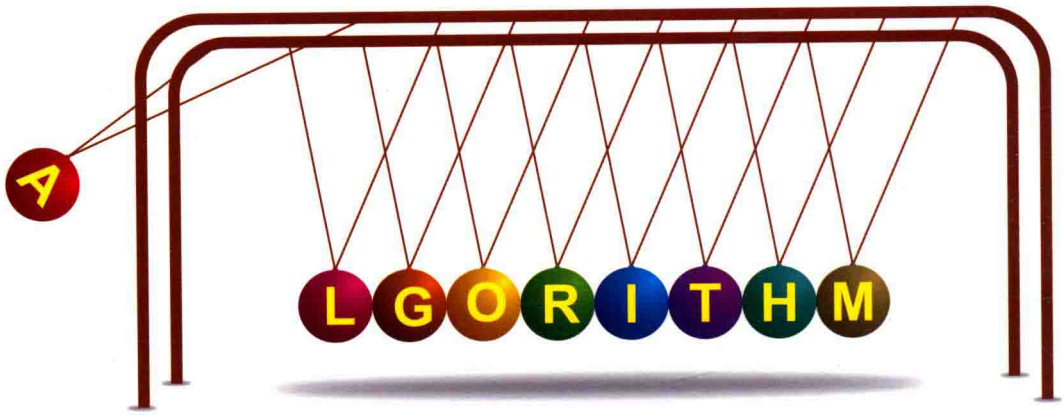
TURING 图灵原创

算法新解

刘新宇◎著

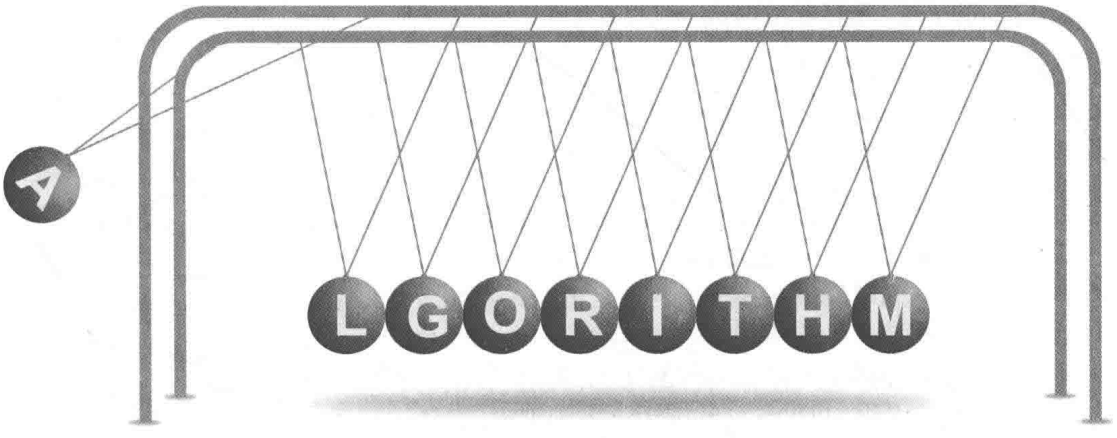


中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵原创

算法新解

刘新宇◎著



人民邮电出版社
北京

图书在版编目（CIP）数据

算法新解 / 刘新宇著. -- 北京 : 人民邮电出版社，
2017.1
（图灵原创）
ISBN 978-7-115-44035-8

Ⅰ. ①算… Ⅱ. ①刘… Ⅲ. ①电子计算机－算法理论
Ⅳ. ①TP301.6

中国版本图书馆CIP数据核字(2016)第276657号

内 容 提 要

本书分 4 部分，同时用函数式和传统方法介绍主要的基本算法和数据结构，数据结构部分包括二叉树、红黑树、AVL 树、Trie、Patricia、后缀树、B 树、二叉堆、二项式堆、斐波那契堆、配对堆、队列、序列等；基本算法部分包括各种排序算法、序列搜索算法、字符串匹配算法（KMP 等）、深度优先与广度优先搜索算法、贪心算法以及动态规划。

本书适合软件开发人员、编程和算法爱好者，以及高校学生阅读参考。

◆ 著　　刘新宇
责任编辑　毛倩倩
责任印制　彭志环
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市中晟雅豪印务有限公司印刷
◆ 开本：800×1000　1/16
印张：37
字数：829千字　　2017年 1 月第 1 版
印数：1－4 000册　　2017年 1 月河北第 1 次印刷

