Introduction to Computational Complexity Theory

Sumant Hegde

August 18, 2014

Contents

Abstract

This article aims to provide motivation for the subject of Compuational Complexity Thoery(CCT), using some interesting algorithmic problems. At the end of the article we define the scope of the subject. The article is also a scribing of lecture 1 on the subject of CCT.

1 Multiplication of two n-bit numbers

Let us discuss various methods of multiplying two numbers and analyze their efficiency. Instead of two n-bit (binary) numbers we may sometimes use two n-digit (decimal) numbers if that conveys an idea more easily.

1.1 Using repeated addition

The most basic algorithm of multiplication implements the very definition of multiplication, i.e. *repeated addition*. So, given two numbers a and b, the algorithm obtains $a * b$ by adding a to 0 b times (cumulatively). If a and b are of n-digits (or bits) each, then each step of addition contains $\Theta(n)$ digit-wise (or bitwise) operations. So the total number of basic operations becomes $a * n = 2^n * n \in O(2^n)$

1.2 High school algorithm

Obviously, repeated addition is very inefficient. A better algorithm, taught in schools, takes advantage of the place-value notation of numbers as well as the distributive property of multiplication, as follows
If a and b are the two numbers such that
$a = a_{n-1}X^{n-1} + a_{n-2}X^{n-2} + ... + a_1X + a_0$ and
$b = b_{n-1}X^{n-1} + b_{n-2}X^{n-2} + ... + b_1X + b_0$,
then
$a \times b = ab_{n-1}X^{n-1} + ab_{n-2}X^{n-2} + ... + ab_1X + ab_0$ where X is the base.
For example, let $a = 12 = 1 \cdot 10^1 + 2 \cdot 10^0$ and $b = 34 = 3 \cdot 10^1 + 4 \cdot 10^0$. Then

$$\begin{array}{r} \frac{12\times 34}{48} \\ \underline{36} \\ 408 \end{array} \tag{1}$$

Here, ab_0 is 48 and ab_1X is 360. The algorithm just calculates ab_1 (here 36), and shifts it to the left hand side by one digit-place, to achieve multiplication by X (here 10). In general, ab_k is shifted to the left hand side by k places, achieving multiplication by X^k to make ab_kX^k.

If the numbers being multiplied have different number of digits(bits), the smaller number can be padded with necessary number of 0's and the above algorithm can be run.

There are several variants of the algorithm. For example, instead of writing down partial products(rows) with appropriate shifts and summing them up at the end, one can compute individual digit of the *final* answer on each iteration, preferably starting from the rightmost digit for easy handling of carry. (Champions of "fast multiplication tricks" recommend the latter approach, pointing out that you do not have to memorize partical products as much.) Nevertheless the total number of multiplications and additions will remain the same.

Complexity. Computing each *row*, that is, partial product, requires n multiplication operations. There are n such rows. Hence there are n^2 multiplication operations. Further, at the end we have roughly n^2 digits to be added. Therefore it requires $O(n^2)$ multiplications + $O(n^2)$ additions.

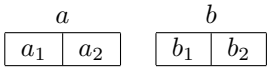
Multiplication of two digits is a constant time operation - and hence is no more expensive than addition of two digits - provided we have the multiplication table. Thus overall complexity is in $O(n^2)$, significantly better than repeated-addition algorithm.

For a long time it was believed that $\Theta(n^2)$ was the lower bound, until Karatsuba invented a better algorithm described below

1.3 Karatsuba algorithm

Karatsuba algorithm uses divide-and-conquer technique along with a trick that saves one multiplication operation out of four, at the cost of a few extra addition(/subtraction) operations. Since addition is far less expensive than multiplication (especially for large numbers), all in all, we gain.

Given two n-bit numbers a and b, split them in the middle to get a_1 and a_2, and b_1 and b_2 respectively, each of size roughly $n/2$ bits:



In other words,
$a = a_1 2^{n/2} + a_2$ and
$b = b_1 2^{n/2} + b_2$.

