C++ 算法讲义：高精度除法

一、 引言

1.1 问题引入

我们已经掌握了如何让计算机处理大整数的加、减、乘法。现在迎来了高精度运算中逻辑最复杂的一环：**高精度除法。**

思考一个场景：处理宏观经济数据时，计算一个国家千万亿级别的 GDP 除以数亿人口的人均产值；或是密码学中，对两个极长的大素数进行除法和取模运算。

例如：`1,000,000,000,000,000,000 ÷ 123,456,789 = ? ... ?`

C++ 内置的 `/` 和 `%` 运算符在面对这种规模的数字时会因溢出而失效。我们必须设计一种算法，教会计算机像人类一样进行“大数 ÷ 大数”的精确计算。

1.2 价值阐述

高精度除法是高精度运算系列中综合性最强的算法。因为它的实现不仅需要严密的逻辑，还深度依赖之前学过的高精度减法和比较算法。并且它是实现更高级数学算法（如大数求最大公约数、快速幂取模、大数开方等）的基础工具。

二、 核心思想：模拟竖式除法

为了让计算机“学会”除法，我们回顾人类是如何手工计算的。以 `859 ÷ 21` 为例：

```
      4  0   <-- 商 (Quotient)
    ______
21 ) 8 5 9   <-- 被除数 (Dividend)
     8 4     <-- 21 * 4
     ---
       1 9   <-- 85 - 84 = 1，然后落下 '9'，组成新余数 19
       0 0   <-- 19 不够除 21 (商 0)
       ---
       1 9   <-- 最终余数 (Remainder)
```

竖式除法的动作分解：

1. **取位**：从被除数的最高位开始，依次向右提取一个“片段”。
2. **试商**：判断当前“片段”里最多包含几个除数。得出的数字即为商的一位。
3. **相减**：从当前片段中减去 `除数 × 商`。
4. **落位** ：将被除数的下一位“落”下，拼接到相减后的余数末尾，形成下一轮的新片段。
5. **循环往复**：重复上述过程，直到被除数的所有位都被处理完毕。

如何让计算机进行“试商”？

人类可以通过直觉快速估算出 `85` 里面有 `4` 个 `21`。但计算机需要精确的指令告诉它怎么做。
最直接的“试商”方法是**连减法**：
在一个内部循环中，不断地从 `当前片段` 中减去 `除数`，并用一个计数器记录减成功的次数。直到 `当前片段` 小于 `除数` 为止。**这个计数器的值，就是当前这一位的商。**

算法核心逻辑总结：
从被除数 `A` 的最高位起，逐位构建 `当前余数 R`。每一步，将 `A` 的新一位字符拼到 `R` 的末尾，然后用一个内部 `while` 循环，统计 `R` 能够减去多少次除数 `B`。减去的次数存入最终的商，减完后剩下的值作为下一轮的初始余数。

三、 实例与图解演示

具体案例：用算法逻辑推演 `859 ÷ 21`。

前置要求：我们假定已经拥有了两个辅助函数：`isGreaterOrEqual(A, B)`（判断大数A是否大于等于B）和 `sub(A, B)`（执行大数减法），其中 `R` 代表余数部分。

分步演示表：

步骤	落位	拼接前 R	拼接后 R	试商过程 (连减法寻找商 d)	记录商	更新后的 R
1	`A[0] = '8'`	`""`	`"8"`	`while("8" >= "21")` 假，`d=0`	`"0"`	`"8"`
2	`A[1] = '5'`	`"8"`	`"85"`	`while("85" >= "21")` 真，共减4次，`d=4`	`"04"`	`"1"`
3	`A[2] = '9'`	`"1"`	`"19"`	`while("19" >= "21")` 假，`d=0`	`"040"`	`"19"`