定价：99.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

站在巨人的肩上

Standing on Shoulders of Giants

iTuring.cn

序　一

我几年前曾经粗略读过这本书的英文书稿*Elementary Algorithms*，五六百页的一本算法书，是作者多年来在学术界和工业界实践中不断思考的结晶。那本英文书稿因为对几个算法问题的独到解读，成为英文世界中很多人学习算法的参考资料。这次我拿到新宇的中文版书稿《算法新解》，感觉内容更加充实丰富，文笔更加流畅，引文更加全面，结构更加紧凑，我也更加不读完不能释手。过去有人告诉我又一本中文版算法书要出版时，我通常的反应都是“又一本？”，并嗤之以鼻。毕竟中文的计算机算法书质量良莠不齐，而且存在大量照翻英文书的现象，而这本《算法新解》则是作者多年知识积累下的原创之作。更加难得的是，作者在书后给出了详细的参考文献，供读者进一步学习参考。

从计算机语言知识的角度看，这本书虽然原叫“初等”算法，但是对读者的要求并不低，书中大量使用了 C/C++、Python 和 Haskell 这几种语言。C/C++ 和 Python 作为传统而又实用的两种语言，包含了指令语言、面向对象和动态语言等几种形态，为从业人员和普通大众所广泛使用和熟悉。那么，为什么要用 Haskell 呢？孙中山曾经说过“世界潮流，浩浩荡荡，顺之则昌，逆之则亡”，计算机语言的进化过程也一样。计算机语言从最初的“让机器能理解人的交流工具”，逐渐进化到可以描述人类思考过程的更加抽象、数学上更严谨，也更加强大的函数式编程语言，而 Haskell 就是函数式编程语言的代表。如果要读懂这本书的最深刻精髓，你需要一点 Haskell 知识；如果你之前没有学过，也让这本书成为你学习一门函数式编程语言的动力之一吧！

从算法理论的角度看，这本书深入浅出。作者先花大量篇幅讲述最重要的几种数据结构和相关的处理方法，然后用大量经典而又新颖的问题来讲述计算机算法的核心问题：如何排序和搜索。这本书在前半部分介绍基本的数据结构时没有落入俗套，而是从树、堆讲起，最后介绍“并不简单”的队列，让人不禁想起张筑生先生先讲积分、后讲微分的经典名著《数学分析新讲》。我很喜欢作者介绍排序算法时对“冒泡排序”的处理：本书并不花费篇幅谈论如此低效的算法，读者可到维基百科自行花 5 分钟理解。相反，作者用了 100 多页的篇幅，通过经典而又有趣的谜题详细讲解了各种搜索算法。更难得的是，作者大量使用函数式编程语言来做范例，展示了这类语言的强大之处，这在很多中文算法书中是看不到的。

从思考解决各种有趣问题的角度看，这本书适合任何知识背景和层次的读者“享用”。算法不仅仅可以用来解决现实世界中的种种实际问题，比如通过关键词寻找网上最有用的信息，寻找最短的旅游路线来游历所有的景点，再比如无线通信中的信道编码和解码；很多美妙的算法源于人类对一些挑战自身智力的谜题的思考，比如经典的华容道问题，寻找数独的解法，

再比如用程序战胜九段围棋高手等。书中的这些谜题深刻却不枯燥，作者还给出了很多幽默的插图，它适合也值得任何学术背景的人花时间阅读和思考。这让我看出作者写这本书不但花费了很多心血，而且收获了很多欢乐；使用和发现算法是快乐的，这也是我觉得这本书特别美好之处。毕竟人类的最高阶段就是认识宇宙，了解生命起源，而更高阶段是创造和优雅地解决那些有趣的谜题！

无论作为一个需要使用各种算法的从业人员，还是一个喜欢不停思考有趣问题的人，我都觉得新宇的这本《算法新解》是一本难得的好书。如果可以，我希望能够立刻把全书上传到大脑之中。

—— 常成博士，4 数网主编

序　二

我和刘新宇认识快 10 年了。2007 年的时候我在诺基亚工作，和他所在的公司有技术项目合作，每隔几周都要一起开会，所以渐渐混熟了。那时他的职位是项目经理，虽然做着管理的工作，但是我感觉他的技术水平比大多数工程师都要好。

