求最大公约数算法

Catalog

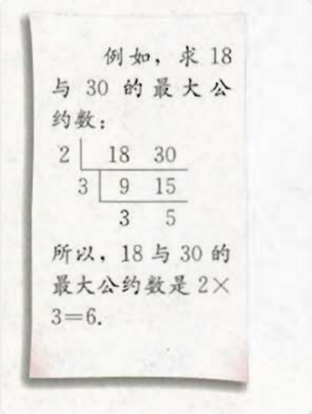
New Words

- **divisor [dɪ'vaɪzɚ] --n.除数; 因数; 因子**
    - Greatest common divisor. 最大公约数
    - In 4 divided by 2, the number 2 is the divisor and 4 is the dividend. 在 4 被 2 除中, 2 是除数, 4 是被除数.

Content

0. 前置知识 `辗转相除法` 和 `更相减损术`

- 前置知识: 来自高中数学必修 3
- 在小学, 我们学过求 *两个正整数的最大公约数* 的方法: 先用两个数公有的质因数连续去除, 一直除到所得的商是互质数为止, 例如:


例如，求 18 与 30 的最大公约数：

$$\begin{array}{r|rr} 2 & 18 & 30 \\ \hline 3 & 9 & 15 \\ \hline & 3 & 5 \end{array}$$

所以，18 与 30 的最大公约数是 2×3=6.

但是, 当两个数公有的质因数较大时(如 8251 与 6105), 使用上述方法求最大公约数就比较困难. 下面介绍 2 种古老而有效的算法.

0.1 `辗转相除法`

- 这种算法是由欧几里得在公元前 300 年左右首先提出的, 因而又叫 `欧几里得算法` .

  Added 辗转相除法的定理: 两个正整数 `a` 和 `b` (a > b), 它们的最大公约数等于 `a` 除以 `b` 的余数 `c` 和 `b` 之间的最大公约数.

  例如: 求 10 和 25 之间的最大公约数 -- A: 25 除以 10 商 2 余 5, 那么 10 和 25 的最大公约数等同于 10 和 5 的最大公约数, 即 5.

  例如, 用辗转相除法求 `8251` 与 `6105` 的最大公约数, 我们可以考虑用两数中较大的数除以较小的数, 求得商和余数:

  ```
  8251 = 6105 x 1 + 2146.
  ```

  由此可得, 6105 与 2146 的公约数也是 8251 与 6105 的公约数, 反过来, 8251 与 6105 的公约数也是 6105 与 2146 的公约数, 所以它们的最大公约数相等.

  对 6105 与 2146 重复上述步骤

  ```
  6105 = 2146 x 2 + 1813.
  ```

  同理, 2146 与 1813 的最大公约数也是 6105 与 2146 的最大公约数. 继续重复上述步骤:

  ```
  2146 = 1813 x 1 + 33
  1813 = 333 x 5 + 148,
  333 = 148 x 2 + 37,
  148 = 37 x 4.
  ```

  最后的除数 37 是 148 和 37 的最大公约数(tip: 37 后无余数), 也就是 8251 与 6105 的最大公约数这就是辗转相除法. 由除法的性质可以知道, 对于任意两个正整数, 上述除法步骤总可以在有限步之后完成, 从而总可以用辗转相除法求出两个正整数的最大公约数.

0.2 `更相减损术`

- `《九章算术》` 是中国古代的数学专著, 其中的 `更相减损术` 也可以用来求 2 个数的最大公约数, 即 "可半者半之, 不可半者, 副置分母, 子之数, 以少减多, 更相减损, 求其等也, 以等数约之."

  翻译为现在语言如下:

- 第 (1) 步: 任意给定 2 个正整数, 判断他们是否都是偶数, 若是, 用 2 约减; 若不是, 执行第 (2) 步.
- 第 (2) 步: 以较大的数减去较小的数, 接着把所得的差与较小的数比较, 并以大数减小数. 继续这个操作, 直到所得的数相等为止[(1)], 则这个数(等数)或这个数与约简的数的乘积就是所求的最大公约数.
  - (1): 这里说的两数相等, 我们根据下面的例子来说, 就拿最后一行 14 - 7 = 7 来说, **即: "较大数" - "较小数" 得到的结果, 和较小数相等** .

下面我们用一个例子来说明这个算法.

例(1): 用 `更相减损术` 求 98 与 63 的最大公约数.
解: 由于 63 不是偶数, 把 98 和 63 以大数减小数, 并辗转相减, 如下所示:

```
98 - 63 = 35
63 - 35 = 28
35 - 28 = 7
28 - 7 = 21
21 - 7 = 14
14 - 7 = 7
```

所以, 98 与 63 的最大公约数等于 7.

