深 圳 大 学 实 验 报 告

课程名称 高级算法设计与分析

项目名称 分治法求解大整数乘法

学　　院 计算机与软件学院

专　　业 计算机科学与技术

指导教师 卢亚辉

报 告 人 付志远 学号 2100271069

实验时间 2022 年 6 月

提交时间 2022 年 6 月

教务处制

实验目的

(1) 掌握分治法；
(2) 学会测试和分析算法的时间性能。

大整数乘法

计算机整型变量无法直接表示过大的整数，进行乘法运算会发生溢出错误；而采用浮点数来保存大整数时，则会丢失低位的数据，因此在计算大整数的乘法运算时，需要定义新的存储结构来保存数据[1]。

数据结构设计

因为大整数的长度不确定，所以我们采用向量（动态数组）来存储一个大整数。其中，大整数按位位从高到低分别放在数组的每一项中[2][3]。

例如，大整数 12345，所对应的程序代码表示是[5, 4, 3, 2, 1]。

Data Structure: big integer

```
1.  struct BigInteger {
2.      big_integer: Vec<i32>
3.  }
```

我们对向量进行封装，一个向量就可以表示一个大整数。通过调用向量的 **len()**方法，来计算并分配大整数需要的内存空间，向量的长度即为大整数的长度。

普通方法

以大整数 123 和 789 的乘法计算过程为例，归纳求出大整数乘法的一般流程。在进行竖式运算时，大整数 123 分别与 789 的每一位相乘，并且进行移位，最后将几次相乘结果求和。

```
    1 2 3
  × 7 8 9
-----------
    1 1 0 7
    9 8 4
  8 6 1
-----------
  9 7 0 4 7
```

观察竖式乘法的计算过程，我们可以发现，最终结果的每一位之间相互独立，并不受中

间大整数相加顺序的影响。因此，中间计算过程的大整数并不需要额外开辟空间进行存储，而是把结果直接加入最后的大整数向量中[4]。

算法的伪代码如下，

Algorithm 1: big_integer_multiply

Input: *bi1*, a big integer;
bi2, another big integer;
Output: *bi*, the result of *bi1* times *bi2*

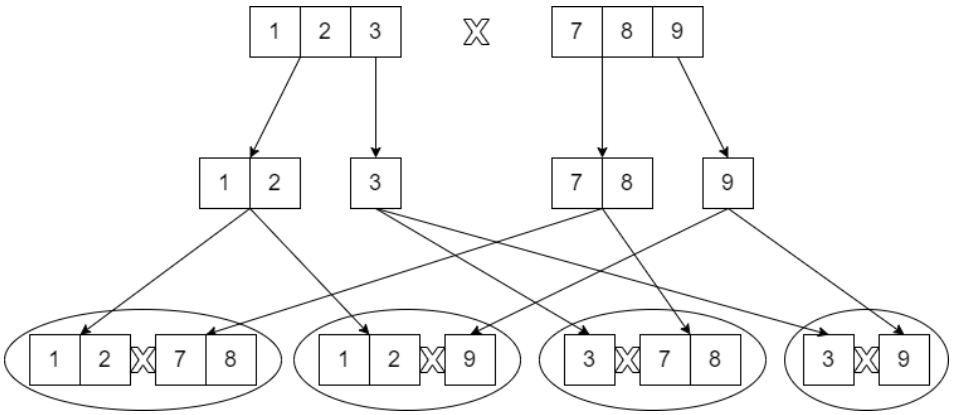
```
1.  let len = bi1.len();
2.  let bi = BigInteger(len*2+1);
3.  for i in 0..len:
4.      for j in 0..len:
5.          let mul = bi1[i] * bi2[j];
6.          let total = mul + bi[i+j];
7.          bi[i+j+1] += total / 10;
8.          bi[i+j]    = total % 10;
9.  remove leading 0's of bi.
10. return bi;
```

根据伪代码我们可以发现，该算法的主要是两层循环，如果假设输入的大整数长度为n，那么该算法的时间复杂度为$O(n^2)$, 空间消耗主要是需要为乘积结果分配新的动态数组[5]，空间复杂度为$O(n)$。

分治法

我们还是以大整数 123 和 789 的乘法为例，不过这次采用分治法进行求解。首先，如果两个大整数中有一个大整数长度为 1，那么我们直接使用普通的大整数乘法，完成一次平凡的乘法计算。否则，进行分治求解[6]。我们将大整数平均分为两部分，高位部分（对应的向量下标范围是$[\lfloor len/2 \rfloor, len)$）和低位部分（对应的向量下标范围是$[0, \lfloor len/2 \rfloor)$），两部分的区间都是左闭右开。

