Using Generating Functions to Solve Simple Recurrences

Richard C Kelley

August 28, 2008

Abstract

Generating functions are powerful tools for understanding the behavior of a sequence of numbers. By making the terms of the sequence the coefficients of a power series, we can often gain information about the sequence that would otherwise be hard to obtain. Best of all, when this method works, we can apply it in almost a completely mechanical fashion. In this paper we give a simple example that illustrates the spirit of working with generating functions.

1 Introduction

When we attempt to characterize an algorithm's resource usage in terms of space, time, bandwidth, or anything else for that matter, we often end up dealing (explicitly or implicitly) with sequences of numbers. For example, if we have an algorithm operating on arrays of numbers, we may want to know how many basic operations our algorithm performs on an input of size n. This measure is often used to describe the running time of an algorithm, which we'll call T. Moreover, since the running time depends on the input size in some way, we'll say that $T = T(n)$. This is the standard function notation, which is just fine. However, it's much more common to think of functions defined on the natural numbers as being *sequences*, so we'll do that and we'll use the standard notation for sequences instead of the function notation above. That is, we'll write $T = T_n$ to express the idea that the running time of our algorithm depends on the parameter n.

One of the goals of analysis of algorithms is to, for a given algorithm, come up with mathematical expressions that describe the resource usage of the algorithm. Often we only need to know this information at a very coarse level of detail. When this is the case we can just look at the asymptotic behavior of our algorithm and estimate its complexity using "big-oh" notation. Sometimes, though, this isn't good enough, and we'll want more precise estimates of resource usage. Ideally, such an estimate would be a closed form expression of the running time in terms of the input size, such as $T_n = n$ or $T_n = 2^n$. Generating functions give us one way to possibly obtain such expressions. Even when that isn't possible, the information that generating functions give us may still provide useful estimates. If you finish reading this and find the subject interesting, you can find a lot more about generating functions in computer science in [1]. Or look at some of the other items in the references.

Looking at things a bit more generally now, let's say we have a sequence of the form a_n, where $n = 0, 1, 2 \ldots$. Then we say that the *generating function* for the sequence a_n is

$$A(x) = \sum_{n=0}^{\infty} a_n x^n = \sum_{n \geq 0} a_n x^n.$$

So you see, a generating function is just an infinite series such that the coefficient of x^n is just a_n. This may seem like a really dumb thing to do, but this really is the most powerful way to deal with sequences of numbers. This is especially true when we aren't given the sequence explicitly, but are given a *recurrence*.

Before continuing, a somewhat technical point (which means you can skip to the next paragraph if you don't care about this sort of thing). If you remember your calculus, you know that infinite series of the kind described above aren't always *convergent*; that is, they don't always define a function from the real numbers to the real numbers. However, for our purposes this doesn't matter. We almost never care about convergence when we're working with generating functions, unless for example we're using complex analysis to estimate the coefficients of the series. So for most purposes we just ignore the question of convergence

and get on with our lives. If this sloppiness bothers you, rest assured that all of this business about infinite series can be put on a firm foundation by treating the infinite series as purely *formal* objects existing in what's called the *ring of formal power series* (which you would study in a course on abstract algebra).

An Example

To demonstrate the power of generating functions when dealing with recurrences, let's look at an example. Suppose we're working on a problem, and we come across the following recurrence: $a_0 = 1$ and $a_n = 2a_{n-1}$. Writing out a few terms of the sequence (always a good idea) shows us the answer *right* away: 1,2,4,8,16,.... As a computer scientist, you should instantly see that this is just the sequence of powers of two, so that $a_n = 2^n$. However, we can't always count on brilliant insight to lead us to the solution to our problems, so let's see if we can get the same answer using generating functions.

Start by defining our generating function to be $A(x) = \sum_{n \geq 0} a_n x^n$. We then go through the following steps:

1. Rewrite your recurrence so that a single equation captures all the parts in the definition of the recurrence.
2. Multiply each side of your recurrence by x^n.
3. Sum over all values of n for which the recurrence is defined.
4. You now have an expression in terms of infinite series - convert this into an equation involving the unknown generating function (in our example, $A(x)$).
5. Solve the *functional equation* you got in the previous step to find a closed-form expression for your generating function in terms of the variable x.
6. Expand the closed form expression from the previous step to (hopefully) get a closed-form expression for your sequence in terms of n.

Let's start with the first rule. We need a single equation to cover *both* the case where $n = 0$ and the case where $n > 0$. For the second case, we can use the equation $a_n = 2a_{n-1}$. But when $n = 0$ this equation is doesn't really make sense: a_{-1} isn't defined. However, if we agree that $a_n = 0$ for all $n < 0$, then we can get around that problem. Even so, the equation is still not true: $a_0 = 1$ and $2a_{-1} = 0$.