Warning: 例(2): 此处要说一个问题, 就是根据上面翻译的结果, 我们来试着求一下 64 和 6 的最大公约数:

- 根据步骤 (1): $64 \div 2 = 32$, $6 \div 2 = 3$;
- 接着根据步骤 (2):

  ```
  32 - 3 = 29
  29 - 3 = 26
  26 - 3 = 23
  23 - 3 = 20
  20 - 3 = 17
  17 - 3 = 14
  14 - 3 = 11
  11 - 3 = 8
  8 - 3 = 5
  5 - 3 = 2
  3 - 2 = 1
  2 - 1 = 1
  ```

  结果为 1 , 看到问题了吗? 64 和 6 的最大公约数实际上为 **2**, 所以我才说按照上面的翻译求出的结果是有问题的, 此时, 我们应该把步骤 (1) 的除 2 再乘回来, $1 \times 2 = 2$, 这样答案才是正确的.

1. 使用 JavaScript 实现求最大公约数算法

根据前置知识中的讲解, 我们该怎么用代码实现一个最大公约数算法呢?

1.1 普通方法求两数的最大公约数

- 我们先看普通的 "用两个数公有的质因数连续去除, 一直除到所得的商是互质数为止(上图)" 的方法如何用代码来实现. 下面是代码和注解:

```
// - greatest common divisor 最大公约数
function gcd(a, b) {
    const bigger = a > b ? a : b;           // {1-1}
    const smaller = a < b ? a : b;          // {1-2}
    if (bigger % smaller === 0) {           // {1-3}
        return smaller;
    }
    let divisor = 1;                        // {1-4}
    let i = 2;                              // {1-5}
    let length = smaller / 2;               // {1-6}
    for (; i <= length; i++) {              // {1-7}
        if (a % i === 0 && b % i === 0) {   // {1-8}
            divisor = i;                    // {1-9}
        }
    }
    return divisor;
}

console.log('gcd(18, 30):', gcd(18, 30));   // 6
console.log('gcd(8251, 6105):', gcd(8251, 6105)); // 37
```

代码分析:

  - `行{1-1}` 和 `行{1-2}` 先判断传入的数 a, b 谁大谁小;
  - `行{1-3}` 首先做一个判断, 如果大数(bigger)是小数(smaller) 的倍数, 那直接返回 smaller 即可. 比如: "9 和 27" 或 "8 和 64" 它们的最大公因数皆为两数中较小的一个.
  - `行{1-4}` 初始化一个变量 divisor(因数) 用于保存将来函数返回的最大公约数;
  - `行{1-5}` 声明 for 循环需要的索引值 i, 两个数的最大公因数不会小于 1, 因此初始化 i = 2.
  - `行{1-6}` 我们把较小数值大小的一半设置为 for 语句的执行次数. ("Hint:" 这里为什么设置为较小数值的一半仍不理解, 路过的小伙伴, 明白的可以告知俺, 谢谢.)
  - `行{1-7}` , `行{1-8}` , `行{1-9}` 使用暴力枚举的方法, 试图寻找到一个合适的整数 i, 看看这个整数能否被 a 和 b 同时整除. 这个整数 i 从 2 开始循环累加, 一直累加到 a 和 b 中较小参数的一半位置. 循环结束后, 上一次(即循环结束的前一次)寻找到的能被两个整数整除的最大 i 值, 就是两数的最大公约数.
    - **Added:** 这里可能会有小伙伴问为什么 i 就成了最小公约数了? 我把文章的第一张图稍微转换一下, 你便会明白:

      ```
        2 |  18   30                   6 |  18   30
          -----------                    -----------
          3 | 9   15                         3    5
            ---------
               3    5
      ```

      2 x 3 = 6, 是不是和第二个直接使用 6 的结果是一样的?

到这里, 第一种最笨求两数的最大公约数算法是完成了, 但是在工作环境中, 这个算法的效率却不行, 想想看, 比如我传入 gcd(10000, 10001), 用这个方法需要循环 10000 / 2 - 1 = 4999 次! 如果更大的数, 效率可想而知了……

1.2 `辗转相除法`

- 接下来第二种方法是前置知识中的 `欧几里得算法` , 此算法大大缩小了求两数的最小公约数的计算次数, 根据前置知识的讲解, 此处先给出代码实现:

```
// - 辗转相除法
// - 给出 (8251, 6105) 辗转相除法的步骤作为下面代码的参考:
//   8251 = 6105 x 1 + 2146.
//   6105 = 2146 x 2 + 1813.
//   2146 = 1813 x 1 + 33
//   1813 = 333 x 5 + 148
//   333 = 148 x 2 + 37
//   148 = 37 x 4.
function gcd (a, b) {
    let result = 1;             // {2-1}
    if (a > b) {                // {2-2}
        result = divide(a, b)   // {2-3}
    }
    else {
        result = divide(b, a)   // {2-4}
    }
    return result;
}
function divide(a, b) {         // {2-5}
    if (a % b === 0) {          // {2-6}
        return b;               // {2-7}
    }
    else {
        return gcd(b, a%b)      // {2-8}
    }
}
// console.log('gcd(8251, 6105):', gcd(8251, 6105)); // 37
```

代码分析: 代码相对简单, 暂略.