以大整数 123 为例，它的向量表示方法为[3, 2, 1]，高位部分为[2, 1]，低位部分为[3]。



如图所示，123 和 789 在进行第一次递归时，有三个部分已经到达递归边界，而 12 和

78 的乘积计算则需要进一步的递归。

在完成四个子项求解后，最后需要把结果进行合并，两个高位部分的乘积需要补充两个 0，一个高位部分和一个低位部分的乘积则需要补充一个 0，两个低位部分的乘积无需补充 0。

观察结果我们归纳出补充 0 的情况，如下表，

乘法项	补充 0 的数量
高位部分×高位部分	$\lfloor len/2 \rfloor + \lfloor len/2 \rfloor$
高位部分×低位部分	$\lfloor len/2 \rfloor$
低位部分×高位部分	$\lfloor len/2 \rfloor$
低位部分×低位部分	0

其中 len 表示大整数的长度。

递归算法伪代码如下，

Algorithm 2: big_integer_multiply_recursion

Input: *bi1*, a big integer;
bi2, another big integer;
Output: *bi*, the result of *bi1* times *bi2*

```
1.  let len1 = bi1.len();
2.  let len2 = bi2.len();
3.  if len1 or len2 is 1:
4.      return big_integer_multiply(bi1, bi2);
5.
6.  let bi1_low = bi1[0..len1/2];
7.  let bi1_high = bi1[len1/2, len1];
8.  let bi2_low = bi2[0..len2/2];
9.  let bi2_high = bi2[len2/2, len2];
10.
11. let high = big_integer_multiply_recursion(bi1_high, bi2_high);
12. let mi1 = big_integer_multiply_recursion(bi1_high, bi2_low);
13. let mi2 = big_integer_multiply_recursion(bi1_low, bi2_high);
14. let low = big_integer_multiply_recursion(bi1_low, bi2_low);
15.
16. let bi = merge high, mi1, mi2 and low;
17. remove leading 0's of bi.
18. return bi;
```

下面我们分析该递归过程的时间复杂度，显然对于$n = 1$，有$T(n) = O(1)$；对于$n > 1$的情况，我们注意到，我们把$n \times n$的问题，划分为 4 个$n/2 \times n/2$的子问题，并且完成子问题的求解后，还需要进行 3 次的加法将结果合并，因此可以列出递归公式，

$$T(n) = 4T(n/2) + O(n)$$

可以根据主定理带入求解时间复杂度，求出$T(n) = O(n^2)$，具体求解过程列于附录。

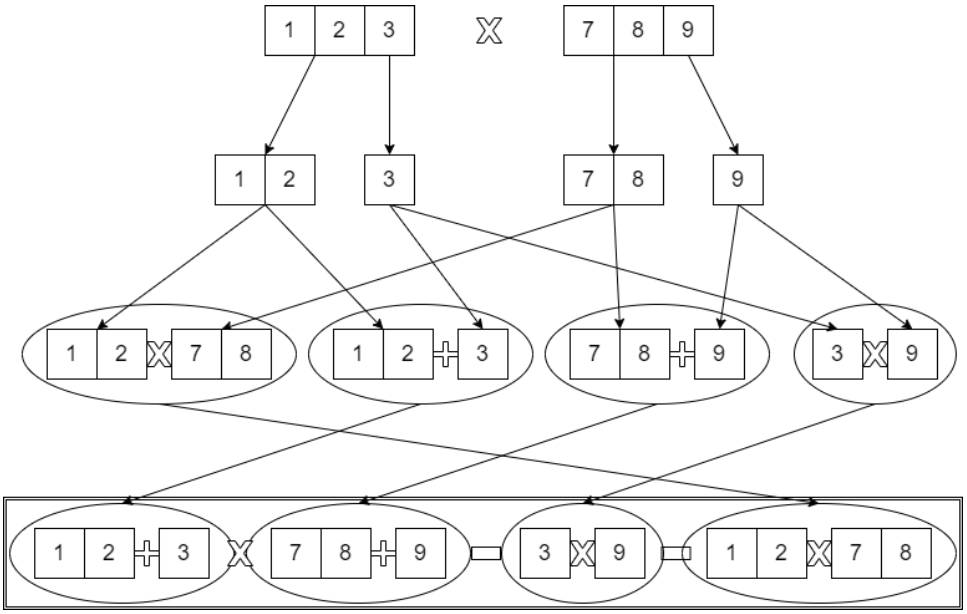
虽然我们在递归过程中传入的都是大整数的引用，但结尾的合并过程，需要用到加法和移位操作，需要额外的空间。递归深度是$O(\log n)$，每次递归的开销是$O(n)$，因此空间复杂度为$O(n\ \log n)$。