With this setup, the naive divide-and-conquer approach would work on the 4 small chunks, namely, a_1, a_2, a_3 and a_4, while requiring 4 multiplication steps, as

$$\begin{aligned} a \times b &= (a_1 \cdot 2^{n/2} + a_2) \cdot (b_1 \cdot 2^{n/2} + b_2) \\ &= a_1 b_1 \cdot 2^n + (a_1 b_2 + a_2 b_1) \cdot 2^{n/2} + a_2 b_2 \end{aligned} \tag{2}$$

We can show that this still runs in quadratic time, by solving the recurrence for (2), which is

$$T(n) = 4T(n/2) + cn \tag{3}$$

where c is a constant such that cn accounts for all the additions and shifting (for multiplying by 2^n) of components in (2).
From the Master theorem, the complexity of (3) is in $O(n^{\log_2 4})$ or $O(n^2)$.
Karatsuba algorithm improves the situation by rewriting (2) as follows

$$\begin{aligned} a \times b &= a_1 b_1 \cdot 2^n + (a_1 b_2 + a_2 b_2) \cdot 2^{n/2} + a_2 b_2 \\ &= a_1 b_1 \cdot 2^n + ((a_1 + b_1) \times (a_2 + b_1) - a_1 b_1 - a_2 b_2) \cdot 2^{n/2} + a_2 b_2 \end{aligned} \tag{4}$$

Crucially, the middle term now needs only one new multiplication operation: $(a_1 + b_1) \times (a_2 + b_1)$, reducing the total number of multiplication operations to 3. (Products $a_1 b_1$ and $a_2 b_2$ were anyway going to be calculated, and we are reusing them here.) Of course, there are extra addition (of n-bit numbers) and subtraction (of 2n-bit numbers) operations now, which increase the size of constant (to C, say) in the recurrence:

$$T(n) = 3T(n/2) + Cn \tag{5}$$

Nevertheless, from the Master theorem, the runtime of (5) is in $O(n^{\log_2 3})$ or $O(n^{1.585})$, an asymptotic improvement certainly.

There are generalized versions of this algorithm. For example, Toom-Cook algorithm splits n-bit multiplicands into 3 (or more) components of size n/3 (or less) each, and applies divide-and-conquer.

1.4 Schonhage-Strassen algorithm

This algorithm is based on Fast Fourier Transforms(FFT) and runs in time $O(n \log n \log\log n)$. Another recent algorithm, called *Furer's algorithm*, also is based on FFT and runs in time $n \log n 2^{O(\log^* n)}$.

1.5 Conclusion

Contrary to the initial — and reasonable — feeling that the lower bound is quadratic, there are subquadratic algorithms available. However, while better algorithms are getting discovered time and again, the question what is the theoretical lower bound for multiplication remains unanswered.

2 GCD of two n-bit numbers

2.1 by factorization

Calculation of GCD by factorization is the most basic method, as it just follows from the definition of GCD. Given two n-bit numbers a and b, this method writes a and b as products of prime factors:
$FACTOR(a) = p_{a1}p_{a2}...p_{aj}$ — p_{ai} is a prime number and
$FACTOR(b) = p_{b1}p_{b2}...p_{bk}$ — p_{bi} is a prime number
It then picks common elements from collections $FACTOR(a)$ and $FACTOR(b)$. The product of common elements(factors) is the GCD.
Complexity. Factorizing a number, say a, requires at least checking for divisibility by numbers from 2 to $\sqrt{a}$. Thus, factorizing an n-bit number takes $O(2^{n/2})$ time. So without doing rigorous analysis of complexity for checking whether a factor is prime or not, or the complexity analysis of selecting common factors from the two colloections, we can declare that the algorithm runs in exponential time.

2.2 Euclid's algorithm

Euclid's algorithm makes use of a property of a GCD: if a, b are integers and $a < b$ then $g = GCD(a, b) = GCD(a, b \mod a)$.

To see why this is true, let $b \mod a = c$. Then $b = ak + c$ where k is an integer. Now, g divides b by the definition of GCD. g also divides a and hence ak. Therefore g has to divide c too according to the rules of divisibility.

