C++算法讲义：高精度乘法

第一部分：大数 × 小数

在学习复杂的“大数 × 大数”之前，我们先解决一个简化版的问题：一个很大的数乘以一个小数（`int` 范围内）。这是我们后续大数 × 大数方法一的基础。

核心思想 (大数 × 小数)

这个过程和高精度加法极其相似，都是逐位处理。不同之处在于，我们不再是 `A[i] + B[i]`，而是用大数的每一位 `A[i]` 去乘以那个小数 `b`，并且进位不再是 `0` 或 `1`，可以是任意大小的数字。

算法步骤：

1. 将大数 `A`（字符串）倒序，方便从个位开始处理。
2. 设置一个进位 `carry = 0`。
3. 遍历倒序后的大数 `A` 的每一位 `A[i]`。
4. 计算 `(A[i] - '0') * b + carry`，得到当前位的乘积结果 `res`。
5. 新结果的当前位是 `res % 10`。
6. 新的进位是 `res / 10`。
7. 循环结束后，如果 `carry` 还有剩余，需要继续处理完。
8. 最后将得到的结果字符串再次反转，即为最终答案。

代码实现 (大数 × 小数)

```
string mul(string a, int b) {
    // 特殊情况：如果乘数是 0 或大数是 "0"，结果直接是 "0"
    if (b == 0 || a == "0") return "0";

    // 反转大数 a，方便从个位开始处理
    reverse(a.begin(), a.end());

    string ans = "";
    int carry = 0; // 进位

    // 逐位与 b 相乘
    for (int i = 0; i < a.size(); i++) {
        // 1. 计算当前位的乘积，并加上上一位的进位
        int res = (a[i] - '0') * b + carry;

        // 2. 将乘积的个位数拼接到 ans 末尾
        ans += to_string(res % 10);

        // 3. 更新进位（进位是乘积整除10）
        carry = res / 10;
    }

    // 循环结束后，如果还有进位，需要全部追加到 ans 末尾
    while (carry) {
        ans += to_string(carry % 10);
        carry /= 10;
```

```
    }

    // 反转得到最终结果
    reverse(ans.begin(), ans.end());
    return ans;
}
```

第二部分：大数 × 大数

有了大数 × 小数的解决办法，我们现在可以开始着手去完成大数 × 大数的问题了。

方法一：分解相加法

核心思想

还记得乘法分配律吗？`a × (b + c) = a × b + a × c`。我们可以把一个大数拆开来看！

`321 × 45`
`= 321 × (5 + 40)`
`= 321 × 5 + 321 × 40`

我们把一个“大数 × 大数”问题，巧妙地转化成了：

1. 一个“**大数 × 小数**”（`321 × 5`）
2. 另一个“**大数 × 小数**”（`321 × 4`），再在其结果末尾**补0**
3. 最后把这两个结果用“**大数 + 大数**”加起来！

这个思路的精髓在于**复用**。我们只需要实现 `mul(string, int)` 和 `add(string, string)` 这两个函数，就可以组合出 `mul(string, string)` 的功能。

代码实现 (分解相加法)

```
// 注意：此函数依赖我们之前已经实现好的高精度加法函数，以及我们上面刚实现的 mul(string, int) 函数
string mul_v1(string a, string b) {
    if (a == "0" || b == "0") return "0";
    reverse(b.begin(), b.end()); // 反转乘数b，从个位开始处理
    string total_ans = "0";

    // 遍历 b 的每一位
    for (int i = 0; i < b.size(); i++) {
        int digit_b = b[i] - '0'; // 1. 取出 b 的当前位数字
        string part_ans = mul(a, digit_b); // 2. 调用大数x小数函数，计算部分积
        // 3. 补0（移位）。个位(i=0)补0个0，十位(i=1)补1个0...
        for (int j = 0; j < i; j++) {
            part_ans += '0';
        }
        total_ans = add(total_ans, part_ans); // 4. 调用“大数+大数”函数，累加到总结果上
    }

    return total_ans;
}
```

方法二：竖式模拟法

这种方法更高效，避免了方法一中频繁的字符串操作和多次相加。

8. 核心思想

我们回顾一下手算 `123 × 45` 的过程。本质上，我们是在计算所有数位的两两乘积，然后把它们放到正确的位置上。

- `3 × 5` 的结果在**个位**
- `2 × 5` 和 `3 × 4` 的结果都在**十位**
- `1 × 5` 和 `2 × 4` 的结果都在**百位**
- ...