分治法改进

改进的分治法和普通的分治法相比，主要是将 mid 中 2 次乘法计算压缩为一次乘法，进而加快了算法速度。

下面我们从数学的角度进行分析，假设大整数a和b表示为$a_1 \times 10^{n/2} + a_0$ 和$b_1 \times 10^{n/2} + b_0$，显然高位部分和低位部分的值分别为$a_1 b_1$和$a_0 b_0$，中间部分$a_0 b_1 + a_1 b_0$等价于$(a_0 + a_1)(b_0 + b_1) - (a_0 b_0 + a_1 b_1)$，而后半部分的结果正是高位部分和低位部分。也就是说，我们在计算中间部分的 2 次乘法和 1 次加法，现在变为计算 3 次加法、1 次减法（或者 2 次加法、2 次减法）和 1 次乘法。显然，当 n 较大时，1 次乘法的开销要大于 3 次加法和 1 次加法的总开销[7]。

还是以大整数 123 与 789 的乘积为例子，如下图所示，



过程与递归法类似，只是不同之处在于，在拆分两个大整数后，不是进行交叉相乘，而是分别计算高位与低位部分的和（$12 + 3$、$78 + 9$），再用两个和相乘的积（$(12 + 3) \times (78 + 9)$）减去高位之积和低位之积（$(3 \times 9 + 12 \times 78)$）。

改进的递归算法伪代码如下，

Algorithm 3: big_integer_multiply_recursion_plus

Input: *bi1*, a big integer;
bi2, another big integer;

Output: *bi*, the result of *bi1* times *bi2*

```
19. let len1 = bi1.len();
20. let len2 = bi2.len();
21. if len1 or len2 is 1:
22.         return big_integer_multiply(bi1, bi2);
23.
24. let bi1_low = bi1[0..len1/2];
25. let bi1_high = bi1[len1/2, len1];
26. let bi2_low = bi2[0..len2/2];
27. let bi2_high = bi2[len2/2, len2];
28.
29. let high = big_integer_multiply_recursion_plus(bi1_high,
    bi2_high);
30. let low = big_integer_multiply_recursion_plus(bi1_low,
    bi2_low);
31.
32. let bi1_hl = big_integer_add(bi1_low, bi1_high);
33. let bi2_hl = big_integer_add(bi2_low, hi2_high);
34. let prod = big_integer_multiply_recursion_plus(bi1_hl, bi2_hl);
35. let h_add_l = big_integer_add(high, low);
36. let mid = big_integer_sub(prod, h_add_l);
37.
38. let bi = merge high, mid and low;
39. remove leading 0's of bi.
40. return bi;
```

时间复杂度分析同上，只是递推公式稍有不同，

$$T(n) = 3T(n/2) + O(n)$$

它的时间复杂度则从$O(n^2)$降低到了$O(n^{1.59})$。空间复杂度和递归法相同，也是$O(n\log n)$，虽然比递归法使用了更多空间来进行加减运算，但数量级并没有产生较大影响。

