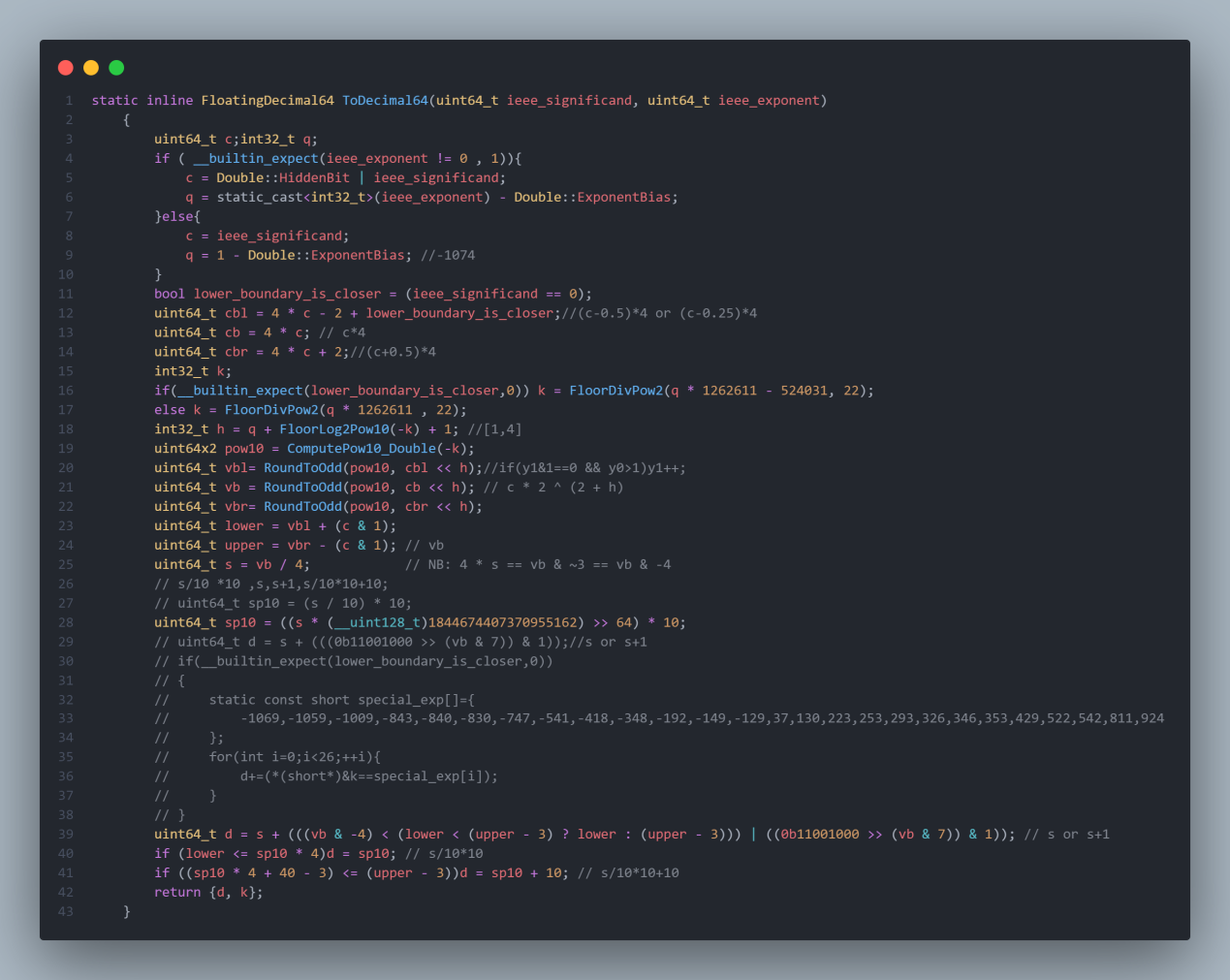


图1-2 schubfach\_xjb算法代码（适用于双精度浮点数）

图1-2优化算法通过减少指令数量，减少分支预测失败惩罚以提高性能，与图1 schubfach算法具有相同的最优输出结果。

本节schubfach\_xjb算法实现C语言代码地址在[xjb714/f2dec\_bench: a simple demo for bench double to shortest decimal algorithm.](https://github.com/xjb714/f2dec_bench)

图2算法具有与图1算法相同的输出，即满足S&W原则，schubfach\_xjb算法具有与其它满足S&W原则的算法相比更好的性能，并且本算法还支持AVX-512实现，其它算法目前并没有AVX-512实现，采用AVX-512实现的源码在gitee.com/xjb714/dtoa-benchmark。（该实现存在问题，读者暂忽略AVX-512实现。）