有一次工作上的原因，我和他一起去匈牙利出差，在飞机上我在看小说，而他拿出一本英文数学书看了一路。他说数学和编程是他的兴趣爱好，每年他都会尝试学习一门编程语言，或者了解一个新的数学领域。

不同的人有不同的追求，当我们大多数人为了房子车子，为了升职加薪而加班忙碌时，刘新宇却把 6 年的业余时间用在了写作上，而且是写一本免费的书。这本书最初的版本是用英文写的，中文版是后来翻译的。

我在大学学计算机编程的时候，看的是严蔚敏老师的《数据结构》，当时书里都是 Pascal 伪代码，老实说我没能坚持看完。

如今，编程语言相比二十多年前极大丰富了，而且产生了很多种编程范式。《算法新解》的一大特色就是提供了多种编程语言的算法实现代码，并且充分利用了各种语言特性。某些算法用函数式编程语言实现会特别简洁，比如用 Haskell 实现的快速排序只有几行代码。

现在，软件开发行业的分工已经非常细了，有人专门负责编写算法提供封装好的库，也许绝大多数程序员很少需要自己实现某些算法。我们学习算法和数据结构，不一定就要实际去编写算法代码，也可以当作一种了解程序底层运作的方式，而这有助于更好地解决问题和优化程序。

阅读算法书是会有点枯燥的，先通过阅读对算法原理有个基本的概念，然后读者可以把它当作手边的参考书，在实际遇到相关问题时拿出来仔细阅读，结合实际场景可以有更深刻的理解。

这本书是按照教材的风格编写的，由浅入深，每个算法从实际应用场景出发，阐述数学原理，再给出伪代码，并且提供实际可以运行的实现。最后，作者还给出了供课后思考练习的习题。我希望这本书有机会成为计算机教学中的基础教材之一。

这本书采用了一种新颖的发布方式，它像一个开源软件项目一样，作者像管理源代码一样管理书的内容，允许读者参与其中贡献思路代码，帮助修改错误，以及在其整个生命周期里持续改进。

在软件和互联网高速发展的这些年里，我们程序员不断开发功能满足需求，但我们也应该为纯技术保留一份好奇、一份执着，刘新宇多年的坚持着实令我感动。虽然算法都是冷冰

冰的符号公式，但是从书里的文字、插图和代码里，我们可以感受到作者的技术情怀。

—— 姚冬，YY 直播架构师

前　言

算法有用吗

“算法有用吗?”经常有人问我这个问题。很多人在工作中根本不用算法。偶尔碰到的时候，也不过是使用一些实现好的库。例如，C++ 标准模板库（STL）中有现成的排序、查找函数；常用的数据结构，如向量（vector）、队列（queue）、集合（set）也都实现好了。日常工作中了解如何使用这些库似乎就足够了。

但是，算法在解决一些“有趣”的问题时会起到关键作用。不过，这些问题本身的价值却是仁者见仁、智者见智。

让我们用例子来说话吧。接下来的两道题目，即使是初学编程的人，应该也很容易解决。

最小可用 ID，算法的威力

这道题目来自 Richard Bird 所著书中的第 1 章 [1]。现代社会中，有很多服务依赖一种称为 ID 的概念。例如身份证就是一种 ID，银行账户也是一种 ID，电话号码本质上也是一种 ID。假设我们使用非负整数作为某个系统的 ID，所有用户都由一个 ID 唯一确定。任何时间，这个系统中的有些 ID 处于使用中的状态，有些 ID 则可以分配给新用户。现在的问题是，怎样才能找到最小的可分配 ID 呢？例如下面是当前正在使用的 ID：

```
[18, 4, 8, 9, 16, 1, 14, 7, 19, 3, 0, 5, 2, 11, 6]
```

最小的可分配 ID，也就是不在这个列表中的最小非负整数，即 10。这个题目看上去如此简单，我们可以立即写出下面的解法：

```
1: function Min-Free(A)
2:     x ← 0
3:     loop
4:         if x ∉ A then
5:             return x
6:         else
7:             x ← x + 1
```

其中符号 $\notin$ 的实现如下：

```
1: function '∉'(x, X)
2:     for i ← 1 to |X| do
3:         if x = X[i] then
4:             return False
5:     return True
```

