Divide & conquer:

Classic example: MergeSort

**Input:** array \( A = [a_1, \ldots, a_n] \) of \( n \) numbers

**Output:** sorted \( A \) (assume \( n \) is a power of 2)

**Idea:** split \( A \) into 2 sublists, recursively sort each sublist, then merge the sorted sublists.

\[
\text{MergeSort}(A) \\
\text{if } n = 1, \text{ return } (A) \\
\text{if } n > 1, \\
\text{ let } B = [a_1, \ldots, a_{n/2}] \\
\text{ let } C = [a_{n/2+1}, \ldots, a_n] \\
D = \text{MergeSort}(B) \\
E = \text{MergeSort}(C) \\
F = \text{Merge}(D, E) \\
\text{Return}(F)
\]
Merge takes 2 sorted arrays \( X \) & \( Y \) & outputs sorted \( Z = X \cup Y \).

**Idea:** take \( \min \{ X_i, Y_j \} \) & then remove & repeat.

\[ \text{Merge}(X, Y) : \]

**Input:** \( X = [x_1, \ldots, x_k] \) & \( Y = [y_1, \ldots, y_l] \)

where \( X \) & \( Y \) are both sorted.

**Output:** Sorted \( Z = [z_1, \ldots, z_{k+l}] = X \cup Y \)

\( i = 1, j = 1, m = 1. \)

while \( (i \leq k \& j \leq l) \):

if \( x_i \leq y_j \) then \( Z_m = x_i, i++, m++ \)

else \( Z_m = y_j, j++, m++ \)

if \( i = k \), return \( (Z, y_j, \ldots, y_l) \)

if \( j = l \), return \( (Z, x_i, \ldots, x_k) \).

**Running time of Merge:** \( O(k + l) \) time.
For MergeSort, let \( T(n) \) = running time on worst case input for \( n \) numbers.

Then, \( T(n) = 2T(n/2) + O(n) \)

Base case: \( T(1) = O(1) \)

We'll see that this solves to: \( T(n) = O(n \log n) \)

In this class we assume basic arithmetic operations (add, multiply, divide) take \( O(1) \) time since we can use hardware implementation.

But for cryptography HUGE # of bits \( \approx 1000 \).

Let \( n \) = # of bits in the input numbers.

What is time for arithmetic operations as a function of \( n \)?
Adding 2 n-bit numbers \( x \& y \)

Example: \( x = 53 = (110101)_2 \)
\( y = 35 = (100011)_2 \)
\[
\begin{array}{c}
110101 \\
+ 100011 \\
\hline
1011000
\end{array}
\]
\( \leq n+1 \) columns \& \( \leq 3 \) bits/column
\( \Rightarrow O(n) \times O(1) = O(n) \) total time.

Multiplying n-bit \( x \& y \)

Easy: \( x + x + \ldots + x \) \( y \) terms
\( \Rightarrow \) takes \( O(ny) \) time
but \( y \leq 2^n \) so \( O(n2^n) \).

Grade school algorithm is better:

Example: \( x = 13 = (1101)_2 \)
\( y = 11 = (1011)_2 \)
\[
\begin{array}{c}
1101 \\
\times 1011 \\
\hline
1101 \\
11010 \\
+ 000000 \\
1101000 \\
\hline
1000111
\end{array}
\]
\( \Rightarrow O(n) \times O(n) = O(n^2) \) time

Is this the best?
No, we'll do faster.
Alternative algorithm from Al-Khwarizmi, mathematician in Baghdad in 9th century AD who wrote books on algorithms, e.g., solving quadratic equations. The term "algorithms" comes from his name.

Take input \( x \) & \( y \)

1) Halve \( y \) (round down) & double \( x \)
2) Stop when \( y = 1 \).
3) Cross out rows where \( y \) is even.
4) Add remaining \( x \)'s.

Example: \( x = 13 \) \hspace{1cm} \( y = 11 \)

\[
\begin{array}{c}
13 \\
26 \\
52 \\
+ 104 \\
\hline
143 \\
\end{array}
\]

\[
\begin{array}{c}
11 \\
5 \\
2 \\
1 \\
\hline
\end{array}
\]

Why does it work?

