We start to understand our first paradigm of algorithm design, **Divide and Conquer**. D&C algorithms follow this strategy:

- Break the problem into subproblems that are smaller instances of the same problem.
- Recursively solve these subproblems.
- Merge (hopefully efficiently) solutions of the subproblems into solutions of the big problem.

**Notes.**
- Often D&C is well-suited to problems for which brute-force search is already polynomial (e.g. finding the closest pair of points among \( n \) in the plane).
- Small improvements in steps of D&C can add up to material improvements in the overall running time.

We focus now on an example of this second point.

**Integer multiplication** Recall that if \( x \) and \( y \) each have \( n \) bits then the grade-school algorithm for computing \( x \cdot y \) has running time \( O(n^2) \).

**Problem 1.** Run the algorithm to compute \( 1100 \times 1101 \), just to make sure you remember how it works. (You are more familiar with it in base-10, but it works the same way in base-2.)

**Problem 2.** Suppose that we are trying to multiply the two \( n \)-bit numbers \( x \) and \( y \). Let \( x_L \) be the first \( n/2 \) bits of \( x \) and \( x_R \) the last \( n/2 \) bits. Similarly, let \( y_L \) be the first \( n/2 \) bits of \( y \) and \( y_R \) the last \( n/2 \) bits.

(a) Write two equations, one that expresses \( x \) in terms of \( x_L \) and \( x_R \) and another that expresses \( y \) in terms of \( y_L \) and \( y_R \).

(b) Multiply your two equations to get an expression for \( xy \) in terms of the four products \( x_Ly_L, x_Ly_R, x_Ry_L, \) and \( x_Ry_R \) of two \((n/2)\)-bit integers.

(c) This suggests a D&C solution, since, using the previous part of the problem, you can compute \( xy \) using four recursive calls to compute \((n/2)\)-bit instances. Explain why the worst-case running time \( T(n) \) of this algorithm on inputs of size \( n \) satisfies the recurrence \( T(n) = 4T(n/2) + O(n) \).

(Unfortunately, as we’ll see, functions \( T(n) \) of this type are \( \Theta(n^2) \). So we have not improved on the grade-school algorithm.)

**Problem 3.**

(a) After meditating on the equation

\[
ab + cd = (a + b)(c + d) - ac - bd,
\]

show that in fact three recursive calls to compute products of \((n/2)\)-bit numbers would suffice in Problem 2(c).

(b) Write your algorithm in pseudocode and explain why its worst-case running time \( T(n) \) satisfies the recurrence \( T(n) = 3T(n/2) + O(n) \).

(c) By analyzing the tree of recursive calls, show that \( T(n) \) is \( O(n^{\log_2 3}) = O(n^{1.59}) \), which is sub-quadratic!

(Hint: You will probably need a standard log trick: \( n^{\log_a b} = a^{\log_a n} \).)

1You’re right! Actually maybe one of them is a product of two \((n/2 + 1)\)-bit integers. But that doesn’t affect the asymptotic analysis, luckily.
Strassen’s Trick for matrix multiplication  A similar trick allows us to speed up matrix multiplication a bit. Suppose that we want to multiply two $n \times n$ matrices, $X$ and $Y$, each of which we divide into four $n/2 \times n/2$ blocks:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}. $$

**Problem 4.** The product $XY$ can be computed *blockwise*. Fill in the remaining entries:

$$XY = \begin{bmatrix} AE + BG \\ \end{bmatrix}$$

This suggests a D&C strategy for multiplying matrices: recursively compute eight $n/2 \times n/2$ matrix products $AE$, $BG$, etc., and do a few $O(n^2)$ additions of $n \times n$ matrices. Unfortunately this gives $O(n^3)$ running time, the same as the usual linear algebra algorithm.

Strassen’s trick allows us to get away with only 7 multiplications. They are these:

$$P_1 = A(F - H) \quad P_5 = (A + D)(E + H)$$
$$P_2 = (A + B)H \quad P_6 = (B - D)(G + H)$$
$$P_3 = (C + D)E \quad P_7 = (C - A)(E + F)$$
$$P_4 = D(G - E)$$

**Problem 5.** Pick a couple of entries of $XY$ and show that they can be computed by adding and subtracting some of the seven Strassen products $P_1, \ldots, P_7$.

**Problem 6.** Explain how we can compound these savings into an algorithm for multiplying $n \times n$ matrices whose worst-case running time $T(n)$ satisfies the recurrence $T(n) = 7T(n/2) + O(n^2)$ and is therefore $O(n \log_2 7) = O(n^{2.807})$. 