The Reason Why Some Divide-and-Conquer Algorithms Cannot Be Efficiently Implemented

Zhengjun Cao[1,*], Lihua Liu[2]

Abstract. In the literature there are some divide-and-conquer algorithms, such as Karatsuba's algorithm and Strassen's algorithm, which play a key role in analyzing the performance of some cryptographic protocols and attacks. But we find these algorithms are rarely practically implemented although their theoretical complexities are attractive. In this paper, we point out that the reason of this phenomenon is that decomposing the original problem into subproblems does not take constant time. The type of problem decomposition results in *data expand exponentially.* In such case the time for organizing data (including assigning address, writing and reading) which is conventionally ignored, accumulates significantly.

Keywords. divide-and-conquer algorithm, data expansion, merge sort, Karatsuba's algorithm, Strassen's algorithm, Fast Fourier Transform.

1 Introduction

In computer science, a divide-and-conquer algorithm works by decomposing recursively a problem into subproblems of the same or related type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Its computational cost is often estimated by solving the corresponding recurrence relation.

There are some famous divide-and-conquer algorithms in the literature [5, 8, 9], such as Karatsuba's algorithm and Strassen's algorithm. These algorithms play a key role in analyzing the performance of some cryptographic protocols and attacks. But we find they are rarely practically implemented although their theoretical complexities are attractive. The existing explanations on the phenomenon mainly include [4]: (1) the constant factor hidden in the running time is larger than the constant factor in the naive method; (2) the subproblems consume space; (3) they are not quite as numerically stable as the naive method.

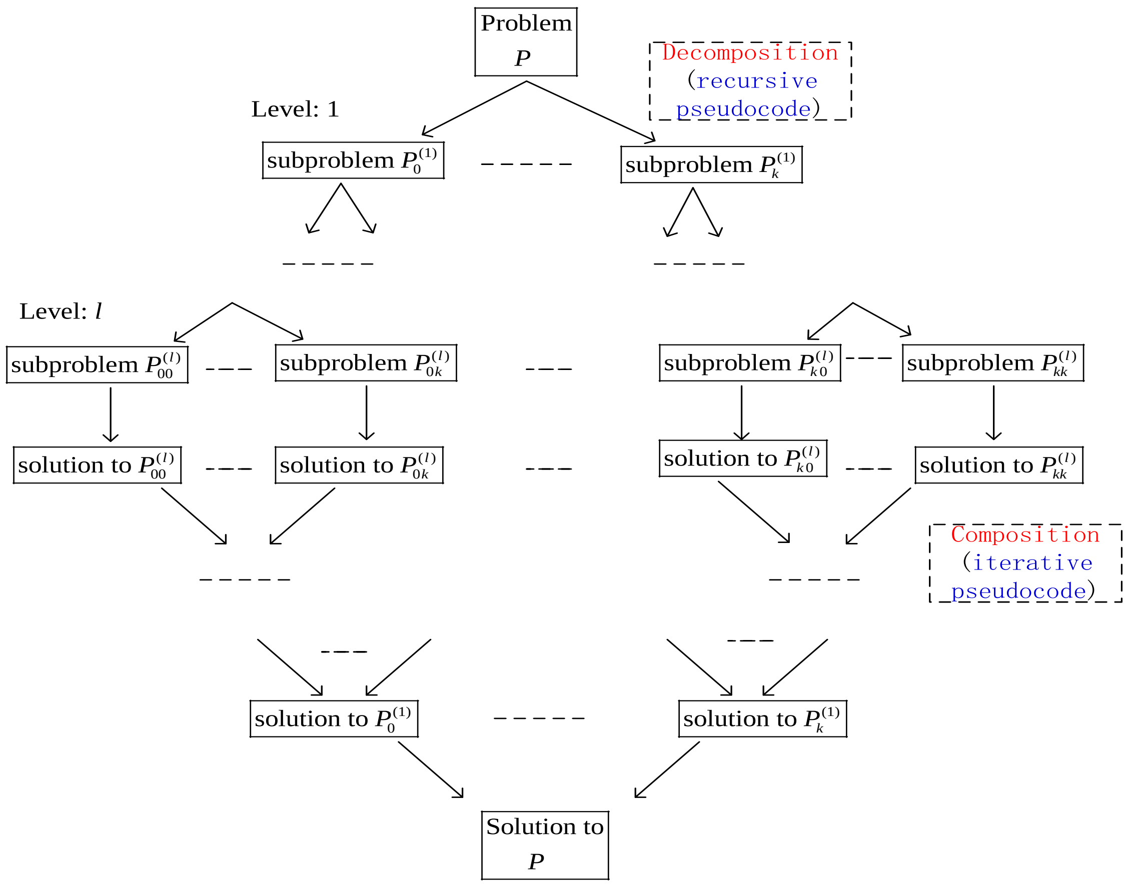
[1]Department of Mathematics, Shanghai University, Shanghai, China. * caozhj@shu.edu.cn
[2]Department of Mathematics, Shanghai Maritime University, Shanghai, China.

