Computing Binomial Numbers by Recursion

Ted Szylowiec

1 Introduction

It's possible, of course, to compute binomial numbers using loops and the definition of binomials in terms of factorials:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{1}$$

But we want to find a more elegant way of computing them by recursion. The strategy for writing recursive programs is to first find a relationship that breaks down the problem into *smaller versions* of the *same* problem. Once we have this relationship, it's often easy to write the code that implements it.

However, if we try to write recursive programs without a clear idea of how our computation is broken up into smaller, similar computations, the task of programming becomes hopelessly confused. I think that this wrong approach is what many students try to do, hence their inability to write recursive procedures and their frustration with the concept of recursion itself.

In this article I will give two examples of writing recursive procedures by first looking for a relationship that breaks down the problem into smaller versions of the same problem.

2 The n/k identity

Many number theory books have chapters on binomial numbers and their peculiar properties. Most books on combinatorics do as well. The number of identities and relationships involving binomial numbers is bewildering. Despite that, we can look through some of these books to find an identity or relationship that has the right form: it breaks down the problem into smaller versions of the same problem. In other words, we are looking for an identity whereby binomial coefficients are written in terms of smaller binomial coefficients. And of course we want a simple one. Such as this one:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}. \tag{2}$$

As far as I know, it doesn't have a name. We can call it "the n/k identity." It is clear that (2) has the form we are looking for: the binomial number $\binom{n}{k}$ is computed in terms of a smaller binomial $\binom{n-1}{k-1}$.

Before we do anything with the n/k identity, let's prove it. First, an algebraic proof using (1).

$$\begin{aligned}\frac{n}{k}\binom{n-1}{k-1} &= \frac{n}{k}\,\frac{(n-1)!}{(k-1)!((n-1)-(k-1))!}\\ &= \frac{n}{k}\,\frac{(n-1)!}{(k-1)!(n-k)!}\end{aligned}$$

But $n(n-1)!$ is just $n!$ and $k(k-1)!$ is just $k!$, so we have

$$\frac{n}{k}\binom{n-1}{k-1} = \frac{n!}{k!(n-k)!} = \binom{n}{k}.$$

Next, a combinatorial proof. Suppose we have n people to choose from and we want to choose a team of r players along with a team captain. Choose a team first, and then choose a captain from the team members. There are $\binom{n}{k}$ ways to choose the team and k ways to choose the captain. In all there are

$$k \times \binom{n}{k}$$

possibilities. Now choose the captain first. There are n ways to do that. Once we have the captain, we must choose the other $k-1$ team members from the $n-1$ people left for us to choose from. There are

$$n \times \binom{n-1}{k-1}$$

ways to chose a captain and a team this way. Since the two ways we have described both count the same thing, they must be equal. Therefore (2) follows. This is a typical strategy in combinatorial proofs. Counting the same thing in two different ways implies both results must be equal.

The n/k identity gives us the right sort of decomposition for a recursive procedure. We keep applying it, making the binomial computation smaller and smaller, until we reach a point where the computation cannot be reduced further. These stopping points are sometimes called *base* cases. Here, the base case is at $k = 0$, where $\binom{n}{0} = 1$.

Begin our Racket program and add the unit testing library.

```
#lang racket
(require rackunit)
```

We should check if k is greater than n. If so, return 0 immediately. This check is not part of the recursion loop—it is just a check on the arguments. So we put it outside of `cond`. Our first recursive procedure, `binomial1`, follows effortlessly from the base case and the n/k identity:

```
(define (binomial1 n k)
  (when (> k n) 0)
  (cond ((= k 0) 1)
        (else (* (/ n k)
                 (binomial1 (- n 1)
                            (- k 1))))))
```

It is straightforward to make a tail-recursive version of `binomial1` by using a helper function and an accumulator variable, which we call `result`. When we hit the base case, `result` is returned.

```
(define (binomial2 n k)
  (define (helper n k result)
    (cond ((= k 0) result)
          (else (helper (- n 1)
                        (- k 1)
                        (* (/ n k)
                           result)))))
  (if (> k n)
      0
      (helper n k 1)))
```

We have several binomial functions to test. Each of them must pass a standard set of tests. To make testing easier, write a function `check-binomial` that can test any of them.

```
(define (check-binomial my-binomial)
  (check-equal? (my-binomial 1 3) 0)
  (check-equal? (my-binomial 0 0) 1)
  (check-equal? (my-binomial 1 0) 1)
  (check-equal? (my-binomial 1 1) 1)
  (check-equal? (my-binomial 6 0) 1)
  (check-equal? (my-binomial 6 6) 1)
  (check-equal? (my-binomial 6 3) 20))
```

Run the standard tests on `binomial1` and `binomial2`.

```
(check-binomial binomial1)
(check-binomial binomial2)
```

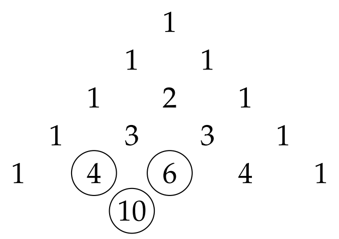
3 The Pascal identity

Looking through the books again, we find the following identity:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{3}$$

This is the most fundamental and most important of all binomial identities. It is a mathematical expression of the way we construct Pascal's triange: the current element is made of the sum of two elements in the row above.

For example, the element at $n = 5$, $k = 2$ is the sum of the two elements above it:



$$\binom{5}{2} = \binom{4}{1} + \binom{4}{2} = 4 + 6 = 10.$$

Let's prove (3) by a combinatorial argument. We have n objects. One of these objects is special, call it x. How many ways can we choose k objects such that x is not among them? It's the same as deleting x from our set of objects and choosing k from the remaining $n-1$ objects: $\binom{n-1}{k}$ ways. Next, how many ways can we choose k objects such that x is among them? Choose x first. We now have $n-1$ objects to choose from. Since we made one choice already, we have $k-1$ choices left. Total ways are $\binom{n-1}{k-1}$. The sum of all the ways with x and all the ways without x is just all the ways to choose k from n, i.e., $\binom{n}{k}$.

The Pascal identity (3) decomposes the computation of a binomial number into a computation of the sum of smaller binomial numbers. It makes a perfect foundation for a recursive procedure.

A recursive procedure based on (3) requries three base cases. When $k = 0$ and when $k = n$, the binomial number is 1. If you examine the right hand side of (3) you'll see that it's possible for n to become less than k during the process of the recursion. This is the third base case. Unlike what we did before, we must move the check for $k > n$ into the loop of the recursion and return 0 in the case of $k > n$.

The rest is a straightforward translation of (3) into Racket code:

```
(define (binomial3 n k)
  (cond ((> k n) 0)
        ((= k 0) 1)
        ((= k n) 1)
        (else (+ (binomial3 (- n 1) k)
                 (binomial3 (- n 1) (- k 1))))))
```

Let's not forget the tests!

```
(check-binomial binomial3)
```

□