1.3 使用 `更相减损术` 方实现求两数的最大公约数:

- 因为 `辗转相除法` , 当 2 个数很大时, 对 $a\%b$ 的求模运算的性能会比较低. 所以我们改为使用 `更相减损术` .

  我们根据上面对 `更相减损术` 的定义 (1) : 先判断给定的 2 个数是不是都是 `偶数` , 如果是先用 2 约减. 那么现在就面临第一个问题.

  如何在编程语言实现判断一个数是 奇数 还是 偶数 ?

  A: 在编程中一般有 2 种方式判断一个数是不是偶数.

  - (1) 使用 `按位与(&)` 操作符. JS 代码如下:

```
// - 在 JS 中可以使用 `按位与(&)`. 判断一个数字是奇数还是偶数
```

```
// - `按位与(&)`: 按位与操作只在两个数值的对应位都是 1 时才返回 1,
//   任何一位是 0, 结果都是 0.
function isEven(num) {
    console.log('num & 1:', num & 1);
    // - 请一定记得在 (num & 1) 外围加括号.
    if ((num & 1) === 0) {
        return true;
    }
    return false;
}
console.log(isEven(6));     // true
console.log(isEven(7));     // false
console.log("Boolean(0):", Boolean(0));     // false
```

- (2) 求余数(%). 求余数的写法是平时使用最多的:

```
function isEven(num) {
    if (num % 2 === 0) {
        return true
    }
    return false;
}
```

现在解决了判断一个数奇偶的问题, 我们给出使用 `更相减损术` 求两数最大公约数的算法实现:

```
(function() {
    // - 更相减损术

    let num64 = 64;
    // - 此注释讲解来自: 《JS高程》
    // - 左移(`<<`): 这个操作符会将数值的所有位向左移动指定的位数.
    //   例如: 如果将数值 2 (二进制码为 10) 向左移动 5 位,
    //   结果就是 64 (二进制码为 1000000).
    // - 有符号的右移(`>>`): 这个操作符会将数值向右移动,
    //   但保留符号位(即正负号标记). 有符号的右移操作与左移操作恰好相反,
    //   即如果将 64 向右移动 5 位, 结果将变回 2.
    // console.log(num64 >> 1);    // 32
    // console.log(32 >> 1);       // 16
    // console.log(32 << 1);       // 64
    // console.log(num64 << 1);    // 128

    var i = 0;
    function gcd(a, b) {
        let twice = 1;
        // - (1) 如果 2 数相等, 就返回其一
        if (a === b) {
            return a;
        }

        // - (2) 如果较大的数除以较小的数等于 0, 就返回较小的数.
        if (a > b && (a % b) === 0) {
            return b;
```

```
        }
        if (b > a && (b % a) === 0){
            return a;
        }

        // - (3) 如果较大数和较小数都是偶数, 就同时除 2
        if ((a & 1) === 0 && (b & 1) === 0) {
            a = a >> 1;
            b = b >> 1;
            if (a < b) {
                twice = gcd(b - a, a);
            }
            else {
                twice = gcd(a - b, b);
            }
            return twice * 2;
        }
        // console.log('i++: ', i++);

        if (a < b) {
            return gcd(b - a, a);
        }
        else {
            return gcd(a - b, b);
        }
    }
    console.log('更相减损术 gcd(8251, 6105):', gcd(8251, 6105)); // 37
    // console.log("gcd(64, 8)", gcd(64, 8));  // 8
    console.log("更相减损术 gcd(64, 6)", gcd(64, 6))   // 2
    // console.log(gcd(10001, 1));  //
})();
```

更相减损术的缺点: 更相减损术依靠两数求差的方式来递归, 运算的次数远大于辗转相除法的取模方式. 所以更相减损术是不稳定的算法, 当两数相差悬殊时, 比如计算 10000 和 1, 就要递归 9999 次.