In this paper, we would like to stress that the ultimate reason is that decomposing the original problem into subproblems does not take constant time. The type of problem decomposition results in data expand exponentially. As a consequence, it seems impossible to convert such a recursive algorithm into its iterative version. The time for organizing data (including assigning address, writing and reading) which is conventionally neglected, accumulates significantly.

2 Analysis of divide-and-conquer algorithms

The overall running time of a divide-and-conquer algorithm is often be described by a recurrence equation. Let $T(n)$ be the running time on a problem of size n. Suppose that each division of the problem yields a subproblems, each of which is $1/b$ the size of the original. Suppose that the straightforward solution to the problem with small enough size, takes constant time $\Theta(1)$.

Figure 1: Working flow of a divide-and-conquer algorithm


Problem P
Decomposition (recursive pseudocode)
Level: 1
subproblem $P_0^{(1)}$
subproblem $P_k^{(1)}$
Level: l
subproblem $P_{00}^{(l)}$
subproblem $P_{0k}^{(l)}$
subproblem $P_{k0}^{(l)}$
subproblem $P_{kk}^{(l)}$
solution to $P_{00}^{(l)}$
solution to $P_{0k}^{(l)}$
solution to $P_{k0}^{(l)}$
solution to $P_{kk}^{(l)}$
Composition (iterative pseudocode)
solution to $P_0^{(1)}$
solution to $P_k^{(1)}$
Solution to P

Let $D(n)$ be the running time to divide the problem into subproblems and $C(n)$ be that to combine the solutions to the subproblems into the solution to the upper level problem. The recurrence equation for such case is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Aho, Hopcroft, and Ullman [1] popularized the use of recurrence relations to describe the running times of recursive algorithms. Seemingly, it is very convenience to describe and analyze a divide-and-conquer algorithm according to its recursive version.

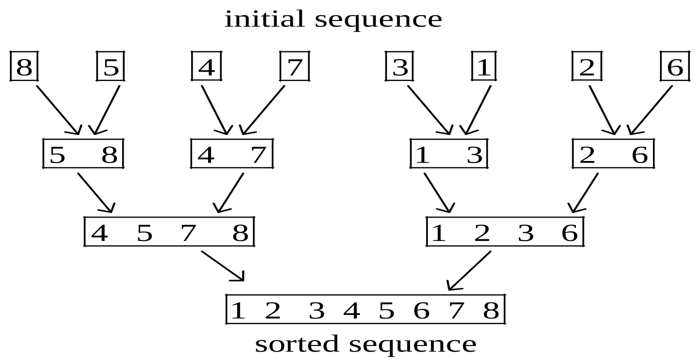
In order to implement practically a recursive algorithm, it is usual that one has to write down a programming pseudocode for problem decomposition and another programming pseudocode for problem composition. Note that the pseudocode for problem composition is iterative (see Fig. 1).

3 Iterativable divide-and-conquer algorithms

3.1 A trivial example: merge sort

The merge sort runs as follows. Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each, where $n = 2^l$ for some positive integer l. Sort the two subsequences recursively using merge sort. Merge the two sorted subsequences to produce the sorted answer. For example, the sequence $\{8, 5, 4, 7, 3, 1, 2, 6\}$ can be sorted by the following steps.

Figure 2: Working flow of merge sort



Since the problem decomposition just groups elements in sequence, it does take constant time, i.e., $D(n) = \Theta(1)$. Merging two n-element ordered arrays takes time $C(n) = \Theta(n)$. Thus, the recurrence equation for merge sort is

$$T(n) = 2T(n/2) + \Theta(n), \ n > 1.$$

By "master theorem" [4], we have $T(n) = \Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$.

