Integer Multiplication

Motivation: multiply two 30-digit integers?
153819617987625488624070712657
x 925421863832406144537293648227

- Multiply two 300-digit integers?
- Cannot do this in Java with built-in data types
- 64-bit unsigned integer can only represent integers up to ~20 digits ($2^{64} \approx 10^{20}$)

Warm-Up: Addition

Input: two $n$-digit binary integers $x$ and $y$
Goal: compute $x + y$

Let's do everything in base-10 instead of binary to make examples more familiar.
Grade-school algorithm:

```
1854
+ 3242
-------
5096
```

Running time? $\Theta(n)$

Grade-School Algorithm (Long Multiplication)

Example: $n = 3$

```
287
x 132
------
574
861
287
-------
37884
```

$287 \times 132 = (2 \times 287) + 10 \cdot (3 \times 287) + 100 \cdot (1 \times 287)$

Running time? $\Theta(n^2)$

But $xy$ has at most $2n$ digits. Can we do better?

Divide and Conquer: First Try

Idea: split $x$ and $y$ in half (assume $n$ is a power of 2)

```
x = 3380 2367
  x1 y0

y = 4508 1854
  y1 y0
```

Then use distributive law

$$xy = (10^{n/2}x_1 + x_0) \times (10^{n/2}y_1 + y_0)$$
$$= 10^n x_1 y_1 + 10^{n/2}(x_1 y_0 + x_0 y_1) + x_0 y_0$$

Have reduced the problem to multiplications of $n/2$-digit integers and additions of $n$-digit numbers
Divide and Conquer: First Try

Recursive algorithm:

\[ xy = 10^n x_1 y_1 + 10^{n/2} (x_1 y_0 + x_0 y_1) + x_0 y_0 \]

Running time? Four multiplications of \( n/2 \) digit numbers plus three additions of at most \( n \)-digit numbers

\[
T(n) \leq 4T\left(\frac{n}{2}\right) + cn \\
= O(n \log_2 4) \\
= O(n^2)
\]

We did not beat the grade-school algorithm. :(

Better Divide and Conquer

Total: three multiplications of \( n/2 \)-digit integers, six additions of at most \( n \)-digit numbers

\[
T(n) \leq 3T\left(\frac{n}{2}\right) + cn \\
= O(n \log_2 3) \\
\approx O(n^{1.59})
\]

We beat long multiplication!

Idea can be generalized to be even faster (split \( x \) and \( y \) into \( k \) parts instead of two)

Finding Minimum Distance between Points

> **Problem 1**: Given \( n \) points on a line \( p_1, p_2, \ldots, p_n \in \mathbb{R} \), find the closest pair: \( \min_{i \neq j} |p_i - p_j| \).
> 1. Compare all pairs \( O(n^2) \)
> 2. Sort the points and compare adjacent pairs \( O(n \log n) \)

> **Problem 2**: Now what if the points are in \( \mathbb{R}^2 \)?
> 1. Compare all pairs \( O(n^2) \)
> 2. Sort? Points can be close in x-coordinate and far in y, and vice-versa.
> 3. We’ll do it in \( O(n \log n) \) steps using divide-and-conquer.

Minimum Distance Algorithm

> **Input**: set of points \( P = \{ p_1, \ldots, p_n \} \) where \( p_i = (x_i, y_i) \)
> **Assumption**: we can iterate over points in order of \( x \)- or \( y \)-coordinate in \( O(n) \) time. Pre-generate data structures to support this in \( O(n \log n) \) time.

Recursive Algorithm

1. Find vertical line \( L \) to divide points into sets \( P_L \) and \( P_R \) of size \( n/2 \), \( O(n) \)
2. Recursively find minimum distance in \( P_L \) and \( P_R \):
   > \( \delta_L = \) minimum distance between \( p, q \in P_L, p \neq q \). \( T(n/2) \)
   > \( \delta_R = \) same for \( P_R \). \( T(n/2) \)
3. \( \delta_M = \) minimum distance between \( p \in P_L, q \in P_R \). ??
4. Return \( \min(\delta_L, \delta_R, \delta_M) \).

Naive Step 3 takes \( \Omega(n^2) \) time. But if we do it in \( O(n) \) time we get

\[
T(n) \leq 2T(n/2) + O(n) \implies T(n) = O(n \log n)
\]
Making Step 3 Efficient

- **Goal**: given \( \delta_L, \delta_R \), compute \( \min(\delta_L, \delta_R, \delta_M) \)

- **Observation**: Let \( \delta = \min(\delta_L, \delta_R) \). If \( p \in P_L, q \in P_R \) differ by at least \( \delta \) in either the \( x \)- or \( y \)-coordinate, they cannot be the overall closest pair, so we can ignore the pair \((p, q)\).

- Let \( S \) be the set of points within distance \( \delta \) from \( L \). We only need to consider pairs that are both in \( S \).

- For a given point \( p \in S \), how many points \( q \) are within \( \delta \) units of \( p \) in the \( y \) coordinate?
  - **Claim**: at most 12
  - **Algorithm**: iterate through points \( p \in S \) in order of \( y \) coordinate and compare \( p \) to 12 adjacent points in this order. \( O(n) \).

- Intuition: the set \( S \) is “nearly one-dimensional”. Points cannot be packed in tightly, because two points on the same side of \( L \) are at least distance \( \delta \) apart. **Proof sketch on board.**