最终结果：

- 得到原始商 `"040"`，我们需要去除前导零，最终商为 `"40"`。
- 最终 `R` 的值为 `"19"`，这就是余数。

四、 代码实现

高精度除法它实现严重大数比较和减法。

```
// 判断大数 a 是否大于或等于 b (a >= b 返回 true)
bool isGreaterOrEqual(string a, string b) {
    if (a.length() != b.length()) {
        return a.length() > b.length();
    }
    return a >= b; // 长度相同时，按字典序比较即可等效于数值比较
}

// 高精度减法 (前提是 a >= b，且均为正数)
string sub(string a, string b) {
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());
    while (b.size() < a.size()) b += '0';

    string ans = "";
```

```
    int borrow = 0;

    for (int i = 0; i < a.size(); i++) {
        int diff = (a[i] - '0') - (b[i] - '0') - borrow;
        if (diff < 0) {
            diff += 10;
            borrow = 1;
        } else {
            borrow = 0;
        }
        ans += to_string(diff);
    }

    // 去除减法结果末尾多余的 '0'（反转后就是前导零）
    while (ans.size() > 1 && ans.back() == '0') {
        ans.pop_back();
    }
    reverse(ans.begin(), ans.end());
    return ans;
}

// 高精度除法
// 返回 pair<string, string>，分别代表 {商, 余数}
pair<string, string> divide(string a, string b) {
    // 特判处理
    if (b == "0") return {"除0错误", ""};
    if (a == b) return {"1", "0"};
    if (!isGreaterOrEqual(a, b)) return {"0", a}; // 被除数小于除数

    string quotient = "";  // 用于累加商
    string remainder = ""; // 用于维护当前片段的余数

    // 模拟竖式除法：从最高位向低位遍历被除数 a
    for (int i = 0; i < a.length(); i++) {
        // 落位：将当前位拼接到余数末尾
        remainder += a[i];

        // 清理余数可能产生的前导零（例如 05 -> 5）
        if (remainder.size() > 1 && remainder[0] == '0') {
            remainder.erase(remainder.begin());
        }

        // 试商（连减法）
        int count = 0;
        // 只要当前余数大于等于除数，就继续减
        while (isGreaterOrEqual(remainder, b)) {
            remainder = sub(remainder, b); // 执行大数减法
            count++;
        }

        // 记录本位计算出的商
        quotient += to_string(count);
    }
```

```
    // 清理商的前导零（例如 040 -> 40）
    while (quotient.size() > 1 && quotient[0] == '0') {
        quotient.erase(quotient.begin());
    }

    // 处理恰好整除的情况
    if (remainder.empty()) remainder = "0";

    return {quotient, remainder};
}
```

五、 算法瓶颈与优化方向

思考：连减法试商的效率问题

上述代码中，我们采用了“连减法”进行试商：`while (isGreaterOrEqual(remainder, b))`。
这是一种直观但效率不高的策略。最坏的情况下（例如某一位的商是 `9`），内部的 `while` 循环需要执行 `9` 次高精度减法。如果被除数极长，这将极大地拖慢程序运行速度。

二分法试商

为了提高试商效率，我们可以将其转化为一个**查找问题**。每一位的商必然是一个在 `0` 到 `9` 之间的整数。我们可以利用**二分查找法**在这个小区间内快速定位商的值。

对于一个猜测的商 `mid`，我们只需计算 `B × mid`（大数乘以单精度数），然后将其与 `remainder` 比较：

- 如果 `B × mid > remainder`：说明猜测的商太大了，正确答案在左半区间 `[0, mid - 1]`。
- 如果 `B × mid <= remainder`：说明猜测的商可能是对的，但也可能还能更大，我们记录下当前的 `mid`，并继续在右半区间 `[mid, 9]` 尝试。

通过二分查找，每次试商的次数从最多 9 次降低到大约 3-4 次。这在处理超大规模数字时，可以带来显著的性能提升。

六. 总结

1. **核心思想**：高精度除法是对手动**竖式除法**的模拟。
2. **关键步骤**：“落位” -> “试商” -> “相减” -> “记录商” 的循环。
3. **实现难点**：“试商”过程。简单实现是**循环减法**，高效实现是**二分查找+乘法**。
4. **基础依赖**：高精度除法建立在**比较和高精度**减法之上。