Remark 1. *Notice that it is easy to convert the recursive algorithm into its iterative version.*

3.2 An explicit example: Fast Fourier Transform

The straightforward method of multiplying two polynomials of degree n takes $\Theta(n^2)$ time. The Fast Fourier Transform (FFT), popularized by a publication of Cooley and Tukey [3], can reduce the time to multiply polynomials to $\Theta(n \lg n)$.

3.2.1 Recursive FFT

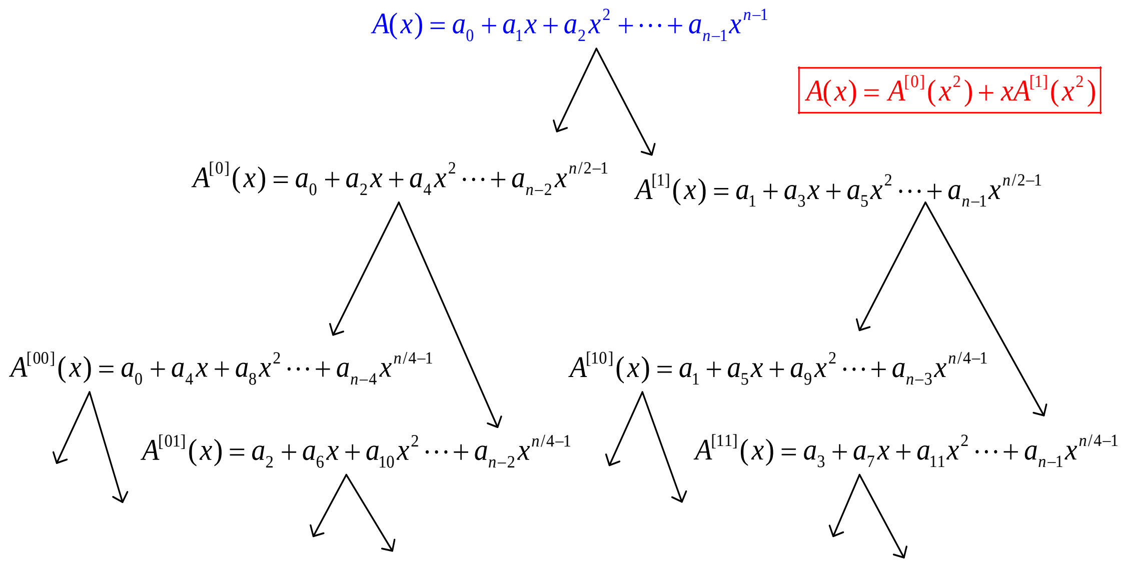
Let $\omega = e^{2\pi i/n}$ be a primitive nth root of unity. Let $f(x) = \sum_{0 \le i < n} f_i\, x^i \in \mathbb{C}[x]$ be a polynomial of degree less than n with its coefficient vector $(f_0, \cdots, f_{n-1}) \in \mathbb{C}^n$, where $n = 2^l$ for some positive integer l. The map

$$\mathbf{DFT}_\omega : \begin{cases} \mathbb{C}^n & \to \quad \mathbb{C}^n \\ (f_0, \cdots, f_{n-1}) & \mapsto \quad \big(f(1), f(\omega), f(\omega^2), \cdots, f(\omega^{n-1})\big) \end{cases}$$

which evaluates a polynomial at the powers of ω is called the Discrete Fourier Transform.

There are two common descriptions of FFT. One is recursive [4]. Let $A(x) = \sum_{0 \le i < n} a_i\, x^i \in \mathbb{C}[x]$ be a polynomial of degree less than n. Define $A^{[0]}(x) = \sum_{i=0}^{n/2-1} a_{2i} x^i$, $A^{[1]}(x) = \sum_{i=0}^{n/2-1} a_{2i+1} x^i$. We have $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$. The working flow of the recursive DFT algorithm can be depicted as follows (see Fig. 3).

Figure 3: Working flow of the recursive DFT algorithm



$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$
$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$
$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 \cdots + a_{n-2} x^{n/2-1}$
$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 \cdots + a_{n-1} x^{n/2-1}$
$A^{[00]}(x) = a_0 + a_4 x + a_8 x^2 \cdots + a_{n-4} x^{n/4-1}$
$A^{[01]}(x) = a_2 + a_6 x + a_{10} x^2 \cdots + a_{n-2} x^{n/4-1}$
$A^{[10]}(x) = a_1 + a_5 x + a_9 x^2 \cdots + a_{n-3} x^{n/4-1}$
$A^{[11]}(x) = a_3 + a_7 x + a_{11} x^2 \cdots + a_{n-1} x^{n/4-1}$

Clearly, the problem decomposition takes constant time, i.e., $D(n) = \Theta(1)$. Combining the solutions to the subproblems into the solution to the upper level problem takes time $C(n) = \Theta(n)$.