To get around this, we introduce a useful "trick." It's called *Iverson's convention*, and has a tendency to make life easier when working with summations. The convention to take any predicate (statement that can be true or false), enclose it in square brackets, and let it evaluate to 1 if the predicate is true and 0 if it's false. In other words, if $P(x)$ is our predicate, we have

$$[P(x)] = \begin{cases} 1 & \text{if } P(x) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Now we rewrite our recurrence as follows: We get rid of the special case $a_0 = 1$ and write the second case as $a_n = 2a_{n-1} + [n = 0]$. Then for $n \geq 1$, the term $[n = 0]$ evaluates to 0 and doesn't affect the recurrence. But for the case where n *is* 0, the equation still works out:

$$\begin{aligned} a_0 &= 2a_{0-1} + [0 = 0] && (1) \\ &= 2 * 0 + 1 && (2) \\ &= 1 && (3) \end{aligned}$$

So our single new equation agrees with our old pair of equations for all possible values of the input, which means that both approaches define the same sequence. All is well in the mathematical universe.

Now we can apply the next two steps of our procedure to get the following:

$$\sum_{n \geq 0} a_n x^n = 2 \sum_{n \geq 0} a_{n-1} x^n + 1.$$

(That second term on the right evaluates to 1 because it's $[n = 0]x^n$, which is zero for everything but $n = 0$ and 1 at $n = 0$.) The left side of that equation is easy: it's just $A(x)$. The right side requires a bit of work. In particular, we need to figure out how to express $\sum_{n\geq 0} a_{n-1}x^n$ in terms of our (currently unknown) function $A(x)$. But that's easy. We can pull an x out of the sum and make a change of variables (for n) to get the following:

$$\sum_{n\geq 0} a_{n-1}x^n = x\sum_{n\geq 0} a_{n-1}x^{n-1} \tag{4}$$

$$= x\sum_{n+1\geq 0} a_{(n+1)-1}x^{(n+1)-1} \tag{5}$$

$$= x\sum_{n\geq -1} a_n x^n \tag{6}$$

$$= x\sum_{n\geq 0} x_n x^n \tag{7}$$

$$= xA(x), \tag{8}$$

where we can go from equation 6 to equation 7 because when $n = -1$, the term $a_n x^n$ equals zero. (You agreed to that, remember?)

So now we have the functional equation

$$A(x) = 2xA(x) + 1,$$

which we can solve for $A(x)$ using just algebra to get

$$A(x) = \frac{1}{1-2x}.$$

So there. This is our generating function for the sequence a_n. It may not seem like it, but we now know a lot more about our sequence than we did when we started. For example, suppose we want to find a closed form expression for a_n. Then we can expand $A(x)$ as a power series in x to get our answer. There are a few ways to do this:

- Use calculus to figure out the answer from scratch.
- Know the answer
- Look it up.

The first method is the most reliable and versatile. In this case though, we can use the second or third methods (because we remember from calculus that $1/(1-x) = \sum x^n$) to see that

$$A(x) = \sum_{n\geq 0}(2x)^n$$
$$= \sum_{n\geq 0} 2^n x^n.$$

Since two formal power series are equal if and only if the coefficients on corresponding terms are equal, we can see from the equation

$$\sum_{n\geq 0} a_n x^n = \sum_{n\geq 0} 2^n x^n$$

that $a_n = 2^n$.

That probably seemed like a lot of work to come to a really simple conclusion. For this particular problem, it's probably true that the generating function approach is overkill. However, for more complicated problems, it's unlikely that you'll be able to just see the answer, and it may be the case that generating functions are the only way to go.

A Fun Problem

Consider a checkerboard with dimensions $2 \times n$. Suppose we want to cover this board with 2×1 tiles in such a way that the entire board is covered with tiles and no two tiles overlap. Clearly this can be done. The question here is: how many ways are there to do it? *Hint:* Let the number of ways to tile a $2 \times n$ board with 2×1 tiles be denoted f_n. Try answering the question for small values of n to find a pattern. (You'll probably see it right away. The real meat of this problem is to *prove* the relationship.) Define a recurrence for f_n, and *prove* that your recurrence is correct. One way to do this is by induction on n. Once you have your recurrence, see if you can find the generating function for your sequence, and expand the generating function you find as a power series to find a closed form (*non*recursive) expression for f_n in terms of n.

2 Further Reading

The study of generating functions is generally considered a part of the branch of mathematics called *Combinatorics*, or discrete math. You can find the above problem and a lot of other fun counting problems in [2]. Other equally fun but *much* more challenging problems (often involving more advanced math like algebra and topology) can be found in [3]. Another challenging book from a computer science perspective is the very entertaining [4]. If you're the practical type, you can find similar problems of an applied nature in [5].

References

[1] Donald Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1997.

[2] Richard Brualdi, *Introductory Combinatorics*, Prentice Hall, 2004.

[3] Richard Stanley, *Enumerative Combinatorics*, Cambridge University Press, 2001.

[4] Ronald Graham, Donald Knuth, Oren Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley Professional, 1994.

[5] Fred Roberts, Barry Tesman, *Applied Combinatorics*, Prentice Hall, 2003.