规律发现：设被乘数 `A` 的第 `i` 位和乘数 `B` 的第 `j` 位相乘（`i`, `j` 从0开始，代表个位、十位...），其结果 `A[i] × B[j]` 应该累加到最终结果 `C` 的第 `i+j` 位上！

优化策略：为了避免复杂的实时进位，我们采用**先累加，后统一进位**的策略。

1. 创建一个足够长的结果数组 `res` (长度为 `n+m`)。
2. 用两层循环，计算所有 `A[i] × B[j]`，并直接累加到 `res[i+j]` 上，此时 `res` 里可能存着大于9的数。
3. 最后，从头到尾遍历 `res` 数组，一次性处理完所有进位。

9. 实例 (123 × 45)

存储与初始化

- `A = {3, 2, 1}` (123倒序)
- `B = {5, 4}` (45倒序)
- `res = {0, 0, 0, 0, 0}` (长度为 3+2=5)

逐位乘积累加 (不进位)

操作	累加位置	计算	res 数组状态
A[0]×B[0]	`res[0]`	`+= 3×5`	`{15, 0, 0, 0, 0}`
A[1]×B[0]	`res[1]`	`+= 2×5`	`{15, 10, 0, 0, 0}`
A[2]×B[0]	`res[2]`	`+= 1×5`	`{15, 10, 5, 0, 0}`
A[0]×B[1]	`res[1]`	`+= 3×4`	`{15, 22, 5, 0, 0}`
A[1]×B[1]	`res[2]`	`+= 2×4`	`{15, 22, 13, 0, 0}`
A[2]×B[1]	`res[3]`	`+= 1×4`	`{15, 22, 13, 4, 0}`

统一处理进位

位置 `i`	`res[i]` (处理前)	`res[i] + carry`	`ans += ...`	新 `carry`
0	15	`15 + 0 = 15`	`ans += "5"`	`1`
1	22	`22 + 1 = 23`	`ans += "3"`	`2`
2	13	`13 + 2 = 15`	`ans += "5"`	`1`
3	4	`4 + 1 = 5`	`ans += "5"`	`0`
4	0	`0 + 0 = 0`	`ans += "0"`	`0`

清理与输出

- 处理后得到的 `ans` 是 `"53550"`。
- 去除末尾多余的 '0' -> `"5355"`。
- 反转 -> `"5535"`。正确!

10. 代码实现 (竖式模拟法)

```
string mul_v2(string a, string b) {
    // 特殊情况：如果乘数是 0，结果直接是 0
    if (b == "0" || a == "0" ) return "0";
    // 反转，让下标 0 对应个位，1 对应十位...
    reverse(b.begin(), b.end());
    reverse(a.begin(), a.end());

    // 创建结果数组(不处理进位)
    // n位乘以m位，积最多是 n+m 位
    // 初始全部填充为 0
    int n = a.size(), m = b.size();
    vector res(n + m, 0);
    // 逐位相乘并累加
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            int digitA = a[i] - '0'; // 将字符转换为数字进行乘法
            int digitB = b[j] - '0';
            // a的第i位与b的第j位相乘，放在结果数组的第 i+j 位
            res[i + j] += digitA * digitB;
        }
    }

    string ans = ""; // 最终的结果
    int carry = 0;   // 储存进位，初始化为 0
    for (int i = 0; i < res.size(); i++) {
        // 计算当前位的数值，开始处理进位
        int num = res[i] + carry;
        // 计算下一位的进位
        carry = num / 10;
        // 当前位只保留个位数添加到结果
        ans += to_string(num % 10);
    }
    while(carry){  // 处理最后的进位
        ans += to_string(carry % 10);
        carry /= 10;
```

```
    }
    // 去除末尾多余的 '0'（即原数的前导零）
    while (ans.size() > 1 && ans.back() == '0') {
        ans.pop_back();
    }

    // 翻转回来
    reverse(ans.begin(), ans.end());
    return ans;
}
```

方法对比

结论：在算法竞赛中，**方法二（竖式模拟法）是主要的选择。**方法一则胜在思路巧妙，体现了优秀的模块化和问题分解思想。

总结

1. 高精度乘法有两种主流思路：**分解相加**和**竖式模拟。**
2. **分解相加**：将 `大数×大数` 转化为多次 `大数×小数` 和 `大数+大数`，核心是**复用。**
3. **竖式模拟**：用 `vector` 存储中间结果，核心是 `C[i+j] += A[i] × B[j]` 公式和**先累加后进位**的思想。
4. 在性能上，**竖式模拟法**优于分解相加法。