Thus, the recurrence equation for FFT is

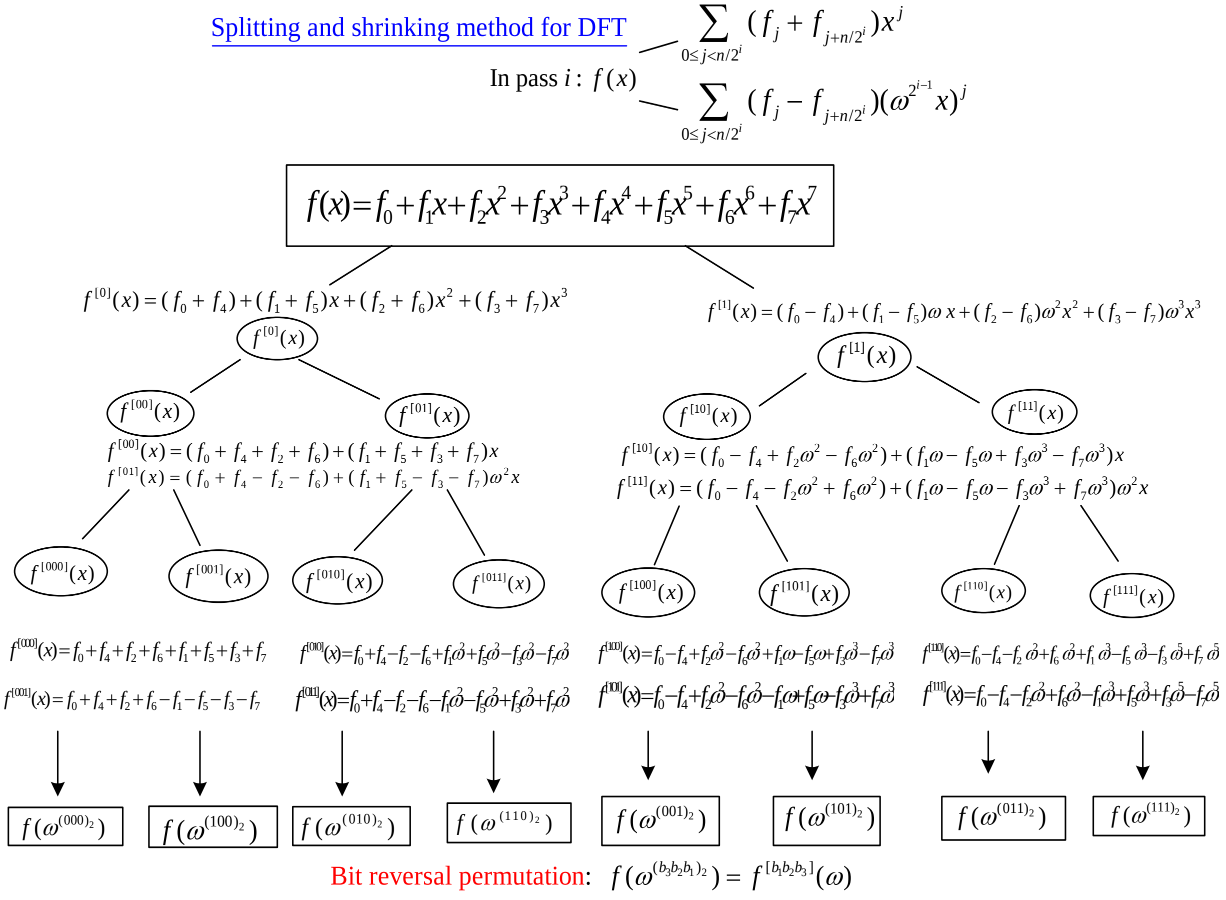
$$T(n) = 2T(n/2) + \Theta(n), \ n > 1.$$

By "master theorem", we have $T(n) = \Theta(n \lg n)$.

3.2.2 Iterative FFT

The other description of FFT is iterative [6]. We refer to the following working flow for the details of DFT (see Fig. 4).