实验环境配置

软件/硬件	版本信息
OS	Linux/Ubuntu20.02
Vim	8.1.2269
Rust(rustc)	1.61.0
Cargo	1.61.0

实验结果

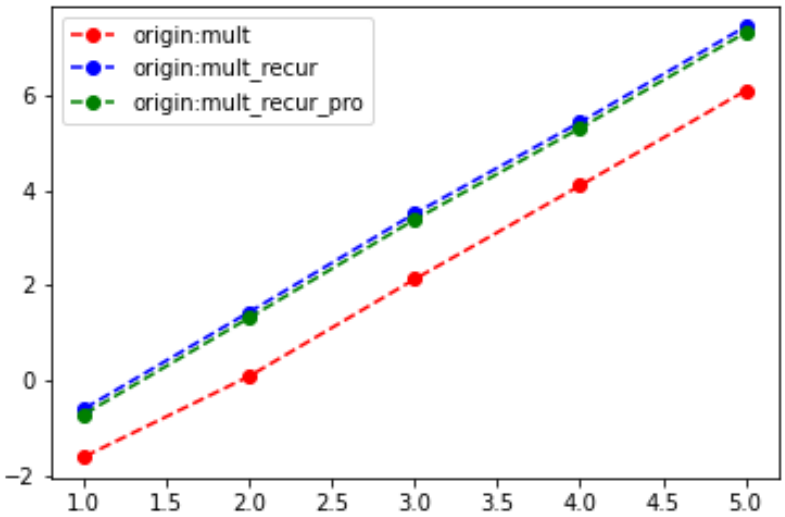
对于每次实验，我们随机生成两个长度位 len 的大整数，分别送入到三种算法中，测量程序运行时间。

针对长度低于 1000 位的大整数，我们运行 1000 次实验，计算 1000 次结果的平均值作为运算结果；不低于 1000 位的大整数我们只进行一次运算，实验结果的单位是毫秒（ms）。实验结果见于下表，

	普通大整数乘法的时间/ms	分治大整数乘法的时间/ms	改进的分治大整数乘法的时间/ms
10 位	0.025	0.255	0.196
100 位	1.236	27.514	20.973
1000 位	135	3234	2415
10000 位	12492	267729	201903
10 万位	1229886	27203927	20365492

```
fuzhiyuan@hp-HP-Z8-G4-Workstation ~/big-integer-multiplication main* ↓
❯ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
fuzhiyu+ 2480674  0.0  0.0  21964  7796 pts/4    Ss   5月23   0:00 zsh
fuzhiyu+ 2480686  0.0  0.0  18692  4288 pts/5    SNs+ 5月23   0:00 zsh
fuzhiyu+ 2481248 99.9  0.0  27016 17380 pts/4    R+   5月23 18941:16 target/debug/big-integer-multiplication
fuzhiyu+ 3988463  0.1  0.0  21996  7964 pts/2    Ss   6月05   0:00 -zsh
fuzhiyu+ 3988474  0.0  0.0  18636  4292 pts/6    SNs+ 6月05   0:00 -zsh
fuzhiyu+ 3989514  0.0  0.0  18908  3544 pts/2    R+   00:02   0:00 ps -u
```

观察结果我们可以看出，10 万位的大整数乘法，在递归法求解时大约需要 8 小时，而在我们运行大整数乘法的 13 天后，我们仍旧没能得出 100 万位的大整数乘法结果。



此外，按照理论分析，三种方法的时间复杂度分别为$O(n^2)$、$O(n^2)$和$O(n^{1.59})$，虽然改进递归法的大整数乘法确实比递归法所花费的时间更少，但后两种算法的时间开销是前者的 10 倍，已经超出了一个数量级，这与理论计算的时间复杂度并不相符。

在检查代码，发现移位操作（处理递归后合并高位和中位部分对齐）可能是产生耗时的一方面原因，因为$\times 10^n$和$\times 10^{n/2}$也近似地被当作进行乘法处理。在优化移位操作后，运行

时间减少了大约 14%，但并没有改变数量级的差异。

改进数据结构

经过观察和测试，可以发现递归法及其改进法中加减运算等需要重新分配数组，有较大的时间开销；此外，在每次向量空间不够存储新元素时，向量需要进行一个拷贝操作，从而到新空间存储更大的向量，这种再分配操作在向量较小时很频繁，拷贝操作则在向量较大时很费时。

因此，我们必须减少向量的扩展（resize）和拷贝（copy）。总体而言，有两个思路，一是结合编程语言的特型，二是使用空间换时间。前者可以结合 Rust 所有权转移的特点，在递归过程中减少新向量分配，直接传入向量而非切片，但有些向量部分需要重复使用，整体处理起来较为复杂；后者在为大整数分配空间时，直接开辟一个较大的内存空间，并且设定两个“指针”，start 和 len 来标注我们关注的大整数范围。

因此，本次实验，拟采用后一种设计思路。

Data Structure: big integer

```
1.  struct BigInteger {
2.      start: usize,
3.      len: usize,
4.      nums: Vec<i32>
5.  }
6.  SIZE = 100, 000, 000 //A big number
7.  new() -> BigIntger {
8.      BigInteger {
9.          Start:0, len:1, nums: [0; SIZE]
10.     }
11. }
12. …
```