适用于单精度浮点数的schubafch算法原理类似于双精度浮点数，上述优化算法的核心过程也适用于单精度浮点数，本文不作介绍。

### xjb32、xjb64算法

目前其他算法使用大量分支，易造成分支预测失败惩罚，以及过多高昂的乘法开销。本文算法将尽量减少分支预测失败开销，以及减少乘法的运算次数来提高性能。

并且本文算法实现核心代码只有二十行左右，还支持并行计算，很容易改写为AVX-512实现，可利用x86架构的并行指令集AVX-512大幅提高性能。（但AVX-512实现难度较大且跨平台性弱，场景应用似乎很有限。）

将所有双精度浮点数分为irregular值和regular值两种，irregular值为低52位全为0，即frac值为0，有效的irregular值共有2046个（即exp值从1到2046），除irregular值即为regular值。同理，单精度浮点数中共有254个irregular值。当exp为0时称为subnormal浮点数。

regular浮点数中有效的值域为：



irregular浮点数中有效的值域为：



subnormal浮点数中有效的值域为：



不属于subnormal范围内的浮点数称为normal浮点数。

regular浮点数占据浮点数所有可能值的绝大部分，是最值得讨论的部分。故以下除非特别说明，只讨论regular浮点数。

假设将浮点数转换为满足S&W原则的最优解为opt，表示为：



例如双精度浮点数“1.3”，该浮点数的实际值为1.3000000000000000444089209850062616169452667236328125，浮点数的十六进制表示为3FF4CCCCCCCCCCCD，则满足S&W原则的opt值为1.3。

#### 回顾schubfach算法及本文算法推导

根据schubfach算法，d的可能取值为以下四种情况之一（本文不作证明）：



其中中的计算方式如下：



在float和double值域内，可等效为：



或者等效为以下四种情况之一：



假设的整数部分值为，小数部分值为，则有：



则的小数部分表示为：



由得的可能取值为：



等效为：



其中为最小可能取值，为最大可能取值。

假设用表示，的可能取值共有4种情况，，表示为：



计算将转换为计算和。其中的采用近似的方式。有以下几个难点：（1）如何选取；在结果正确的情况下尽量减少计算量（2）如何计算和； （3）如何在4个候选值中确定的最终值。

的最终可能取值如下：

* 

当以下条件成立时，结果为（或等效为）。即浮点数减去最小可能取值小于舍入半径。



或者当时，还需满足。故有以下条件成立：



* 

当以下条件成立时，结果为（或等效为）。最大可能取值减去浮点数小于舍入半径。



或者当时，还需满足。 故有以下条件成立：



* 或

当都不满足为或的条件时，为或，根据的小数部分来确定最终值，小数部分为0.5则舍入到最近的偶数值，小数部分不为0.5则四舍五入到最近值。对于irregular浮点数，还需判断是否在舍入区间内，不在舍入区间内则为。

综上，schubfach算法变种的步骤如下，也就是本文算法（xjb32、xjb64）：



该算法流程适用于float和double浮点数，以一个浮点数作为输入，提取和，返回计算结果（第14行）和（第2行），返回结果表示的实数值为，该值满足S&W原则，即与ryu、dragonbox、grisu-exact等算法有相同的输出。的计算过程比较简单，可由计算得出，故以下只重点介绍的快速计算过程。

以下将分为5个部分来介绍算法流程：（1）介绍算法查找表的预计算过程。（2）快速计算。（3）快速判断或。（4）快速计算出，并根据的小数部分确定出还是。（5）irregular浮点数的处理。

#### 预计算查找表

schubafch算法中float采用64位精度，double采用128位精度的查找表。本文算法的查找表也采用与schubfach相同位数精度的查找表。（本小节代码实现为gen.py。）

假设查找表中单个值数据比特长度为，对于float有，对于double有。假设有整数和实数，。有：



则：



计算得出，有以下结论：



计算查找表的方式如下（采用向上取整方式）：



对于float，当，中为整数。

对于double，当，中为整数。

* Float

通过中的值域计算得出的范围为[-32, 44]，故查找表包含了10的-32次方到10的44次方表示值。计算过程如下：



当时，查找表变量表示值与相等，其他情况下相对误差小于。表示为：



* Double

通过中的值域计算得出的范围为[-293, 323]，故查找表包含了10的-293次方到10的323次方表示值。计算过程如下：



当时，查找表变量表示值与相等，其他情况下相对误差小于。表示为：



以下用表示float范围内查找表表示值的所有可能误差，表示double范围内查找表表示值的所有可能误差，表示float或double范围内查找表表示值的所有可能误差。