Figure 4: Working flow of DFT algorithm for $n = 8$



Splitting and shrinking method for DFT
In pass i: $f(x)$
$\sum_{0\le j<n/2^i}(f_j + f_{j+n/2^i})x^j$
$\sum_{0\le j<n/2^i}(f_j - f_{j+n/2^i})(\omega^{2^{i-1}}x)^j$
$f(x)=f_0+f_1x+f_2x^2+f_3x^3+f_4x^4+f_5x^5+f_6x^6+f_7x^7$
$f^{[0]}(x)=(f_0+f_4)+(f_1+f_5)x+(f_2+f_6)x^2+(f_3+f_7)x^3$
$f^{[1]}(x)=(f_0-f_4)+(f_1-f_5)\omega x+(f_2-f_6)\omega^2x^2+(f_3-f_7)\omega^3x^3$
$f^{[0]}(x)$
$f^{[1]}(x)$
$f^{[00]}(x)$
$f^{[01]}(x)$
$f^{[10]}(x)$
$f^{[11]}(x)$
$f^{[00]}(x)=(f_0+f_4+f_2+f_6)+(f_1+f_5+f_3+f_7)x$
$f^{[01]}(x)=(f_0+f_4-f_2-f_6)+(f_1+f_5-f_3-f_7)\omega^2x$
$f^{[10]}(x)=(f_0-f_4+f_2\omega^2-f_6\omega^2)+(f_1\omega-f_5\omega+f_3\omega^3-f_7\omega^3)x$
$f^{[11]}(x)=(f_0-f_4-f_2\omega^2+f_6\omega^2)+(f_1\omega-f_5\omega-f_3\omega^3+f_7\omega^3)\omega^2x$
$f^{[000]}(x)$
$f^{[001]}(x)$
$f^{[010]}(x)$
$f^{[011]}(x)$
$f^{[100]}(x)$
$f^{[101]}(x)$
$f^{[110]}(x)$
$f^{[111]}(x)$
$f^{[000]}(x)=f_0+f_4+f_2+f_6+f_1+f_5+f_3+f_7$
$f^{[001]}(x)=f_0+f_4+f_2+f_6-f_1-f_5-f_3-f_7$
$f^{[010]}(x)=f_0+f_4-f_2-f_6+f_1\omega^2+f_5\omega^2-f_3\omega^2-f_7\omega^2$
$f^{[011]}(x)=f_0+f_4-f_2-f_6-f_1\omega^2-f_5\omega^2+f_3\omega^2+f_7\omega^2$
$f^{[100]}(x)=f_0-f_4+f_2\omega^2-f_6\omega^2+f_1\omega-f_5\omega+f_3\omega^3-f_7\omega^3$
$f^{[101]}(x)=f_0-f_4+f_2\omega^2-f_6\omega^2-f_1\omega+f_5\omega-f_3\omega^3+f_7\omega^3$
$f^{[110]}(x)=f_0-f_4-f_2\omega^2+f_6\omega^2+f_1\omega^3-f_5\omega^3-f_3\omega^5+f_7\omega^5$
$f^{[111]}(x)=f_0-f_4-f_2\omega^2+f_6\omega^2-f_1\omega^3+f_5\omega^3+f_3\omega^5-f_7\omega^5$
$f(\omega^{(000)_2})$
$f(\omega^{(100)_2})$
$f(\omega^{(010)_2})$
$f(\omega^{(110)_2})$
$f(\omega^{(001)_2})$
$f(\omega^{(101)_2})$
$f(\omega^{(011)_2})$
$f(\omega^{(111)_2})$
Bit reversal permutation: $f(\omega^{(b_3b_2b_1)_2})=f^{[b_1b_2b_3]}(\omega)$

Remark 2. *FFT has been broadly used in sciences related to numerical computation rather than symbolic computation. The reason is just due to that its iterative description can be easily converted into its pseudocode (see Ref.[6]).*

4 Initerativable divide-and-conquer algorithms

4.1 Karatsuba's algorithm for polynomial multiplication

Karatsuba's algorithm for polynomial multiplication reduces the cost from the classical $\Theta(n^2)$ for polynomials of degree n to $\Theta(n^{1.59})$ in theory.