Algorithm 1 Euclid's algorithm

```
1: procedure EUCLID(a, b)
2:     r ← b mod a
3:     while r ≠ 0 do
4:         b ← a
5:         a ← r
6:         r ← b mod a
7:     end while
8:     return a                                  ▷ The gcd is a
9: end procedure
```

complexity. The worst case occurs when the algorithm converges very slowly. For this to happen, at every step, a and b should be such that

$$b \mod a = b - a \tag{6}$$

In other words, if a_i and b_i represent the numbers at hand on the i^th iteration, the worst case occurs when $(a_{i+1}, b_{i+1}) = (b_i, a_i + b_i)$. This is nothing but the case of 2 consecutive numbers of Fibonacci series (with possibly different base values). Hence the number of steps required for the algorithm to converge is roughly equal to the rank of the fibonacci number nearest to b, which is given by

$$n(F) = \lfloor \log_\phi(\sqrt{5} + 1/2) \rfloor \tag{7}$$

Hence,

$$T(a, b) \leq \log_\phi(\sqrt{5}b) \tag{8}$$

2.3 conclusion

Even for a notion like GCD, whose definition is based on factors, which are hard to compute, there is an algorithm significantly efficient than the factorization algorithms: It improves the runtime from exponential time to poly-time. However, again, it is not clear whether there can exist an asymptotically better algorithm for the problem.

3 Solving a system of n linear equations of n variables

Let us represent the system of n linear equations of n variables as follows
$a_{11}X_1 + a_{12}X_2 + ... + a_{1n}X_n = b_1$
$a_{21}X_1 + a_{22}X_2 + ... + a_{2n}X_n = b_2$
$\vdots$
$a_{n1}X_1 + a_{n2}X_2 + ... + a_{nn}X_n = b_n$

Equivalently there is a matrix representation, as follows
$AX = B$, where

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}, \quad X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}, \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

3.1 Gaussian elimination

In Gaussian elimination, we convert A into an *upper triangular matrix* by applying *elementary operations on equations(rows)*, like replacing an equation with a nonzero multiple of it, or with a sum of or difference between this and another equation etc. These operations are guaranteed to *not* alter the solutions. Also, each such operation can be done in linear time with respect to n. Since the operations should be applied on the RHS of the equation as well, matrix B may also change. The resultant system (with altered A and B) will look like the following
$A'X = B'$, where

$$A' = \begin{bmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix}, \quad X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}, \quad B' = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

Now, the value of X_n is readily available: it's b'_n. By substituting b'_n on $(n-1)^t h$ equation we can find X_{n-1}. By continuing the process of *backward substitution* we can solve for all X_i's.

complexity. The elimination step will apply at most $n/2$ operations on A, to make all the $n(n-1)/2$ elements of the lower triangle of A zero. Each of these operations manipulates one equation (or two), which means $\Theta(n)$ elements. Hence the run time of forward elimination step is in $O(n^3)$.

The step of backward substitution will work on the resultant upper triangular matrix. Its runtime is in $O(n^2)$, as it applies one constant-time operation (either substitution and/or addition) on each of the $n(n+1)/2$ elements in the upper triangular matrix (plus n more elements of matrix B', strictly speaking). So, the overall runtime is dominated by the elimination step, which takes $O(n^3)$ time.

3.2 Strassen's matrix multiplication

Alternatively, the system of equations could be solved by finding inverse of the matrix A, since if $AX = B$ then $X = A^{-1}B$. It is said that inversion of a matrix is computationally equivalent to multiplying two matrices. Hence it is sufficient to examine matrix multiplication algorithms.