我们以分治法为例，每次计算递归乘法前，我们修改大整数的 start 和 len 来选取我们关注的部分，比如选取高位部分只需要令$start = \lfloor len/2 \rfloor$, $len = \lceil len/2 \rceil$，这样我们计算高位部分与高位部分的乘积则会直接从$start = start1 + start2$开始，最终在合并 high、mid 和 low 三部分时，无需移位操作。而且全部运算过程并没有超出 SIZE 范围，无需重新分配向量，只是移动“指针”，要比之前开销少很多。

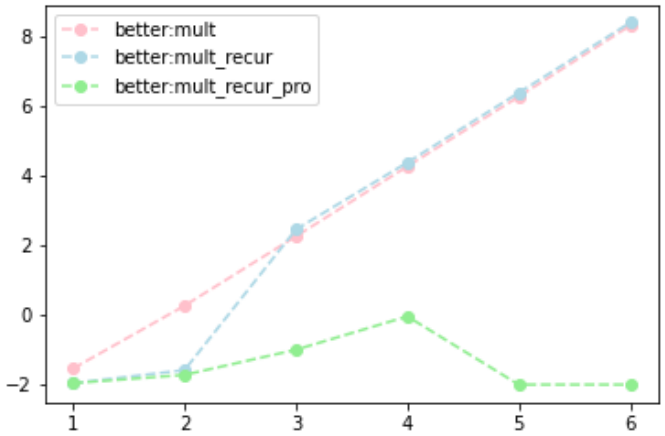
改进后的结果

实验运行配置同上，结果列于下表，

	普通大整数乘法的时间/ms	分治大整数乘法的时间/ms	改进的分治大整数乘法的时间/ms

10 位	0.029	0.011	0.011
100 位	1.827	0.026	0.019
1000 位	178	280	0.1
10000 位	18009	23286	0.903
10 万位	1798834	2323249	**0.01**
100 万位	196749851	238737294	**0.01**

观察实验结果，改进后的数据结构算法运行时间在 1 小时内完成了前一种设计 16 个小时的运行结果。特别是分治法及其改进法，效率上都有明显提升。

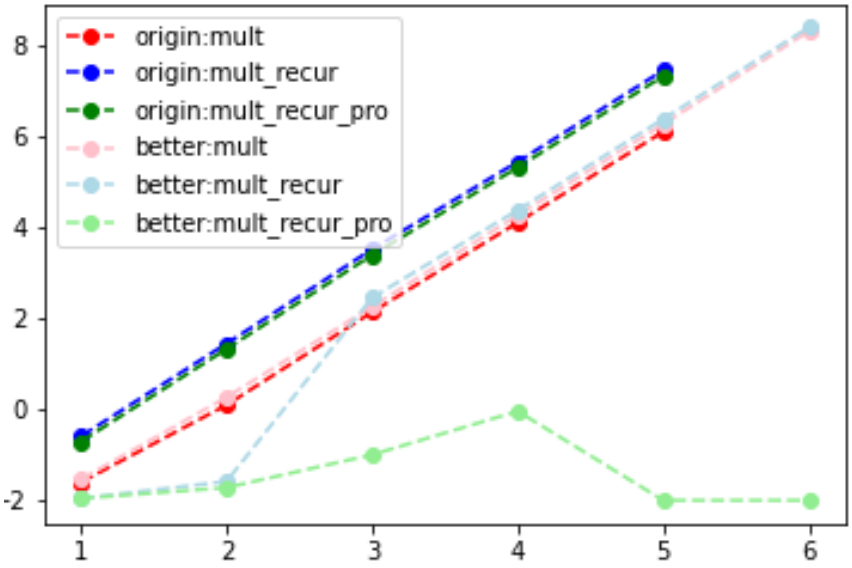


普通大整数乘法在时间消耗上略有提升，但没有明显的副作用。开销变大的原因可能在于原本只需要分配$len1 + len2$的新向量，现在则需要分配远大于该值的$SIZE$大小的向量。

分治法的时间开销在整数长度较小时比普通方法要少，但大整数长度达到 1000 位后比普通方法高，但并没有超过普通法的两倍，也在同一个数量级上，是符合理论上的时间复杂度分析的。

改进的分治法结果似乎出人意料地优秀，在 10 万位大整数乘法运算上也仍旧不超过 1 ms。改进法在 10 万位及其这个数量级的计时上，产生了 0.01ms 的骤降（相比于 10000 位的 0.903 ms）。推测可能的原因是随机产生的数中间有较多的 0，被改进的分治法充分利用，或者算法在实现过程中有疏漏。