有些编程语言内置了这一线性查找的实现，例如 Python。我们可以直接将这一解法翻译成下面的程序：

```
def brute_force(lst):
    i = 0
    while True:
        if i not in lst:
            return i
        i = i + 1
```

但是这道题目仅仅是看上去简单。在存储了几百万个 ID 的大型系统中，这个方法的性能很差。对于一个长度为 n 的 ID 列表，它需要 $O(n^2)$ 的时间才能找到最小的可分配 ID。在我的计算机上（双核 2.10 GHz 处理器，2 GB 内存），使用这一方法的 C 语言程序平均要 5.4 s 才能在 10 万个 ID 中找到答案。当 ID 的数量上升到 100 万时，平均用时则长达 8 min。

改进一

改进这一解法的关键基于这一事实：对于任何 n 个非负整数 $x_1, x_2, \cdots, x_n$，如果存在小于 n 的可用整数，必然存在某个 x_i 不在 $[0, n)$ 范围内。否则这些整数一定是 $0, 1, \cdots, n-1$ 的某个排列，这种情况下，最小的可用整数是 n。于是我们有如下结论：

$$minfree(x_1, x_2, \cdots, x_n) \leqslant n \tag{1}$$

根据这一结论，我们可以用一个长度为 $n+1$ 的数组，来标记区间 $[0, n]$ 内的某个整数是否可用：

```
1: function Min-Free(A)
2:     F ← [False, False, ···, False] where |F| = n + 1
3:     for ∀x ∈ A do
4:         if x < n then
5:             F[x] ← True
6:     for i ← [0, n] do
7:         if F[i] = False then
8:             return i
```

其中第 2 行将标志数组中的所有值初始化为`False`，这一步骤需要 $O(n)$ 的时间。接着我们遍历 A 中的所有元素，只要小于 n，就将相应的标记置为`True`，这一过程也需要 $O(n)$ 的时间。最后我们线性查找标志数组中第一个值为`False`的位置。整个算法的性能是线性时间

$O(n)$。注意，我们使用了 $n+1$ 个标志，而不是 n 个标志。这样无需额外处理，就可以应对 $sorted(A) = [0, 1, 2, \cdots, n-1]$ 的特殊情况。

虽然这个方法只需要线性时间，但是它需要 $O(n)$ 的空间来存储标志。

这一方法比之前的暴力解法快很多。在我的计算机上，相应的 Python 程序平均只用 0.02 s，就可以在 10 万个整数中找到答案。

我们还可以继续优化。每次查找，我们都要申请长度为 $n+1$ 的数组；查找结束后，这个数组又被释放掉了。反复的申请和释放会占用不少时间，因此我们可以预先准备好足够长的数组，然后每次查找都复用它。另外，我们可以使用二进制的位来保存标志，这样能节约不少空间。下面的 C 语言程序实现了这两点小改进：

```
#define N 1000000 // 100万
#define WORD_LENGTH sizeof(int) * 8

void setbit(unsigned int* bits, unsigned int i)  {
    bits[i / WORD_LENGTH] |= 1<<(i % WORD_LENGTH);
}

int testbit(unsigned int* bits, unsigned int i) {
    return bits[i/WORD_LENGTH] & (1<<(i % WORD_LENGTH));
}

unsigned int bits[N/WORD_LENGTH+1];

int min_free(int* xs, int n) {
    int i, len = N/WORD_LENGTH+1;
    for(i=0; i<len; ++i)
        bits[i]=0;
    for(i=0; i<n; ++i)
        if(xs[i]<n)
            setbit(bits, xs[i]);
    for(i=0; i<=n; ++i)
        if(!testbit(bits, i))
            return i;
}
```

在我的计算机上，这段 C 程序处理 100 万个整数，平均用时仅仅 0.023 s。最后一个`for`循环还能进一步改进如下，但都是一些微调了：

```
for(i=0; ; ++i)
```

```
if(~bits[i] !=0)
    for(j=0; ; ++j)
        if(!testbit(bits, i*WORD_LENGTH+j))
            return i*WORD_LENGTH+j;
```