Suppose that $A(x) = \sum_{i=0}^{n-1} a_i x^i, B(x) = \sum_{i=0}^{n-1} b_i x^i \in \mathcal{R}[x]$ of degree less than n over a commutative ring $\mathcal{R}$ with 1, where $n = 2^l$ for some positive integer l. Write $A(x), B(x)$ as follows

$$A(x) = A_1(x)x^{\frac{n}{2}} + A_0(x), \quad B(x) = B_1(x)x^{\frac{n}{2}} + B_0(x),$$

where $A_1(x), A_0(x), B_1(x), B_0(x)$ are polynomials of degree less than $\frac{n}{2}$. Clearly,

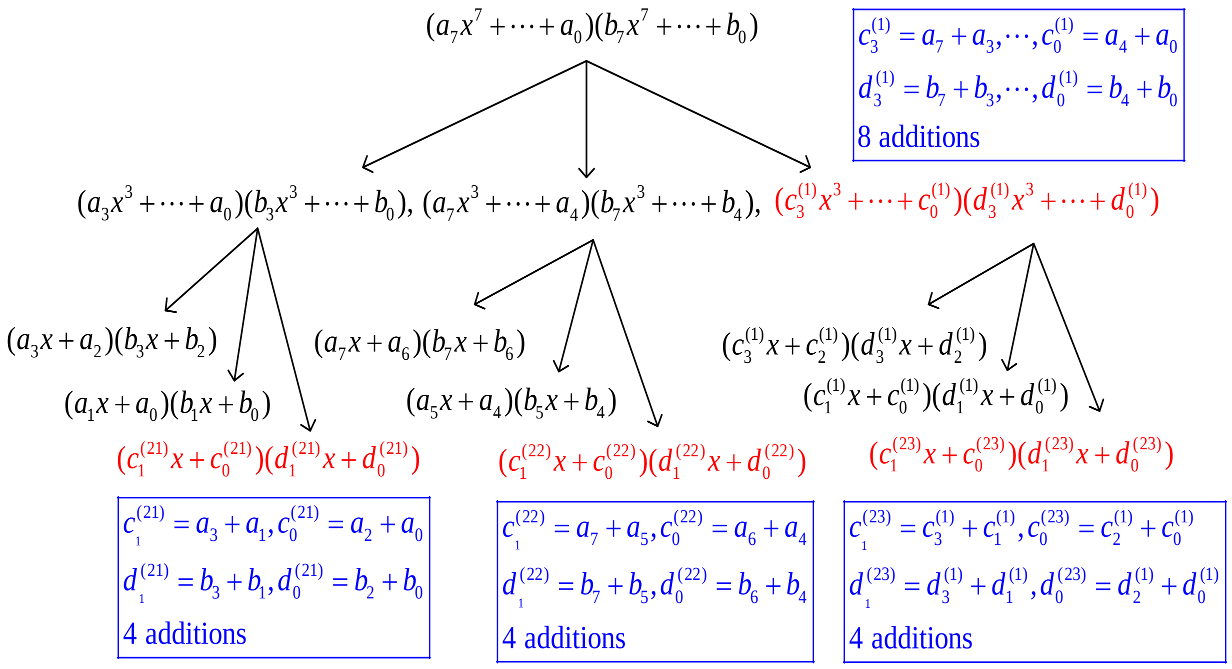
$$\begin{aligned} A(x)B(x) &= A_1(x)B_1(x)x^n + A_0(x)B_0(x) \\ &\quad +\{[A_1(x) + A_0(x)][B_1(x) + B_0(x)] - A_1(x)B_1(x) - A_0(x)B_0(x)\}x^{\frac{n}{2}} \end{aligned}$$

Hence, the problem of computing $A(x)B(x)$ is divided into the following three subproblems:

$$A_1(x)B_1(x), \quad A_0(x)B_0(x), \quad [A_1(x) + A_0(x)][B_1(x) + B_0(x)].$$

For example, taking $n = 8$, the working flow can be depicted as follows (see Fig. 5).