改进数据结构的比较



上图中，浅色系是改进后的时间开销，深色系是改进前的时间开销。对比可以看出，改进后的普通大整数乘法和递归法都与未改进前的普通大整数乘法开销在一个数量级上，这与理论推导的时间复杂度一致。浅色系总体要低于深色系的时间开销，也证明了改进数据结构的结果是有效的。

总结

实验仍有不少能够再完善的地方，一下三点是我认为还能够继续完善的地方。

(1) 递归法可以设置更优的最小计算单元，长度较小的部分或许使用普通大整数方法更快；此外较小的部分可以考虑使用整形数据进行计算；

(2) 分配大整数时，优化空间分配，不是一次性分配完，而是预估可能需要的最大空间进行分配，进而减少分配空间时的开销；

(3) 优化大整数访问过程，尽可能采用迭代器替代数组下标，减少地址边界检查开销。

实验心得

在写大整数递归法时，将大整数a和b表示为$a_1 \times 10^{n/2} + a_0$ 和$b_1 \times 10^{n/2} + b_0$，按照数学推理，相乘后的结果是$a_1b_1 \times 10^n + (a_0b_1 + a_1b_0) \times 10^{n/2} + a_0b_0$，高位进行移位进行补充零的数量应该是$n$个。但在写代码中，要考虑整除的因素，高位部分和低位部分的值为a_1b_1进行移位操作时需要补充的 0 的数量在代码中应当是$\lfloor n/2 \rfloor + \lfloor n/2 \rfloor$个。我一开始写代码时直接参考了课件中的结论，直接进行n个 0 补充，就会发现在奇数位大整数乘法的时候会出错，检查了好久才发现不是代码写错了，而是没有清楚理解数学表达式。

在写改进递归法的时候，一开始犹豫出现减法如何处理，害怕因为减法的出现，在运算的过程过出现了负数，导致情况更为复杂。这之后，通过观察到中间部分$a_0b_1 + a_1b_0$等价于$(a_0 + a_1)(b_0 + b_1) - (a_0b_0 + a_1b_1)$，这个减法的结果就是原来两部分之和，一定不会是负数，或者说被减数一定不小于减数，可以保证计算过程中不会有负数问题。

在改进数据结构时，发现递归法时间消耗仍然较高，之后通过调整四部分计算顺序（按照格雷编码的顺序），减少了 3 次指针调整，从而使递归法的消耗时间大大减少，提升了计算效率。

通过本次实验，我意识到了认真和细心的必要性，一开始认为实验流程比较简单，但开始做之后才发现自己许多地方被卡住，写出来的代码也还有许多小细节需要去完善。因此，我要在以后的学习工作中更加认真，不轻视每一个问题，细心处理每一点。

参考文献

[1] BRASSARD G, BRATLEY P, Algorithmics: theory and practie[M].[s.l.]: Prentice- Hall, 1988.

[2] 李文化.大整数精确运算系统研究与开发[D].重庆: 重庆大学, 2005- 12

[3] 李文化，董克家．大整数精确运算的数据结构与基选择[J]．计算机工程与应用，2006，42(32) : 24-26．

[4] 杜青.基于类的大整数乘法运算的实现[J].微型机与应用,2017,36(02):8-9+13.DOI:10.19358/j.issn.1674-7720.2017.02.003.

[5] 杨灿，桑波． 大整数乘法运算的实现及优化［J］． 计算机工程 与科学，2013，35(3) : 183-190．

[6] 王念平，金晨辉． 用分治算法求大整数相乘问题的进一步分 析[J]． 电子学报，2008，36(1) : 133-135．

[7] 周健，李顺东，薛丹． 改进的大整数相乘快速算法［J］． 计算 机工程，2012，38(16) : 121-123．

附录

当$n=1$时，显然有$T(n)=1$。
当$n>1$时，不妨假设$n=2^k$，更方便求解，此时$k=\log_2 n$。
不断迭代递推公式，有，

$$T(n)=4T\left(\frac{n}{2}\right)+O(n)$$
$$=4^2T\left(\frac{n}{2^2}\right)+O(n)$$
$$=\cdots$$
$$=4^kT\left(\frac{n}{2^k}\right)+O(n)$$
$$=4^{\log_2 n}T(1)+O(n)$$
$$=n^2+O(n)$$
$$=O(n^2)$$