中需要通过查找表得到10的次方的近似表示值，由和得当为以下范围时查找表表示值无误差：



当不为范围时，查找表表示值的误差范围有如下结论：



查找表计算过程介绍完毕。float范围查找表所需存储空间为616byte，double范围查找表所需存储空间为9872byte。

本文后续会使用到的相关定理：

对于float，的最大有效比特数为24，查找表值的最大有效比特数为64，则的最大有效比特数为88。对于double，的最大有效比特数为53，查找表值的最大有效比特数为128，则的最大有效比特数为181。

假设的二进制尾随0个数为，的二进制尾随0个数为，则的尾随0个数为。例：



#### 计算

法雷级数定义：法雷级数（Farley number column），又称法里数列，是按递增顺序排列分母不超过n的真分数构成的数列，第n阶法雷级数从0/1开始至1/1结束，每阶保留低阶全部项并插入分母为n的新分数。其包含以下几个性质：

* 严格递增。
* 每个值的分数形式最简，即分子分母互质。
* 分母小于等于n。
* 假设与相邻，则。
* 中项定理：称为和的中位数，满足。

相关定理（部分来自dragonbox论文中）：假设有正整数，其中与互质，，，，为大于等于的最佳有理数逼近结果，为小于等于的最佳有理逼近结果，且满足，，并且满足不能整除，即表示为：



假设以下成立：



则有：



故的取值范围为：



且有的小数部分范围为：



即时小数部分最小，时小数部分最大。证明如下。

当时，以下表达式取得最小值：



由得出以下表达式也取得最小值：



当时，以下表达式取得最小值：



由得出以下表达式也取得最小值：



综上，故成立。

查找最佳有理逼近函数定义如下（该函数实现在test1.py文件第15行）：



参数说明：C：整数。P：分子。Q：分母。P/Q为最简分数，P/Q<1，Q>C。返回值DN为最佳下限有理逼近分数。返回值UP为最佳上限有理逼近分数。DN和UP的分母需小于等于C。所有值均为正数。

