Introduction and Problem Statement

The problem we consider is extremely basic: how do we multiply two integers? In elementary school, you were taught a concrete (and fairly efficient) algorithm to multiply two \( n \)-digit numbers \( x \) and \( y \). You first compute a “partial product” by multiplying each digit of \( y \) separately by \( x \), and then you add up all the partial products, shifting where necessary.

\[
\begin{array}{c}
12 \\
\times 13 \\
\hline
36 \\
+12 \\
\hline
156 \\
\end{array}
\quad \begin{array}{c}
1100 \\
\times 1101 \\
\hline
1100 \\
0000 \\
+ 1100 \\
\hline
10011100 \\
\end{array}
\]

Figure 1: The elementary school algorithm for multiplying two integers in decimal and binary representation.

Note that while in elementary school, you learned how to multiply decimal numbers, in computer science, we care about binary numbers. Either way, the same algorithm works.

How long does this algorithm take to multiply two \( n \) bit numbers? Counting a single operation on a pair of bits as one primitive step in the computation, it takes \( O(n) \) time to compute each partial product, and \( O(n) \) time to combine it in with the running sum of partial products so far. Since there are \( n \) partial products, this gives a total running time of \( O(n^2) \).

In fact, there exists a faster algorithm which uses the divide and conquer strategy to achieve a much better runtime.

Designing the Algorithm

The improved algorithm is based on a more clever way to break up the product into partial sums. Let’s assume we are in base 2 (it doesn’t really matter) and we are given two binary numbers \( x \) and \( y \) which we want to multiply. We start by writing \( x \) as

\[
x = x_1 \cdot 2^{n/2} + x_0
\]

In other words, \( x_1 \) corresponds to the \( n/2 \) higher-order bits and \( x_0 \) corresponds to the \( n/2 \) lower-order bits. Similarly, write \( y = y_1 \cdot 2^{n/2} + y_0 \). Then we have

\[
xy = \left(x_1 \cdot 2^{n/2} + x_0\right) \left(y_1 \cdot 2^{n/2} + y_0\right)
= x_1 y_1 \cdot 2^n + \left(x_1 y_0 + x_0 y_1\right) \cdot 2^{n/2} + x_0 y_0
\]
Hence we have reduced the problem of solving a single $n$-bit instance (multiplying the two $n$-bit numbers $x$ and $y$) into the problem of solving four $n/2$-bit instances (computing the products $x_1y_1, x_1y_0, x_0y_1$, and $x_0y_0$). So we have a first candidate for a divide-and-conquer algorithm: recursively compute the results for these four $n/2$-bit instances, and then combine them using the above equation. The combining requires a constant number of additions and shifts of $O(n)$-bit numbers, so it takes $O(n)$ time. Thus the running time is given by the recurrence

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

(where we have ignored the base case for simplicity). Here $c$ is just a constant.

Unfortunately, you can check that solving this recurrence gives us $T(n) = \Theta(n^2)$, which is no better than the elementary school algorithm!

In order to speed up our algorithm, we can try to reduce the number of recursive calls made from four to three. In that case, we’d have $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.59})$ which is much better than $\Theta(n^2)$.

Now in order to compute the value of the expression $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ using only three recursive calls instead of four, we need to use a little trick. Remember, as part of our computation, we need to compute $(x_1y_0 + x_0y_1)$. In particular, we don’t need to explicitly compute $x_1y_0$ and $x_0y_1$ separately: we just need their sum!

Hence consider the result of the single multiplication $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. This has the four products above added together, at the cost of a single recursive multiplication. If we now also determine $x_1y_1$ and $x_0y_0$ by recursion, then we get the outermost terms explicitly, and we get the middle term by subtracting $x_1y_1$ and $x_0y_0$ away from $(x_1 + x_0)(y_1 + y_0)$.

This gives the full algorithm:

**Fast Integer Multiplication**

*Input:* Two $n$-bit numbers $x$ and $y$, where $n$ is assumed to be a power of 2.

*Output:* The product $xy$

**Recursive-Multiply** $(x, y)$

Let $x = x_1 \cdot 2^{n/2} + x_0$

Let $y = y_1 \cdot 2^{n/2} + y_0$

Let $a = x_1 + x_0$

Let $b = y_1 + y_0$

Let $c = \text{Recursive-Multiply} (a, b)$

Let $p = \text{Recursive-Multiply} (x_1, y_1)$

Let $q = \text{Recursive-Multiply} (x_0, y_0)$

Return $p \cdot 2^n + (c - p - q) \cdot 2^{n/2} + q$

**Runtime**

As stated above, the running time is $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.59})$. This comes from the following recurrence:

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

where $c$ is a constant and where we ignore the base case.
We can then solve the recurrence (ignoring the constant \( c \)) as follows:

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

\[
= 3 \left( 3T(n/2^2) + n/2 \right) + n = 3^2T(n/2^2) + \frac{3}{2}n + n
\]

\[
= 3^2 \left( 3T(n/2^3) + n/2^2 \right) + \frac{3}{2}n + n = 3^3T(n/2^3) + \left( \frac{3}{2} \right)^2n + \frac{3}{2}n + n
\]

\[
\vdots
\]

\[
= 3^kT(n/2^k) + n \cdot \sum_{i=0}^{k-1} \left( \frac{3}{2} \right)^i
\]

Assuming a base case of \( T(1) \leq 1 \), the recurrence bottoms out when \( n/2^k = 1 \Rightarrow k = \lg n \). Plugging this in gives:

\[
3^kT(n/2^k) + n \cdot \sum_{i=0}^{k-1} \left( \frac{3}{2} \right)^i = 3^k + n \cdot \left( \left( \frac{3}{2} \right)^k - 1 \right) \left( \frac{1}{\frac{3}{2} - 1} \right)
\]

\[
= 3^k + n \cdot \left( \frac{3^{\lg n} - 1}{\frac{3}{2} - 1} \right)
\]

\[
= n^{\lg 3} + 2n \cdot (n^{\lg 3-1} - 1)
\]

\[
= n^{\lg 3} + 2n^{\lg 3} - 2n
\]

\[
= 3n^{\lg 3} - 2n
\]

\[
= \Theta(n^{\lg 3})
\]

**Problems**

**Element index matching**

**Problem 1.** You are given a sorted array of \( n \) distinct integers \( A[1...n] \). Design an \( O(\lg n) \) time algorithm that either outputs an index \( i \) such that \( A[i] = i \) or correctly states that no such index \( i \) exists.

**Solution.**

**Algorithm 1** Modified Binary Search

1. procedure FINDINDEXMATCHINGELEM(A[...])
2. \( l \leftarrow 0 \) \( \triangleright \) left bound pointer
3. \( r \leftarrow \text{length}(A) - 1 \) \( \triangleright \) right bound pointer
4. while \( l \leq r \) do \( \triangleright \) ”\( \leq \)” takes care of single elem case
5. \( m \leftarrow (l + r)/2 \) \( \triangleright \) midpoint
6. if \( A[m] == m \) then
7. \( \quad \text{return } m \)
8. else if \( A[m] < m \) then
9. \( \quad l \leftarrow m + 1 \)
10. else
11. \( \quad r \leftarrow m - 1 \)
12. \( \quad \text{return } -1 \)
Local maximum

Problem 2. You are given an integer array with the following properties:

- Integers in adjacent positions are different
- \( A[\text{length} - 2] > A[\text{length} - 1] \)

A position \( i \) is referred to as a local maximum if \( A[i] > A[i - 1] \) and \( A[i] > A[i + 1] \).

Example: You have an array \([0, 1, 5, 3, 6, 3, 2]\). There are multiple local maxes at 5 and 6.

Propose an efficient algorithm that will find a local maximum and return its index.

Solution. While we can solve this problem quite easily in linear time by running through the array and checking each element with its neighbors until we find a possible solution, there is a more efficient way of solving this problem.

The first thing to notice is that there must be a local maximum in the array (Why must this be the case?). Run through a few examples and convince yourself of this fact.

If we want to approach this problem with the Divide & Conquer methodology, we need to figure out how we can cut the problem into subproblem(s). Does this remind you of any technique? (Hint: binary search.) Of course, we need to apply a slight modification to the original binary search algorithm.

Let’s take the middle element of the array. If that element meets the requirements for a local maximum, then great, we are done. We can rejoice and go sleep soundly. However, this may not be the case. What can we do? We can compare the element to its rightmost neighbor (let’s call it \( x \)). If \( x \) is bigger than our current element, what do we know? Well, there must be a solution in the right half of the array! Why? Similar to our previous reasoning that the array must have some solution, we know that if \( x \)'s neighbor is smaller than \( x \), then \( x \) is a local max! Otherwise, we have the same case until we reach the end of the array (and the second to last element is bigger than the last). Using this logic, we know that if our middle element is greater than \( x \), our solution lies in the left half.

The running time of our algorithm is thus \( T(n) = T(\frac{n}{2}) + O(1) \). Solving this recurrence yields \( O(\log n) \) (it is binary search after all!).

Algorithm 2 Modified Binary Search

```
1: procedure LOCALMAX(arr[...])  \( \triangleright \) finds index of a local maximum
2:  \( l \leftarrow 0 \)  \( \triangleright \) left pointer
3:  \( r \leftarrow \text{length}(arr) - 1 \)  \( \triangleright \) right pointer
4:  while \( l < r \) do
5:    \( m \leftarrow (l + r) / 2 \)
6:    if \( arr[m] > arr[m - 1] \) \&\& \( arr[m] > arr[m + 1] \) then
7:      return \( m \)
8:    else if \( arr[m] < arr[m + 1] \) then  \( \triangleright \) cut array in half and move right
9:      \( l \leftarrow m + 1 \)
10:   else  \( \triangleright \) cut array in half and move left
11:     \( r \leftarrow m - 1 \)
12:  return \( l \)
```