下面我们结合 `辗转相除法` 和 `更相减损术` 来给出较优解决方案: 接 (1.4)

1.4 求两数的最大公约数的较优方法: 组合使用 `辗转相除法` 和 `更相减损术`

- 众所周知, 移位(位移)运算的性能非常快. 对于给定的正整数 a 和 b, 不难得到如下的结论:
    - (1) 当 a 和 b 均为偶数时:

      `gcd(a, b)` = `2 * gcd(a/2, b/2)` = `2 * gcd(a >> 1, b>> 1)`
    - (2) 当 a 为偶数, b 为奇数:

      `gcd(a, b)` = `gcd(a/2, b)` = `gcd(a >> 1, b)`
    - (3) 当 a 为奇数, b 为偶数:

      `gcd(a, b)` = `gcd(a, b/2)` = `gcd(a, b >> 1)`

当 a 和 b 均为奇数, 利用 `更相减损术` 运算一次(`gcd(a, b)`), 运算之后 a - b 必然是偶数(tip: 奇数减奇数必然是偶数), 又可以继续进行位移运算.

例如: 计算 10 和 25 的最大公约数的步骤如下: (Hint: 步骤有省略, 代码实现中更详细.)

- (1) 判断 10 为偶数后, 向右(tip: 会把 10 先转换成二进制数)位移 1 位后变为 5, 此时可转换成求 5 和 25 的最大公约数.
- (2) 先判断 5 和 25 是不是都是奇数, 如果是, 调用 `更相减损术` 25 - 5 = 20, 此时便转换成求 5 和 20 的最大公约数.
- (3) 判断 20 为偶数后, 向右位移 1 位后变为 10, 此时变成求 5 和 10 的最大公约数.
- (4) 判断 10 为偶数后, 向右位移 1 位后变为 5, 此时变成求 5 和 5 的最大公约数.
- (5) 判断 5 和 25 都是奇数, 再次调用 `更相减损术` , 因为两数相等, 所以最大公约数为 5.

下面给出完整的 JS 代码实现:

```
// - 组合使用 `辗转相除法` 和 `更相减损术`
(function() {
    function gcd(a, b) {
        // - (1) 如果 2 数相等, 就返回其一
        if (a === b) {
            return a;
      }

        // - (2) 如果较大的数除以较小的数等于 0, 就返回较小的数.
        if (a > b && (a % b) === 0) {
            return b;
        }
        if (b > a && (b % a) === 0){
            return a;
      }

        // - (3)
        if (a < b) {
            return gcd(b, a);
        }
        else {
            // - (a) 如果两数都为偶数
            if ((a & 1) === 0  && (b & 1) === 0){
                return gcd(a >> 1, b >> 1) << 1;        // {4-1}
            }
            // - (b) 如果较大数为偶数, 向右有符号位移 1 位
            else if ((a & 1) === 0 && (b & 1) === 1) {
                return gcd(a >> 1, b);
            }
            // - (c) 如果较小数为偶数, 向右有符号位移 1 位
            else if ((a & 1) === 1 && (b & 1) === 0) {
                return gcd(a, b >> 1)
            }
            // - (d) 如果两数都为奇数, 调用一次 `更相减损术`(即:自身)
            else {
                return gcd(a - b, b);
            }
```

```
        }
    }
    console.log("gcd(64, 8)", gcd(64, 8));  // 8
    console.log("gcd(64, 6)", gcd(64, 6))   // 2
    console.log('gcd(8251, 6105)', gcd(8251, 6105));    // 37
})();
```

代码注释:

- 可能会有小伙伴对 `行{4-1}` 的代码有疑问, 这里添加一下注释: 拿示例求 64 和 6 的最小公约数来说, 它们两个数都是偶数, 也就是符合 `行{4-1}` 的情形, 结合示例 (1.3) 我们知道, 如果在两个数都是偶数的情况下, 当使用 `更相减损术` 求得最大公约数的时候, 最后的结果是需要再乘以 2 的, 那么如果使用按位操作符怎么实现把结果乘 2 呢? 答案就是使用左移操作符(`<<`). 在本示例中, 求 64 和 6 的最大公约数, 在执行 n 次 `行{4-1}` 后得到 b = 1, 那么此时 `b << 1` = 2. 结果返回 2.

  还要一点要提示的是, 在执行完 `b << 1` = 2 后, `gcd()` 函数并不会立即结束, 而是会在执行栈中执行回退操作, 原因是 `gcd(a >> 1, b >> 1) << 1` 这种写法的形式形成了闭包, 所以我们需要执行倒退操作, 把之前执行的 n 步操执行 `n--` , 最后到达 1 时, 结束.

  关于执行栈更详细的解说请见: `JS-book-learning/《深入理解JavaScript系列》--汤姆大叔/11-执行上下文/11-执行上下文(环境).md`