根据法里数列的中项定理计算分母不超过C的最佳有理数逼近结果。DN和UP为法里数列C阶中相邻两项。

由得证明以下等式成立即可：



中表示浮点数，表示为一个10的整数次方有理数，表示通过查找表获取到的的近似表示值，可能相等也可能不相等。

当满足条件时，r为1，等式显然成立。当r不为1时，有：



计算的范围，有：



不为0时，存在：



为0时，，故有：



因为有：



故：



假设：



则有：



假设：



以下用表示为或，float范围为，double范围为。

* 当时，根据调用计算得出每个对应的和。

并计算满足以下条件时，BIT最小值：



* 当时，有：



故：



同理也计算BIT最小值：



综上，不同对应的BIT最小值中的最大值计算结果如下（运行结果在test1.py文件中，运行时长大约1到2秒）：



故存在以下结论：



故3.3.1节中使用的查找表满足最大精度要求。本节证毕。

快速计算出后可很快得到的值。

#### 判断或

本小节将介绍如何快速利用等价条件判断出或是否成立。

讨论中第（12）行和第（14）行，即可能为0或10。

* 讨论的情况（第12行，可能为0），等效为：



* 讨论的情况（第14行，可能为10），等效为：



由于，故有：



故有：



故对于当为整数时，必然等于。

同理，对于有：



故对于当为整数时，必然等于。

综上，即等效为讨论是否为整数。有：



根据的范围，有：



故等效为：



根据的不同范围，分为以下情况讨论：

* 

需满足：



假设：



因为为奇数，故为奇数，故有：



故以下满足：



计算的最大值问题等效为能否找到至少一个正奇数满足。当取得最大值时，一定小于5，因为如果大于等于5，那么也符合要求，与条件矛盾，故一定为1或3。将代入得：



此时float范围无解，double范围内解得。将代入得：



此时float范围解得，double范围内无解。

故有：



* 

此时分母为偶数，分子为奇数，不满足条件。

* 

此时分母为偶数，分子为奇数，不满足条件。

综上，可能为整数的情况如下：



此时的范围为：



此时查找表有一定误差，表示为：



当时，有以下结论：



当时，有以下结论：



关于中是否成立的讨论，即中是否为整数的讨论，或等效为讨论在成立时以下值是否为整数：



采用反证法，假设能够整除：



故成立时有：



此时不能整除，与矛盾，故c不能整除。故对于float，不为整数；对于double，不为整数，即：



故结论正确。

讨论关于是否为的充要条件，对于double同理，表示为：



对于float，由于充分条件必然成立，即只需要讨论必要条件，即讨论成立时是否一定成立，或等效为时一定成立。以下采用反证法。

假设时成立，故有：



等效为：



由知：



当时，为的小数部分，当时为小数部分减1。假设的小数部分表示为，故有：



同理可知double范围的范围。由知等效为以下：



当由知不为整数。

讨论不为整数时即不满足时小数部分的范围：

* 当满足以下条件时：



有：



此时不为整数，故有：



与矛盾，故原命题成立。

* 当不满足条件时：

由并排除条件：



当表达式分母满足以下条件时必然不成立，则成立：



当不满足条件时有，调用计算所有可能的上下限有理数逼近结果和：



故有结论：



通过穷举所有可能性，故有（测试代码文件为test3.py）：



此时也满足。由于与矛盾，故成立。

综上对于命题成立得证。

讨论关于是否为的充要条件。Double同理。即表示为：



同理，采用反证法，对于float，假设成立时成立。即有：



由知：



假设的小数部分为。故有：



即以下成立：



计算的可能范围。与前面计算范围同理，省略论述过程。有以下结论（测试文件为test7.py）：



与矛盾，故成立。

有以下结论：



讨论在满足时以下是否成立，来自3.3.1节到。



有：



前面已证明能够准确计算得出，则成立时能够准确计算得出等式右侧值和。

由有：



代入得：



由于条件，且t为正整数，则其中和为正整数，为正有理非整数，在满足条件下的小数部分表示为：



只需要证明中表达式右侧值相较于左侧值增加的部分加上左侧值的小数部分小于1即可使得成立。由3.3.1节到，则通过穷举法证明以下即可：



为避免浮点数计算误差，将上式等效转化为大整数运算，则可等效为：



通过穷举计算每个下的最大可能值代入上式成立即可。（运算结果在test2.py）

运行结果表明，对于float范围和double范围，总是成立，故成立，故能够准确计算得出和中等式右边的值。

对于和中等式左边的值可通过查找表计算得出，对于确定的q值，等式左边的值是唯一的（或称为一对一映射），对于float，有：



故：



当r=1时，直接移位运算即可得出结果，有：



当r>1时，依然成立。对于double，同理可证：



验证和成立的代码文件为test4.py。

讨论浮点数所有范围内以下两个值的相等或大小关系：



当r=1时，显然两值相等成立。r不为1时，有：



故有：



因为有：



假设：



故有以下结论，条件成立时：



对于所有浮点数，有：



当或时，为的边界条件，为的边界条件，根据是否为偶数确定是否为0或10。对于float（double同理，省略），存在：



假设用表示是否为偶数，为偶数则为1，否则为0。可等效为：



是否可以用判断所有浮点数中为0或10，讨论如下：

由知有以下8个结论：



从以上8个结论可推导出以下内容：

当时，，有：



当时，，有：



当时，，有：



当时，，有：



当时，故存在：



当时，故存在：



当时，故存在：



当时，故存在：



综上，故用来判断或是正确的。以上所有表达式均满足中两种情况。故以上所有表达式均满足，表示为：



对于float，可用小数部分的二进制形式左移36位向下取整快速得到。对于double，同理。能够采用开销很小的加减移位运算来解决所有浮点数的边界取值问题，在代码实现中能够被编译器编译为条件赋值语句，从而避免分支预测失败的惩罚。

本节证毕。

本小节总结：本小节内容较长。核心结果是快速判断one=0和one=10是否成立。

#### 

接下来介绍（0-32）中第5到7行的高效计算过程，即根据的小数部分决定one为还是。

先介绍的小数部分为0.5的情况，则为奇数，也为奇数，即等效为以下表达式为奇数：



根据的范围，有：



故有：



分为以下几种情况讨论：

* 

有，此时分子为偶数，分母为奇数，不满足条件。

* 

此时为偶数，不满足条件。

* 

需满足是的奇数倍。故：



故以下条件满足时为奇数：



可等效为以下条件：



条件或等效为的二进制形式中最后比特值为，则含有的尾随0个数为。此时有：



当的范围为时，通过查找表得到的值误差为0。此时查找表值的计算表达式为：



此时中含有的尾随0个数为：



根据，故的含有的尾随0个数为：



穷举所有情况，的范围如下：



对于float，只保留二进制形式小数点前36位，即舍弃掉小数点后第37位及其以后的值。同理，对于double，只保留前74位。由于，存在：



故小数点前的有效比特数范围为：



由于存在：



对于float，其中为保留的最小比特数，为的最大有效比特数。Double同理。故舍弃掉部分一定为0。故本算法能够正确处理此边界条件。

因为为的奇数倍，故有为5的奇数倍，故为5的奇数倍，即等于5或15，故一定为0.25或0.75，等效为为2.5或7.5，即小数点后第一个比特值为0或1，第二个比特值为1，其余比特值全为0，0.25对应的小数点后前两位比特值为“01”，0.75对应的小数点后前两位比特值为“11”，当为2.5时，舍入到2，当为7.5时，舍入到8。