改进二：分而治之

我们以空间上的消耗为代价做了速度上的改进。由于维护了一个长度为 $n+1$ 的标志数组，当 n 很大时，空间上的性能就成了新的瓶颈。

分而治之的典型策略是将问题分解为若干规模较小的子问题，然后逐步解决它们以得到最终的结果。

我们可以将所有满足 $x_i \leqslant \lfloor n/2 \rfloor$ 的整数放入一个子序列 A'，并将剩余的整数放入另一个序列 A''。根据式 1，如果序列 A' 的长度正好是 $\lfloor n/2 \rfloor$，这说明前一半的整数已经“满了”，最小的可用整数一定可以在 A'' 中递归地找到。否则，最小的可用整数可以在 A' 中找到。总之，通过这一划分，问题的规模减小了。

需要注意的是，当我们在子序列 A'' 中递归查找时，边界情况发生了一些变化：不再是从 0 开始寻找最小可用整数，查找的下界变成了$\lfloor n/2 \rfloor + 1$。因此，我们的算法应定义为 $minfree(A, l, u)$，其中 l 是下界，u 是上界。

递归结束的边界条件是待查找的序列为空，此时我们返回下界作为结果即可。

根据上述思路，分而治之的解法可以形式化为一个函数：

$$minfree(A) = search(A, 0, |A| - 1)$$

$$search(A, l, u) = \begin{cases} l & : \quad A = \phi \\ search(A'', m+1, u) & : \quad |A'| = m - l + 1 \\ search(A', l, m) & : \quad \text{其他} \end{cases}$$

其中有：

$$m = \left\lfloor \frac{l+u}{2} \right\rfloor$$
$$A' = \{\forall x \in A \wedge x \leqslant m\}$$
$$A'' = \{\forall x \in A \wedge x > m\}$$

这一方法并不需要额外的空间 [①]。每次调用我们都需要进行 $O(|A|)$ 次比较，来划分出子序列 A' 和 A''，之后问题的规模减半。所以这个算法用时为 $T(n) = T(n/2) + O(n)$，化简可知其结果为 $O(n)$。我们也可以这样分析其复杂度：第一次需要 $O(n)$ 次比较来划分子

① 有人认为需要 $O(\lg n)$ 的栈空间来做递归调用的簿记（book-keeping）。我们稍后会看到，这一调用实际上是尾递归，有些编译器（例如 GCC）可以通过-O2选项消除递归。我们也可以手工将递归转换为迭代。

序列 A' 和 A''，第二次仅需要比较 $O(n/2)$ 次，第三次需要比较 $O(n/4)$ 次……总时间为 $O(n+n/2+n/4+\cdots)=O(2n)=O(n)$。

某些函数式编程语言（例如 Haskell）将划分一个序列作为库函数提供。下面的代码实现了分而治之的算法：

```
import Data.List

minFree xs = bsearch xs 0 (length xs - 1)

bsearch xs l u | xs == [] = l
               | length as == m - l + 1 = bsearch bs (m+1) u
               | otherwise = bsearch as l m
    where
        m = (l + u) `div` 2
        (as, bs) = partition (<=m) xs
```

简洁与性能：鱼和熊掌

使用命令式编程语言的读者可能会担心这种实现的性能。对于最小的可分配 ID 问题，递归的深度为 $O(\lg n)$，于是调用栈的大小也是 $O(\lg n)$，因此空间复杂度并不能忽略。实际上，我们可以将递归转换为迭代来避免空间上的占用[①]，比如下面的 C 语言程序：

```
int min_free(int* xs, int n) {
    int l=0;
    int u=n-1;
    while(n) {
        int m = (l + u) / 2;
        int right, left = 0;
        for(right = 0; right < n; ++ right)
            if(xs[right] <= m) {
                swap(xs[left], xs[right]);
                ++left;
            }
        if(left == m - l + 1) {
            xs = xs + left;
            n  = n - left;
            l  = m+1;
```

① 由于我们的函数是尾递归形式，大多数函数式编程语言会自动优化尾递归函数。

```
        } else {
            n = left;
            u = m;
        }
    }
    return l;
}
```

这段程序使用了类似“快速排序”中的分割方法，将数组中的元素分成了两部分。所有left之前的元素都不大于m，而所有left和right之间的元素都大于m，如图 1 所示。

