Question 8: analysis of algorithms for multiplication

Given two binary numbers x, y with n-bits each, give an efficient algorithm to calculate the product z = xy.
Analyze the time complexity of this algorithm.

Solution:
Let's look at another problem first:
The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

\[(a + bi)(c + di) = ac - bd + (bc + ad)i\]

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and \((a + b)(c + d)\), since:

\[bc + ad = (a + b)(c + d) - ac - bd.\]

In our big-O way of thinking, reducing the number of multiplications from 4 to 3 seems wasted ingenuity. But this modest improvement becomes very significant when applied recursively.

Let’s move away from complex numbers and see how this helps with regular multiplication. Suppose x and y are two n-bit integers; and assume for convenience that n is a power of 2.

Method 1: Grade-school Multiplication
Multiplying two numbers of n bits each, in grade-school method, is actually multiplying and n-bit number by 1 bit, n times, and summing up the products. Time: \(\Theta(n^2)\)

Example (no need to get into):

\[
\begin{array}{c}
\times \\
10110110 \\
10011101
\end{array}
\begin{array}{c}
\times \\
182_{10} \\
157_{10}
\end{array}
\]

\[
\begin{array}{c}
10110110 \\
00000000 \\
10110110 \\
00000000 \\
00000000 \\
10110110
\end{array}
\begin{array}{c}
182_{10} \\
157_{10}
\end{array}
\]

\[
\begin{array}{c}
10110110 \\
00000000 \\
10110110 \\
00000000 \\
00000000 \\
10110110
\end{array}
\begin{array}{c}
182_{10} \\
157_{10}
\end{array}
\]

\[
\begin{array}{c}
110111110011110
\end{array}
\begin{array}{c}
28574_{10}
\end{array}
\]

Method 2: Recursive Multiplication
As a first step towards multiplying \( x \) and \( y \), split each of them into their left and right halves, which are \( n/2 \) bits long:

\[
x = \begin{bmatrix} x_L \quad x_R \end{bmatrix} = 2^{n/2}x_L + x_R
\]

\[
y = \begin{bmatrix} y_L \quad y_R \end{bmatrix} = 2^{n/2}y_L - y_R.
\]

For instance, if \( x = 10110110 \) then \( x_L = 1011, x_R = 0110 \),
and \( x = 1011 \times 2^4 + 0110 \). The product of \( x \) and \( y \) can then be rewritten as:

\[
x y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2} (x_L y_R + x_R y_L) + x_R y_R.
\]

Let’s analyze the runtime:

- Additions – \( O(n) \)
- Multiplications by powers of two (actually left-shifts) – \( O(n) \)
- Four \( n/2 \)-bit multiplications – \( x_L y_L, x_L y_R, x_R y_L, x_R y_R \) – with recursive calls.

Our method for multiplying \( n \)-bit numbers starts by making recursive calls to multiply these four pairs of \( n/2 \)-bit numbers (four sub-problems of half the size), and then evaluates the expression above in \( O(n) \) time.

Writing \( T(n) \) for the overall running time on \( n \)-bit inputs, we get:

\[
T(n) = 4T(n/2) + O(n).
\]

Using master method we get \( T(n) = \Theta(n^2) \) – no improvement to gradeschool method!

**Method 3: Improved Recursive Multiplication**

This is where Gauss’s trick comes to mind. Although the expression for \( xy \) seems to demand four \( n/2 \)-bit multiplications, as before just three will do: \( x_L y_L, x_R y_R, \) and \( (x_L+x_R)(y_L+y_R) \) since

\[
x_L y_R + x_R y_L = (x_L+x_R)(y_L+y_R) - x_L y_L - x_R y_R.
\]

And the runtime is improved to:

\[
T(n) = 3T(n/2) + O(n).
\]

Again, we use the master method to find \( T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.6}) \)

The point is that now the constant factor improvement, from 4 to 3, occurs at every level of the recursion, and this compounding effect leads to a dramatically lower time bound.
Pseudo-code:

```plaintext
function multiply(x, y)
    input: Positive integers x, y, in binary
    Output: Their product

    n = max(size of x, size of y)
    if n = 1: return xy

    xL, xR = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of x
    yL, yR = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ bits of y

    P1 = multiply(xL, yL)
    P2 = multiply(xL, yR)
    P3 = multiply(xL + xR, yL + yR)
    return P1 × 2^n + (P3 - P1 - P2) × 2^{n/2} + P2
```

Just FYI - We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to get even faster multiplication algorithms. Ultimately, this idea leads to the development of the Fast Fourier transform, a complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in O(n log n) time.

(Question 8 solution adapted from the book “Algorithms”
© 2006 by Dasgupta, Papadimitriou and Vazirani
Available online: http://www.cs.berkeley.edu/~vazirani/algorithms.html
And from lesson 1 of http://www.cs.uiuc.edu/~jeffe/teaching/algorithms// )