The brute force algorithm that multiplies 2 $m \times m$ matrices A and B, performs n^3 multiplications and $n^3 - n^2$ additions:

$$M(n) = 2n^3 - n^2. \tag{9}$$

In particular, for 2×2 matrices, 8 multiplications and 4 additions are required. Strassen's algorithm reduces this to 7 multiplications while increasing the number of additions to 18, using the formulas mentioned below:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix} \tag{10}$$

where

$m_1 = (a_{00} + a_{11}) \cdot (b_{00} + b_{11})$
$m_2 = (a_{10} + b_{11}) \cdot b_{00}$
$m_3 = a_{00} \cdot (b_{01} - b_{11})$
$m_4 = a_{11} \cdot (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) \cdot b_{11}$
$m_6 = (a_{10} - a_{00}) \cdot (b_{00} + b01)$
$m_7 = (a_{00} - a_{11}) \cdot (b_{10} + b11)$

With a 2×2 matrix it may seem that the total number of scalar operations has increased ($7+18 = 25$ as opposed to $8+4 = 12$ of the brute-force algorithm), but the importance of Strassen's algorithm becomes clearer once we realize that the algorithm uses divide-and-conquer, as follows.

Let A and B be two $n \times n$ matrices. Then A, B and C where $C = AB$, each is divided into 4 $n/2 \times n/2$-sized submatrices:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \cdot \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right] \tag{11}$$

Notice the similarity between (10) and (11). Indeed, Strassen's algorithm recursively computes first the subproducts $A_{00}, A_{01}, A_{10}, A_{11}, B_{00}, B_{01}, B_{10}$ and B_{11}. Then it applies 7 multiplications and 18 additions/subtractions on them to compute AB, along the lines of (10).

complexity. 7 multiplications of matrices of size $n/2 \times n/2$ (recursive part), and 18 additions of matrices of size $n/2 \times n/2$ (nonrecursive part) leads us to the following recurrence:

$$T(n) = 7T(n/2) + 18(n^2/4) \tag{12}$$

From the Master theorem, its runtime is in $O(\log_2 7)$.

3.3 Even better algorithms

Copporsmith-Winograd is an improvement over Strassen's algorithm, and has asymptotic complexity of $O(n^{2.375477})$. The most recent algorithm is William's algorithm, whose complexity is in $O(n^2.3728)$.

3.4 Conclusion

The lower bound for matrix multiplication algorithms is conjectured to be $\Omega(n^2)$, where $n \times n$ is the matrix size. (One of the reasons to believe so: the product itself contains n^2 elements.) But yet, the gap between this bound and the best algorithm so far remains unresolved.

4 On proving lower bounds and impossibilities

If someone proves today that William's algorithm *is* the best matrix multiplication algorithm ever possible, then, whether good news or bad news, all computer scientists can stop spending time improving the algorithm. Rather, one can build on that result and come up with some other proofs. Generalizing the above observation, the subject of complexity theory recognizes the importance of proving lower bounds and impossibilities of various algorithms. We have already discussed lower bounds in length in the previous sections. Below we present some examples of proven impossibilities

1. The proof that Euclid's 5 th postulate (a.k.a the parallel postulate) cannot be derived from the first 4 postulates. The proof not only settled a centuries-old debate, but also opened up a new branch in geometry, now known as non-Euclidian geometry.
2. The proof that there is no formula for solutions of equations of degree ≥ 5.
3. The proof that there is no algorithm to solve a system of polynomial equations with integer coefficients, over integers.

At the same time, we equally emphasize on the attempts to improve algorithms whose theoretical lower bound is not yet proven. For it was Karatsuba algorithm that disproved the long-believed conjecture that $O(n^2)$ was the theoretical minimum for the multiplication of n-bit numbers. Another motivation for improvement is, of course, large gap between a proven lower bound and a so-far-the-best algorithm.

5 The course: Computational Complexity Theory

Complexity theory is a subject that classifies problems based on the amount of resources used in solving these problems. The resources may be time, space, randomness, communication etc.

5.1 Syllabus

1. The computational model (Turing machine)
2. NP and NP-Completeness
3. Diagonalization and relativization
4. Space complexity
5. Polynomial time hierarchy
6. Boolean ciruits
7. Randomized computation
8. Interactive proofs
9. Introduction to PCP theorem and hardness of approximation
10. Complexity of counting