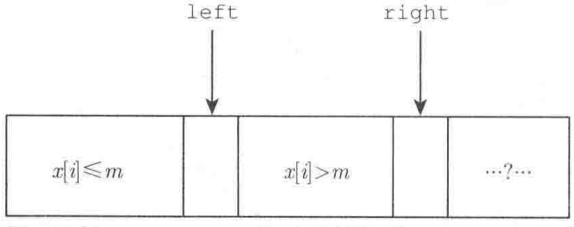


图 1　数组划分的过程。所有 $0 \leqslant i <$left的元素都满足 $x[i] \leqslant m$，所有left$\leqslant i <$ right的元素满足 $x[i] > m$，剩余的元素尚未处理

这一程序运行快并且不需要额外的栈空间，但是和前面的 Haskell 程序比起来，并不那么直观、简洁，需要仔细阅读。有时我们需要在简洁与性能之间进行权衡。

丑数：数据结构的威力

如果说最小可分配 ID 问题还有一些应用价值，那么接下来的这个问题就纯粹只是“有趣”了：寻找第 1500 个“丑数”。所谓“丑数”，就是只含有 2、3 或 5 这 3 个素因子的自然数[①]。前 3 个丑数按照定义分别是 2、3 和 5；数字 $60 = 2^2 3^1 5^1$ 是第 25 个丑数；数字 $21 = 2^0 3^1 7^1$ 由于含有素因子 7，所以不是丑数。前 10 个丑数如下：

2, 3, 4, 5, 6, 8, 9, 10, 12, 15

如果我们认为 $1 = 2^0 3^0 5^0$ 也是一个合法的丑数，则 1 就是第一个丑数。

暴力解法

这道题目看起来并不复杂，我们可以从 1 开始逐一检查所有自然数，对于每个整数，用除法把所有的 2、3 和 5 的因子都去掉，如果结果是 1，则找到了一个丑数，当遇到第 $n = 1500$ 个丑数时就找到答案了。其伪代码如下：

① 丑数也叫正规数、汉明数、5-光滑数，在密码学、音乐理论中有着重要的应用，详见 https://en.wikipedia.org/wiki/Regular-number。

```
1: function Get-Number(n)
2:     x ← 1
3:     i ← 0
4:     loop
5:         if Valid?(x) then
6:             i ← i + 1
7:             if i = n then
8:                 return x
9:         x ← x + 1

10: function Valid?(x)
11:     while x mod 2 = 0 do
12:         x ← x/2
13:     while x mod 3 = 0 do
14:         x ← x/3
15:     while x mod 5 = 0 do
16:         x ← x/5
17:     if x = 1 then
18:         return True
19:     else
20:         return False
```

这一暴力解法对于较小的 n 没有问题。但是根据这个方法编写的 C 语言程序，在我的计算机上耗时 40.39 s 才找到第 1500 个丑数 (859 963 392)。当试图求第 15 000 个丑数时，程序运行了 10 min 也没能找到答案，我只好把它强行停止。

改进一：构造性解法

在上面的暴力解法中，取模运算和除法运算很耗时 [2]，并且这些运算循环执行了很多次。我们可以转换一下思路，不再检查一个数是否是仅含有 2、3 或 5 这 3 个素因子的自然数，而是从这 3 个素因子中构造需要的整数。

我们从 1 开始，分别将其乘以 2 或 3 或 5 来生成整数，这样问题就变成了如何依次生成丑数。我们可以使用队列这种数据结构来解决问题。

队列从一侧放入元素，然后从另一侧取出元素，所以先放入的元素会先被取出。这一特性称为 FIFO（First In First Out，先进先出）。

我们的思路是先把 1 作为唯一的元素放入队列，然后不断从队列的另一侧取出元素，分别乘以 2、3 和 5，这样就得到了 3 个新的元素，然后把它们按照大小顺序放入队列。注意，这样产生的整数有可能已经在队列中存在了，这种情况下需要丢弃重复产生的元素。另外，新产生的整数还可能小于队列尾部的某些元素，所以在插入时需要保持它们在队列中的大小顺序。图 2 描述了这一思路的操作步骤。

