Let's get rid of the π by looking at the function $\sin \pi x$, which has roots at $\pm n x$. Now, $\sin z$ is an **entire function**: It has no infinities – no **poles** – for any z, even for complex z. Polynomials also have no poles. An entire function is analogous to a polynomial: It is an infinite-degree polynomial. Others are e^z and $\sinh z$. Why is the analogy useful? Because your knowledge from the source system helps you generate ideas to use in the destination system. Polynomials are characterized by their zeros, so maybe entire functions are as well. For polynomials, that characterization is done by factoring them. So let's factor entire functions too.

How does $\sin \pi z$ factor? We already have a good idea.

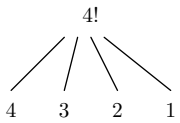
As we'll see in a later chapter, rational functions generalize to what are called **meromorphic functions** in complex analysis: functions with zeros and poles.

3.5 Example: Recursion

Sometimes you make a minilanguage to solve just one problem. The minilanguage or abstraction is reusable, and is reused multiple times in solving that problem. Recursion is an example of this use of abstraction.

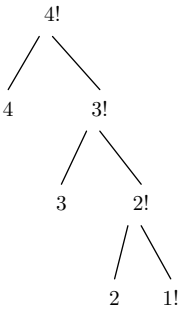
A classic example of recursion is computing $n!$. Here is a nonrecursive definition of factorial:

$$n! \equiv n \times (n-1) \times (n-2) \times \cdots \times 1.$$

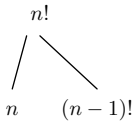


The tree illustrates how to compute 4! using this definition: You multiply 4, 3, 2, and 1.

Then I have a great insight: You notice that $3 \times 2 \times 1$ is also 3!, which is $3 \times 2!$, and so on. This realization turns the flat, seemingly unstructured tree into a tree with a pattern.



Here is the pattern (when $n > 1$). The Python code that implements this idea is




```
def fact(n):
  if n == 1:
    return 1
  else:
    return n * fact(n-1)
```

The tree approach, and the corresponding code, divides the computation of $n!$ into three parts:

1. digging up n, which is easy;
2. computing $(n-1)!$, whose details I don't care about because I know how to compute factorial; and
3. multiplying n and $(n-1)!$, which is easy.

The abstraction is reusable: It works not just for $4!$ but for $n!$ where n is any positive integer.

In keeping with the principle of telling lies first, and removing them later, I confess that multiplying n and $(n-1)!$ is not easy when n is large because then $(n-1)!$ is gigantic, larger than what the central processing unit of my computer can handle in its hardware. As a second example of recursion, I describe a quick way to multiply very large integers. For simplicity, I instead describe a way to square very large integers. In the problems, you get to generalize the method to multiplication of two different integers.

First, I square 35 using the common method, then using a fast method. I use base 10 and small examples to illustrate the methods.

Okay, the common method:

$$35^2 = (3 \times 10 + 5)^2 = (3 \times 10)^2 + 2 \times 3 \times 10 \times 5 + 5^2.$$

In a pictorial abstraction, where $3|5$ represents 35 and in general $x|y$ represents $10x + y$:

$$(3|5)^2 = 3^2|2 \times 3 \times 5|5^2,$$

where $x|y|z$ represents $100x + 10y + z$.

This method is not fast. To square $x|y$ requires squaring x, squaring y, and multiplying x and y (plus a few additions, but those are quick). But isn't that easy, since $x = 3$ and $y = 5$? In this case, it is easy. However, I

want the squaring algorithm to work for giant integers, where x and y are themselves giant integers. So, the algorithm will be used recursively.

Now I'll estimate S_n, the time required to square an n-digit number. The algorithm requires two squarings of $n/2$-digit numbers. It also requires multiplying two $n/2$-digit numbers (x and y). Then

$$S_n = 2S_{n/2} + M_{n/2},$$

where $M_{n/2}$ is the time required to multiply two $n/2$-digit numbers.

To estimate S_n, I need to estimate M_n. Using a similar algorithm as for squaring, multiplying two n-digit numbers involves four multiplications of $n/2$-digit numbers. So

$$M_n = 4M_{n/2}.$$

This recurrence has the solution $M_n \propto n^2$. Call the constant of proportionality A, so $M_n = An^2$.

Then the recurrence for S_n, the time to square an n-digit number, becomes

$$S_n = 2S_{n/2} + \frac{A}{4}n^2.$$

To solve this recurrence, I guess that squaring is not tremendously faster than multiplying. So S_n is not going to be proportional to n or even $n \log n$, and is likely to be proportional to n^2. This guess goes by the fancy name of an Ansatz.

Let B be the constant of proportionality: $S_n = Bn^2$. Then the recurrence for the squaring time becomes:

$$Bn^2 = 2\frac{B}{4}n^2 + \frac{A}{4}n^2.$$

The common n^2 factors divide out, leaving behind

$$B = \frac{B}{2} + \frac{A}{4},$$

whose solution is $B = A/2$. Since this equation is not nonsense, the guess is very likely to be valid. The result is that squaring using the common method is a quadratic operation (as is multiplying).

A slight variation in the common method makes it significantly faster. The problem with the common method is that it uses multiplication, which is quadratic (at least using a similar multiplication algorithm), and the slow method of multiplication contaminates the squaring algorithm. If only

there were a way to avoid multiplying! And there is! To square $x|y$, compute $(x|y)^2$ as follows:

$$(x|y)^2 = x^2|x^2 + y^2 - (x-y)^2|y^2.$$

This new method is significantly faster, as I show with the next estimate. Let S'_n be the time to square an n-digit number using this new method. It requires squaring three $n/2$-digit numbers: x, y, and $x - y$. So

$$S'_n = 3S'_{n/2}.$$

This recurrence has the solution

$$S_n \propto n^{\log_2 3} \approx n^{1.58}$$

The exponent is roughly 1.58 instead of 2. This small decrease has a large effect when n is large. For example, when multiplying billion-digit numbers, the ratio of n^2 to $n^{\log_2 3}$ is roughly 5000.

Why would anyone multiply billion-digit numbers? One answer is to compute π to a billion digits. Why would anyone do that? Computing π to a huge number of digits, and comparing the result with the calculations of other supercomputers, is one way to check the numerical hardware in a new supercomputer.

I haven't told the whole story. The fast algorithm, known as the Karatsuba algorithm after its inventor [18], is not used for absurdly huge numbers. For large enough n, an algorithm using fast Fourier transforms is still faster. The so-called Schönhage–Strassen algorithm [32] requires a time proportional to $n \log n \log\log n$. High-quality libraries for large-number mulitiplication use a combination of regular multiplication, Karatsuba, and Schönhage–Strassen, selecting the algorithm according to the size of the integer.

3.6 Spacetime

An abstraction can be so useful as to be unbelievable. An example is the concept of spacetime, introduced in a famous lecture on relativity given by the mathematician Hermann Minkowski [22]. Minkowski boldly announced (translation from [37]):

> From this hour on, space by itself and time by itself are to sink fully into the shadows and only a kind of union of the two should yet preserve autonomy.