Note traditional algorithm:

\[ \begin{align*}
1101 &= 13 \\
11010 &= 26 \\
00000 &= 0 \text{ b/c 3rd least significant bit of } y \text{ is 0}
\end{align*} \]

+ \[
1101000 = 104
\]

So the 2 algorithms are the same.

Faster approach using Divide & Conquer.
Assume \( n \) is a power of 2 (can pad with 0s & ≤ double the size)

Input: \( n \)-bit numbers \( x \) & \( y \).

Divide & Conquer idea:

Break input into 2 halves

So \[
X = \begin{bmatrix} X_L & X_R \end{bmatrix}
\]

\( 1^{st} \frac{n}{2} \) bits last \( \frac{n}{2} \) bits

\[
Y = \begin{bmatrix} Y_L & Y_R \end{bmatrix}
\]

for example, if \( x = 182 = (10110110) \) then \( \begin{align*}
X_L &= 1011 = 11 \\
X_R &= 0110 = 6
\end{align*} \)

\[ x = 11 \times 2^4 + 6 = 182 \]
in general, \( x = 2^{n/2} x_L + x_R \)

So, \( x = 2^{n/2} x_L + x_R \) & \( y = 2^{n/2} y_L + y_R \)

Then,
\[
xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R)
\]
\[
= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R
\]

Easy idea:

Recursively compute \( x_L y_L \)
\( x_L y_R \)
\( x_R y_L \)
\( x_R y_R \)

Then get \( xy \) by adding & subtracting

\[
\text{EasyMultiply}(x, y) :
\]
\[
x_L = \text{1st } \frac{n}{2} \text{ bits of } x, \quad x_R = \text{last } \frac{n}{2} \text{ bits of } x
\]
\[
y_L = \text{1st } \frac{n}{2} \text{ bits of } y, \quad y_R = \text{last } \frac{n}{2} \text{ bits of } y
\]
\[
\alpha = \text{EasyMultiply}(x_L y_L)
\]
\[
\beta = \text{EasyMultiply}(x_L y_R)
\]
\[
\gamma = \text{EasyMultiply}(x_R y_L)
\]
\[
\delta = \text{EasyMultiply}(x_R y_R)
\]

Return \( 2^n \alpha + 2^{n/2} (\beta + \gamma) + \delta \)
Running time:
\[ T(n) = 4T\left(\frac{n}{2}\right) + O(n) \]
which solves to \[ T(n) = O(n^2) \]
So no faster.

Idea of Gauss:
2 complex numbers \((a+bi)\) & \((c+di)\)

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

This seems to need 4 real number multiplications:
ac, bd, bc, ad

But: \(bc + ad = (a+b)(c+d) - ac - bd\)
So can do with only 3:
ac, bd, \((a+b)(c+d)\)
Back to multiplying $x \& y$:

- Let $a = x_L, b = x_R, c = y_L, d = y_R$

Then $bc + ad = (a + b)(c + d) - ac - bd$

- $x_Ry_L + x_Ly_R$
- $(x_L + x_R)(y_L + y_R)$
- $x_Ly_L$
- $x_Ry_R$

Thus,

$$x_Ry_L + x_Ly_R = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R$$

So recursively solve: $x_Ly_L, x_Ry_R \& (x_L + x_R)(y_L + y_R)$

**Fast Multiply** $(x, y)$:

- $x_L = \text{first } \frac{n}{2} \text{ bits of } x$ & $x_R = \text{last } \frac{n}{2} \text{ bits of } x$
- $y_L = \text{ } y$ & $y_R = \text{ } y$

- $\alpha = \text{Fast Multiply}(x_L, y_L)$
- $\beta = \text{Fast Multiply}(x_R, y_R)$
- $\gamma = \text{Fast Multiply}(x_L + x_R, y_L + y_R)$

Return $2^n \alpha + 2^n \beta (\gamma - \alpha - \beta) + \beta$
Running time:

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

\[ \log_2 3 \approx 1.59. \]