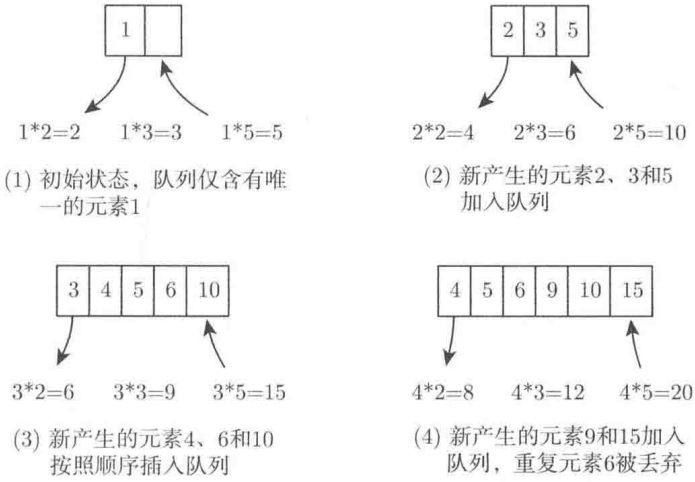


图 2　使用队列依次生成丑数的前 4 个步骤

这一思路的算法实现如下：

```
1: function Get-Number(n)
2:     Q ← NIL
3:     Enqueue(Q, 1)
4:     while n > 0 do
5:         x ← Dequeue(Q)
6:         Unique-Enqueue(Q, 2x)
7:         Unique-Enqueue(Q, 3x)
8:         Unique-Enqueue(Q, 5x)
9:         n ← n − 1
10:    return x

11: function Unique-Enqueue(Q, x)
12:    i ← 0
13:    while i < |Q| ∧ Q[i] < x do
14:        i ← i + 1
15:    if i < |Q| ∧ x = Q[i] then
16:        return
17:    Insert(Q, i, x)
```

在将元素插入队列时，算法需要 $O(|Q|)$ 的时间找到合适位置；如果元素已经存在，则直接返回。

粗略估计，队列的长度会随着 n 增加（每取出一个元素会插入最多 3 个新元素，增加的比率 $\leqslant 2$），所以总运行时间为 $O(1+2+3+\cdots+n)=O(n^2)$。

图 3 显示了队列的访问次数和 n 之间的关系，这些点连成了二次曲线，反映了算法的复杂度，即 $O(n^2)$。

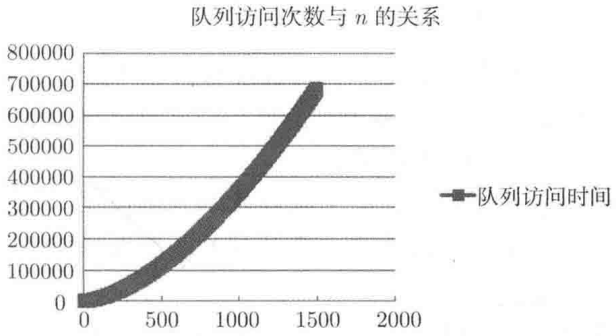


图 3　队列访问次数和 n 的关系

依照此方法实现的 C 语言程序仅用时 0.016 s 就输出了正确答案 859 963 392，比暴力解法快了 2500 多倍。

这一解法也可以用递归的方式给出，即令 X 为所有仅含有素因子 2、3 或 5 的整数的无穷序列。下面的等式给出了一个有趣的关系：

$$X = \{1\} \cup \{2x : \forall x \in X\} \cup \{3x : \forall x \in X\} \cup \{5x : \forall x \in X\} \tag{2}$$

其中符号 $\cup$ 表示去除重复元素并保持大小顺序。若 $X=\{x_1, x_2, x_3, \cdots\}$，$Y=\{y_1, y_2, y_3, \cdots\}$，$X' = \{x_2, x_3, \cdots\}$，$Y' = \{y_2, y_3, \cdots\}$，我们可以定义 $\cup$ 如下：

$$X \cup Y = \begin{cases} X & : \quad Y = \phi \\ Y & : \quad X = \phi \\ \{x_1, X' \cup Y\} & : \quad x_1 < y_1 \\ \{x_1, X' \cup Y'\} & : \quad x_1 = y_1 \\ \{y_1, X \cup Y'\} & : \quad x_1 > y_1 \end{cases}$$

在支持惰性求值的函数式编程语言（例如 Haskell）中，上述无穷序列及函数可以定义为如下代码：

```
ns = 1:merge (map (*2) ns) (merge (map (*3) ns) (map (*5) ns))

merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x <y = x : merge xs (y:ys)
                    | x ==y = x : merge xs ys
                    | otherwise = y : merge (x:xs) ys
```

通过求ns !! (n-1)，我们可以得到第 1500 个丑数：

```
>ns !! (1500-1)
859963392
```