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.

MergeSort(A)

if $n = 1$, return(A)

if $n > 1$,

let $B = [a_1, \ldots, a_{\frac{n}{2}}]$

let $C = [a_{\frac{n}{2}+1}, \ldots, a_n]$

D = MergeSort(B)

E = MergeSort(C)

F = Merge(D, E)

Return(F)

Merge takes 2 sorted arrays X & Y
& outputs sorted $Z = X \cup Y$.
idea: take $\min\{X_1, Y_1\}$ & then remove & repeat

Merge(X, Y):

input: $X = [X_1, \ldots, X_k]$ & $Y = [Y_1, \ldots, Y_\ell]$
where X & Y are both sorted

output: sorted $Z = [Z_1, \ldots, Z_{k+\ell}] = X \cup Y$

$i = 1$, $j = 1$, $m = 1$.
while $(i \le k$ & $j \le \ell)$:
    if $X_i \le Y_j$ then $Z_m = X_i$, $i{+}{+}$, $m{+}{+}$
        else $Z_m = Y_j$, $j{+}{+}$, $m{+}{+}$
if $i == k$, return$(Z, Y_j, \ldots, Y_\ell)$
if $j == \ell$, return$(Z, X_i, \ldots, X_k)$.

Running time of Merge: $O(k+\ell)$ time.

For MergeSort,

let $T(n)$ = running time on worst case input for n numbers.

Then, $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

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 ≈ 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: $x = 53 = (110101)_2$
$y = 35 = (100011)_2$

$$\begin{array}{r} 110101 \\ +\ 100011 \\ \hline 1011000 \end{array}$$

$\leq n+1$ columns & ≤ 3 bits/column

$\Rightarrow O(n) \times O(1) = O(n)$ total time.

Multiplying n-bit x & y

easy: $\underbrace{x + x + \cdots + x}_{y \text{ terms}}$ takes $O(ny)$ time
but $y \leq 2^n$ so $O(n2^n)$.

Grade school algorithm is better:

Example: $x = 13 = (1101)_2$
$y = 11 = (1011)_2$

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 11010 \\ +\ 000000 \\ 1101000 \\ \hline 10001111 \end{array} \quad \left.\right\} \text{adding } n \text{ numbers each} \leq 2n-1 \text{ bits}$$

$\Rightarrow O(n) \times O(n) = O(n^2)$ 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.

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: $x = 13$ $y = 11$

x	y
13	11
26	5
~~52~~	~~2~~
+ 104	1
143	

Why does it work?

Note traditional algorithm:

$$
\begin{array}{rl}
1101 & = 13 \\
11010 & = 26 \\
000000 & = 0 \\
+\ 1101000 & = 104 \\
\hline
\end{array}
$$

b/c 3rd least significant bit of y is 0

So the 2 algorithms are the same.

Faster approach using Divide & conquer.

Assume n is a power of 2 (can pad with 0's & $\leq$ double the size)

Input: n-bit numbers x & y.

Divide & conquer idea:

Break input into 2 halves

So $x = [\ x_L \mid x_R\ ]$

1st $\frac{n}{2}$ bits | last $\frac{n}{2}$ bits

$y = [\ y_L \mid y_R\ ]$

for example, if $x = 182 = (10110110)$ then $x_L = 1011 = 11$
$x_R = 0110 = 6$
$x = 11 \times 2^4 + 6 = 182$

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

EasyMultiply(X, Y):

X_L = 1st $\frac{n}{2}$ bits of X, X_R = last $\frac{n}{2}$ bits of X
Y_L = 1st $\frac{n}{2}$ bits of Y, Y_R = last $\frac{n}{2}$ bits of Y

$$\left.\begin{aligned} \alpha &= \text{EasyMultiply}(X_L, Y_L) \\ \beta &= \text{EasyMultiply}(X_L, Y_R) \\ \gamma &= \text{EasyMultiply}(X_R, Y_L) \\ \delta &= \text{EasyMultiply}(X_R, Y_R) \end{aligned}\right\} 4T(n/2)$$

Return$(2^n \alpha + 2^{n/2}(\beta + \gamma) + \delta)$ ← $O(n)$ time

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 $$\underbrace{bc + ad}_{x_R y_L + x_L y_R} = \underbrace{(a+b)(c+d)}_{(x_L + x_R)(y_L + y_R)} - \underbrace{ac}_{x_L y_L} - \underbrace{bd}_{x_R y_R}$$

Thus, $$x_R y_L + x_L y_R = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

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

FastMultiply(x, y):

$x_L = 1^{st}\ \frac{n}{2}$ bits of x & $x_R =$ last $\frac{n}{2}$ bits of x

$y_L =$ " y & $y_R =$ " y

$\alpha =$ FastMultiply(x_L, y_L)

$\beta =$ FastMultiply(x_R, y_R)

$\gamma =$ FastMultiply$(x_L + x_R, y_L + y_R)$

Return$\left(2^n \alpha + 2^{n/2}(\gamma - \alpha - \beta) + \beta\right)$

Running time:

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

$$\log_2 3 \approx 1.59$$