Figure 5: Working flow of Karatsuba's algorithm for $n = 8$



$(a_7x^7 + \cdots + a_0)(b_7x^7 + \cdots + b_0)$
$c_3^{(1)} = a_7 + a_3, \cdots, c_0^{(1)} = a_4 + a_0$
$d_3^{(1)} = b_7 + b_3, \cdots, d_0^{(1)} = b_4 + b_0$
8 additions
$(a_3x^3 + \cdots + a_0)(b_3x^3 + \cdots + b_0)$, $(a_7x^3 + \cdots + a_4)(b_7x^3 + \cdots + b_4)$, $(c_3^{(1)}x^3 + \cdots + c_0^{(1)})(d_3^{(1)}x^3 + \cdots + d_0^{(1)})$
$(a_3x + a_2)(b_3x + b_2)$
$(a_1x + a_0)(b_1x + b_0)$
$(c_1^{(21)}x + c_0^{(21)})(d_1^{(21)}x + d_0^{(21)})$
$(a_7x + a_6)(b_7x + b_6)$
$(a_5x + a_4)(b_5x + b_4)$
$(c_1^{(22)}x + c_0^{(22)})(d_1^{(22)}x + d_0^{(22)})$
$(c_3^{(1)}x + c_2^{(1)})(d_3^{(1)}x + d_2^{(1)})$
$(c_1^{(1)}x + c_0^{(1)})(d_1^{(1)}x + d_0^{(1)})$
$(c_1^{(23)}x + c_0^{(23)})(d_1^{(23)}x + d_0^{(23)})$
$c_1^{(21)} = a_3 + a_1, c_0^{(21)} = a_2 + a_0$
$d_1^{(21)} = b_3 + b_1, d_0^{(21)} = b_2 + b_0$
4 additions
$c_1^{(22)} = a_7 + a_5, c_0^{(22)} = a_6 + a_4$
$d_1^{(22)} = b_7 + b_5, d_0^{(22)} = b_6 + b_4$
4 additions
$c_1^{(23)} = c_3^{(1)} + c_1^{(1)}, c_0^{(23)} = c_2^{(1)} + c_0^{(1)}$
$d_1^{(23)} = d_3^{(1)} + d_1^{(1)}, d_0^{(23)} = d_2^{(1)} + d_0^{(1)}$
4 additions

Note that in Karatsuba's algorithm the problem decomposition does not take constant time. In fact, $D(n) = \Theta(n)$. Combining the solutions to the subproblems into the solution to the upper level problem takes time $C(n) = \Theta(n)$. Thus, the recurrence equation for Karatsuba's algorithm is

$$T(n) = 3T(n/2) + \Theta(n),\ n > 1.$$

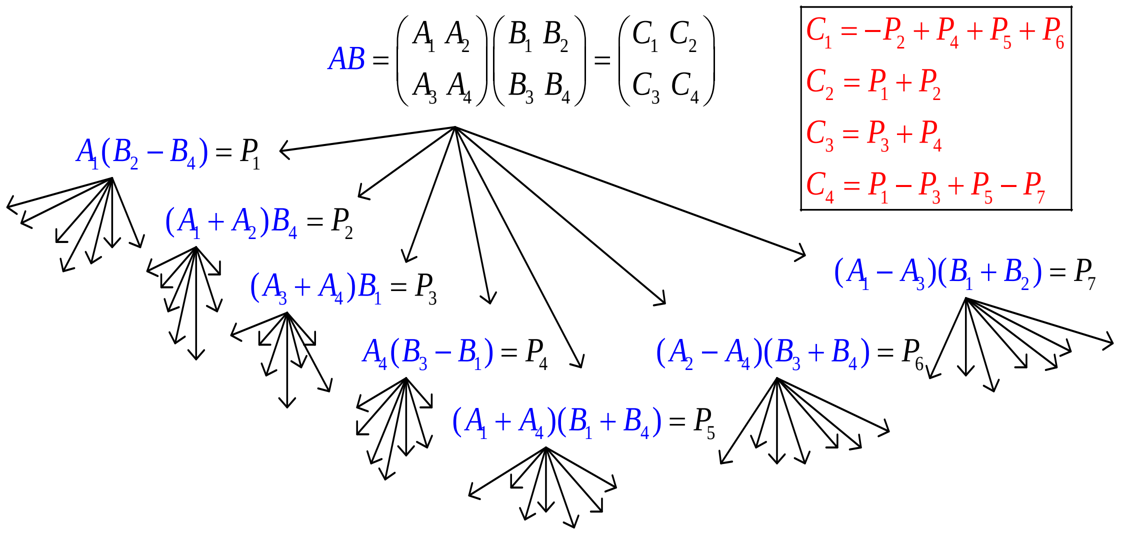
By "master theorem", we have $T(n) = \Theta(n^{\lg 3})$.

