What if n=1???

$n = 1$

What problems are interesting when n is just 1?

Sorting? No

Median finding? No

Addition?

"How long does it take to add one pair of numbers?"

Multiplication?

"How long does it take to multiply one pair of numbers?"

Primality?

"How long does it take to determine with a given number is prime?"

It's all about the digits…

For problems such as these, the runtime is proportional to the number of digits in the numbers.

We rephrase our questions as:

"How long does it take to add one pair of ***d***-digit numbers?"

"How long does it take to multiply one pair of ***d***-digit numbers?"

"How long does it take to determine with a given ***d***-digit number is prime?"

Adding *d*-digit numbers

We will count any ***digit-level*** pairwise math operation as our unit of work since comparisons don't really make sense here.

Grade-school math says we can:

- work from right to left adding digits one column at a time (if there is a carry we add it in too)
- if there are ***d***-digits, there are d columns, so certainly less than 3***d***, and looking at it carefully it can be done in 2***d***, and with custom operators maybe even 1***d*** – either way, the runtime is O(***d***)

Can we do this with fewer than ***d*** operations? No, so it is Θ(***d***).

Multiplying d-digit numbers

Again, we will count any digit-level pairwise math operation as work since comparisons don't really make sense here.

Grade-school math says we can:

- for each position in the second number, work from right to left multiplying it against the digits in the first number and then add all of the resulting numbers
- if there are ***d***-digits, there are ***d*** columns, so the runtime of this is at least d^2

Note: Division has similar issues.

Can we do this with fewer than $O(d^2)$ operations?

Primality Testing runtime?

We could start looping 2 to n-1 and testing whether that value divides our number.

We could loop from 2 to *sqrt*(n) by basic math.

We could iterate through a list of all known primes from 2 to *sqrt*(n) by some slightly more advanced math.

Since this requires that list of all known primes up to that point, randomly picking odd numbers is used in some places, using probable primes by others.

However, these depend on dividing!

Fermat primality test

We won't go into this approach here (I won't call it an algorithm) but relating things back to an earlier topic, there is something called the "Fermat primality test" which is a faster test but is sometimes wrong.

Numbers tested with this approach where a factor is not found are called "probably prime" numbers.

"Solid Math" can help too

Consider the following for algorithm #2:

- The quotient-remainder theorem tells us all integers can be written as 6k+d where k is an integer and d is 0, 1, 2, 3, 4, or 5.
- We could easily prove that:
  2 divides (6k+0), (6k+2), (6k+4)
  3 divides (6k+3)

So, with a little bit of CMSC250 cleverness, we know that if our target isn't divisible by 2 or 3, we only need to test it against numbers that can be written as (6k+1) or (6k+5) for a savings of a constant factor of 3.

Faster Multiplication (and Division)

There are ways to do division based on how we do multiplication, but we will focus on multiplication for conceptual reasons…

Divide&Conquer Multiplying

Without loss of generality, let's talk in terms of ***binary*** numbers.

Let's say we want to multiply $\boldsymbol{a}$ and $\boldsymbol{b}$. We can write $\boldsymbol{a}$ as $\boldsymbol{a_1 a_2}$ where $\boldsymbol{a_1}$ is the first half of $\boldsymbol{n}$ bits and $\boldsymbol{a_2}$ is the second half of the bits and do the same for $\boldsymbol{b}$.

It turns out that we could try a clever divide and conquer approach:

$$\boldsymbol{ab} = \boldsymbol{a_1 b_1} 2^{\boldsymbol{n}} + (\boldsymbol{a_1 b_2} + \boldsymbol{a_2 b_1}) 2^{0.5\boldsymbol{n}} + \boldsymbol{a_2 b_2}$$

The question is, does this actually HELP us at all?

Divide&Conquer Multiplying

$$ab = a_1b_1 2^n + (a_1b_2 + a_2b_1)2^{0.5n} + a_2b_2$$

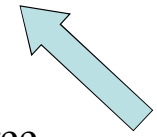
This equation has four multiplications on (n/2)-bit numbers but also has some other multiplication on larger-bit numbers. However, we can be clever in two ways:

- avoid having to multiply a_1b_1 by 2^n but rather add the $a_1b_1 2^n$ and a_2b_2 terms by concatenating the a_1b_1 and a_2b_2 terms
- avoid having to multiply by $2^{0.5n}$ if we instead shift the sum of the $a_1b_2 + a_2b_1$ value over the appropriate number of bits before adding to the a_1b_1 a_2b_2 concatenation result

We know addition is linear in the number of bits, so we get M(n) = 4M(n/2)+cn

M(n) = 4M(n/2)+cn

What does M(n) = 4M(n/2)+cn work out to for runtime?