当的小数部分不为0.5时，即不满足条件，则舍入到最近的整数值，当的小数部分小于0.5时，则，否则，可等价为。此时有：



假设用近似计算结果来替代。则需满足以下条件。

当时，有：



当时，有：



综上只需要证明满足即可，或等效为证明最终结果为。即可满足和。假设有：



假设的小数部分为，的小数部分等同于。计算的小数部分范围有：

当时小数部分范围一定有：



当时计算得出（测试代码文件为test5.py）：



也满足表达式。

假设对于float舍弃部分表示为，同理double舍弃部分表示为。则范围为：



考虑最极端的情况，计算以下表达式的边界情况：



由得：



故存在：



实际上，上述证明过程中对于float有，此时是可能存在的，是因为对于舍弃部分的上限估计值偏大，找到舍弃部分值的最大值实际上是比较困难的，但很容易通过穷举所有有效float值的输出结果来证明本文xjb32的正确性，有效的float大约21亿个，可在几秒内穷举完成。对于double范围，穷举法是非常困难的，因为这包含大约922亿亿个值。本文算法对于double给出了严格的数学证明。

对于double，保留74位比特值似乎是偏高的，从上面证明过程来看似乎保留69位比特也是有效的。

本小节证毕。

减少分支预测失败开销，无分支快速计算法：

* Float：

已穷举证明正确，请查看源码实现，本文不介绍。

* Double：

因为存在：



故保留高于等于小数点后前69位也是有效的，假设此时保留前69位的值为。因为小数点前的最大有效比特数为53，则此时保留122位比特值，则此时的最大有效比特数小于128。计算则可得到值。前面已介绍过当的小数部分为0.5时，只有为0.25或0.75两种可能。故当且仅当为0.25时计算结果错误，此时，正确值为，故当最终结果应该减去1。如何快速判断，判断小数点前64位值是否为0.25即可。证明如下：

当时，前面已证明通过查找表得到的值无误差即能准确判断小数点前64位值一定为0.25。

当时。以下表达式一定不为整数：



计算表达式的小数部分范围，测试文件为test6.py，有：



计算最接近1的两个边界情况，当时有：



当时有：



则有：



同理，对于或也成立。由知：



综上，因此当时一定存在。

故当时将计算结果减去1即可得到正确结果，可等效为将中的更改为中任意值即可。

#### irregular浮点数

由于irregular浮点数有限且数量较少，double浮点数共2046个，float浮点数共254个，可通过穷举法证明本文算法代码的正确性，故本文不作介绍，具体实现过程请参考源码。如有疑问，请读者自行检验，作者已经过测试，相比其他算法如schubfach、dragonbox具有相同的输出结果。

本小节中两个算法实现代码：xjb32算法实现在xjb32.cpp，xjb64算法实现在xjb64.cpp。在保证结果一致的情况下考虑性能最大化，本文代码实现可能与本文算法介绍过程有一定差异，以代码实现为准。

1. 实验对比

对于本文算法实现中判断或最理想的情况应该编译为cmov指令而非分支语句。目前测试结果中gcc（或g++）编译器的性能测试结果明显慢于icpx和clang编译器的测试结果，甚至性能差距达到两倍以上。目前猜测可能gcc将判断或的判断语句编译为条件分支而造成大量分支预测失败导致性能严重下降。

目前在硬件CPU为AMD 7840H和编译器为icpx 2025测试环境中，xjb32算法转换单个浮点数大约耗时2.5ns（或9cycle），xjb64算法转换单个浮点数大约耗时3.5ns（或13.5cycle）。

测试结果待完成。

初步考虑实验环境在苹果M1处理器和AMD zen4处理器R7 7840H（个人笔记本）、AMD zen5 R9 9950x（租用云服务器）。（本人无intel处理器设备，暂无条件）。

如有笔误或证明过程有误，批评指正，敬请谅解。

本文所有有关测试python代码文件附带在压缩包里。