Remark 3. *In Karatsuba's algorithm the problem decomposition not only takes time $\Theta(n)$ (in terms of bit operations), but also consumes $\Theta(n^{\lg 3})$ space for storing the intermediate coefficients. The required space for Karatsuba's algorithm is expanding exponentially. In such case the incurred space is very significant, and the time required for "bookkeeping" or logical steps other than the bit operations, accumulates significantly.*

4.2 Strassen's algorithm for matrix multiplication

Strassen's algorithm for matrix multiplication reduces the cost from the classical $\Theta(n^3)$ for $n \times n$ matrixes to $\Theta(n^{\lg 7}) = \Theta(n^{2.81})$. The working flow of Strassen's algorithm can be depicted as follows (see Fig. 6).

Figure 6: Working flow of Strassen's algorithm


$AB = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}\begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$
$C_1 = -P_2 + P_4 + P_5 + P_6$
$C_2 = P_1 + P_2$
$C_3 = P_3 + P_4$
$C_4 = P_1 - P_3 + P_5 - P_7$
$A_1(B_2 - B_4) = P_1$
$(A_1 + A_2)B_4 = P_2$
$(A_3 + A_4)B_1 = P_3$
$A_4(B_3 - B_1) = P_4$
$(A_1 + A_4)(B_1 + B_4) = P_5$
$(A_2 - A_4)(B_3 + B_4) = P_6$
$(A_1 - A_3)(B_1 + B_2) = P_7$

The algorithm needs $\Theta(n^2)$ scalar additions and subtractions to compute 14 matrixes each of which is $n/2 \times n/2$. Clearly, the problem decomposition takes time $D(n) = \Theta(n^2)$. Combining the solutions to the subproblems into the solution to the upper level problem takes time $C(n) = \Theta(n^2)$. Thus, the recurrence equation for Strassen's algorithm is

$$T(n) = 7T(n/2) + \Theta(n^2),\ n > 2.$$

By "master theorem", we have $T(n) = \Theta(n^{\lg 7})$. The topics on practical implementations of the algorithm are discussed in [2, 7].

Remark 4. *The problem decomposition in Strassen's algorithm takes time $\Theta(n^2)$, and consumes $\Theta(n^{\lg 7})$ space for storing the intermediate matrixes. The incurred space for Strassen's algorithm is expanding exponentially. In such case the time required for "bookkeeping" or logical steps, cannot be neglected as usual.*

5 How to confirm an iterativable recursive algorithm

By investigating merge sort, FFT, Karatsuba's algorithm, Strassen's algorithm, etc, we find there are two types of divide-and-conquer algorithms: one type is iterativable, i.e, a recursive algorithm can be converted into its iterative version; the another type is initerativable. Iterativable divide-and-conquer algorithms can be efficiently and practically implemented, whereas initerativable algorithms have not been practically implemented so far.

We also find that whether the problem decomposition in a recursive algorithm takes constant time is of great significance to its implementation. If decomposition time of a recursive algorithm is constant, then the algorithm can be theoretically converted into its iterative version. Otherwise, the problem that data expand exponentially makes it impossible to accomplish the conversion.

6 Conclusion

We classify divide-and-conquer algorithms into two types, iterativable and initerativable. We find the iterativable algorithms can be efficiently implemented, whereas the initerativable algorithms can not be practically implemented because the time for dealing with data expansion problem counteracts the time saved by conquering each subproblems.

References

[1] A. Aho, J. Hopcroft and J. Ullman, The design and analysis of computer algorithm. Addison-Wesley, 1974.

[2] D. Bailey, K. Lee and H. Simon, Using Strassen's algorithm to accelerate the solution of linear systems. The Journal of Supercomputing, 4(4), 357-371, 1990.

[3] J. Cooley, J. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19, 297-301, 1965.

[4] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 2 edition, MIT Press, 2000.

[5] J. Gathen and J. Gerhard, Modern computer Algebra, 3 edition, Cambridge University Press, 2003.

[6] D. Knuth: The Art of Computer programming, Vol. 3, Sorting and Searching, second edition, Addison-Wesley, 1998.

[7] S. Huss-Lederman, et al., Implementations of Strassen's algorithm for matrix multiplication. In SC96 Technical Papers, 1996.

[8] A. Levitin, Introduction to the Design and Analysis of Algorithms, Addison Wesley, 2002.

[9] V. Shoup, A Computational Introduction to Number Theory and Algebra, Cambridge University Press, 2005.