If you do a quick recursion-tree analysis (work it out for practice) you will find that the runtime is n^2. So, it is perhaps clever but it is no improvement so far…

Getting *really* clever…

We want to multiply $\boldsymbol{a}$ and $\boldsymbol{b}$. We wrote $\boldsymbol{a}$ as $\boldsymbol{a}_1\boldsymbol{a}_2$ and $\boldsymbol{b}$ as $\boldsymbol{b}_1\boldsymbol{b}_2$.

We could say:

$$\boldsymbol{ab} = \boldsymbol{a}_1\boldsymbol{b}_1 2^{n} + (\boldsymbol{a}_1\boldsymbol{b}_2 + \boldsymbol{a}_2\boldsymbol{b}_1)2^{0.5n} + \boldsymbol{a}_2\boldsymbol{b}_2$$

but we could also get even ***more*** clever and say:

$$p_1 = \boldsymbol{a}_1\boldsymbol{b}_1$$

$$p_2 = \boldsymbol{a}_2\boldsymbol{b}_2$$

$$p_3 = (\boldsymbol{a}_1 + \boldsymbol{a}_2)(\boldsymbol{b}_1 + \boldsymbol{b}_2)$$

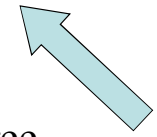
and make the product be:

$$\boldsymbol{ab} = \boldsymbol{p_1} 2^{n} + (\boldsymbol{p_3} - \boldsymbol{p_1} - \boldsymbol{p_2})2^{0.5n} + \boldsymbol{p_2}$$

We have now "replaced" one multiplication with some more addition and some subtraction (which can also be done in linear time like addition).

$M(n) = 3M(n/2)+sn$

What does $M(n) = 3M(n/2)+sn$ work out to for runtime?



If you do a quick recursion-tree analysis (work it out for practice) you will find that the runtime is $n^{\log 2(3)}$. So, some more cleverness which this time leads to a better asymptotic runtime class!

What about division?

There are ways to do it in the same runtime as multiplication…

How many bits?

If you wanted your program to be able to store individual unsigned integers between **1** and ***N***, how many **bits** would your variable need to have?

Answer: $\log_2(N)$

Time and Space

You are monitoring a feed of unique numbers and all numbers between **1** and ***N*** will come through the feed EXCEPT ONE OF THEM. You basically get to see it once and then it's gone into the ether.

- ***N*** will be very large and you are not allowed to use a data structure or file to hold all those values.
- How can you determine the missing number after having seen ***N*-1** numbers pass through the feed.
- How many bits of extra storage do you need?

What if TWO were missing?

Matrix Addition

Adding two 2x2 matrices:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

gives:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Where $c_{ij}=a_{ij}+b_{ij}$.

So, in general, if the matrix is *n*x*n*, an n^2 algorithm.

Matrix Multiplication

Multiplying two 2x2 matrices:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \bullet \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

gives:

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Where $c_{ij} = \sum_{k=1}^{2} (a_{ik} \cdot b_{kj})$

So, in general, if the matrix is nxn, an n^3 algorithm.

Run-Time Improvement?

Using the brute force algorithms that we saw earlier, we get runtimes of:

MM(n)=$\Theta(n^3)$

MA(n)=$\Theta(n^2)$

Do you think there's any way to improve either? If we thought about the lower bounds, what do they feel like they'd be?

Start with a small step…

If we wanted to multiply two 2x2 matrices it could easily be done using **8** multiplication operations.

Is there a way to do it using only **7** multiplication operations, and a *constant* number of *additional* addition operations?

If there were, would this help bring down the asymptotic runtime of full matrix multiplication?

Divide and Conquer

It turns out that matrix multiplication can be solved recursively by first imagining any ***n*** by ***n*** matrix as four smaller matrices, each being ***n/2*** by ***n/2*** and then recursively multiplying those.

So, if 2x2 matrix multiplication could be done using 7 multiplication and some number of additions, the overall problem would be…

MM(2) = constant amount of work

MM(n) = 7 MM($n/2$) + $\Theta(n^2)$

Is this better than $O(n^3)$? Work out the recurrence tree…

Strassen's algorithm

Since MATH240 isn't a prerequisite for this class we won't dive into the very clever Strassen algorithm for doing the 2x2 problem using only 7 multiplication operations, but it's easy to find online if you are curious…