Review of $A = B$
by Marko Petkovšek, Herbert S. Wilf, and Doron Zeilberger
Published by A.K. Peters, Ltd., Wellesley, MA, 1996, 212 pages
Hardcover, ISBN Number 1568810636

Reviewer: Vladik Kreinovich
University of Texas at El Paso
Department of Computer Science

Often, the problem of finding the computational complexity of a certain algorithm can be reduced to computing a combinatorial sum – i.e., a sum like this:

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

This is often true when we are interested in the worst-case complexity, and this is even more frequently true when we are looking for the average-case complexity. There are so many combinatorial sums in these problems that even writing them down is difficult: suffice it to say that Donald E. Knuth has spent so much time making these formulas look right in his famous *The Art of Computer Programming* books that after Volume 3, he decided to take a break from these books and develop an automated way to type such formulas – an effort which eventually led to TEX and its dialects.

Even teaching a computer to *write down* all these formulas is so tough a task that it requires a multi-year effort – well, teaching a computer to *derive* these formulas is even a greater challenge. Knuth himself formulated this challenge in the very first volume (*Fundamental Algorithms*) of his book, as Exercise 1.2.6.63: *Develop computer programs for simplifying sums that involve binomial coefficients.* Knuth gave this problem the rating of 50 – meaning that it is one of the toughest problems in the book. It was tough. For several decades, it avoided solution. The good news is that after 30 years of extraordinary efforts (largely the efforts of this book's authors), this problem is largely solved: there are algorithms, there are effective computer programs which make handling the cumbersome combinatorial sums unbelievably easy. This book, proudly starting with the formulation of the original Knuth's problem and with an enthusiastic preface by Knuth himself, is a culmination of these efforts. This book is about the problem, the history of its solution, the resulting algorithms, and finally – about the programs.

Actually, there are *two* problems here. The *original problem* is when we have an expression for the sum, and we would like to have a more compact expression for it. Usually, when we cannot immediately find an answer, a natural idea is to compute this sum for $n = 0, 1, 2, \ldots$, and try to guess the answer. Very often, the answer is easy to guess, and then we run into a *second problem:* to prove that this guess is indeed a correct formula for all n. For example, for the above sum (1), Knuth computed it for $n = 0, 1, 2, 3$, and came up with the values $1, 4, 16, 64$. A natural guess is that for every n, the desired sum is equal to 4^n, but proving that this sum is indeed always equal to 4^n turned out to be hard.

In general, the second problem is as follows: given two combinatorial expressions A and B, check whether they always coincide, i.e., whether $A = B$ (this is where the unusual book title comes from).

One of the main reasons why these problems turned out to be so hard is that not only we did not have efficient *programs* for checking such equalities, but researchers were not even sure that this problem was algorithmically solvable at all. Indeed, we want to check formulas of the type $\forall n(A(n) = B(n))$, with a universal quantifier running over all natural numbers. For computable functions $A(n)$ and $B(n)$, checking such formulas is equivalent to solving a halting problem – an algorithmically undecidable task.

However, in spite of the fact that checking such equalities is impossible for *general* computable functions, Knuth believed that it may be possible for a *specific* class of computable functions – functions which contain sums of combinatorial expressions. This hope and optimism came naturally from decades of frustration. This phrase sounds somewhat paradoxical, so let me clarify it. Few (maybe none) computer science theoreticians are fond of handling combinatorial sums (which appear again and again in complexity problems). These sums are challenging, tough, often require a lot of effort – sometimes as much (or even more) effort than the computer science part of the problem – but the reward and recognition for solving a combinatorial problem is usually much-much smaller than for solving the computer science problem. Why? Because the solution is usually very technical, rather short, not very exciting, usually – a clumsy complicated combination of simple ideas and transformations rather than than a stroke of genius. This combinatorial part is very frustrating, but in this very frustration lies hope: since all these sums seem to be computed by using the same ideas, maybe these ideas are indeed sufficient, and a computer can take care of finding the appropriate combination of these ideas? This hope turned out to be true.

The history of the resulting algorithms starts with Carl Friedrich Gauss (1755-1837), a genius who was recognized, in his time, as the King of Mathematicians. Probably everyone heard the story how a ten-year old Gauss, when asked to find the sum $1+2+\ldots+100$, came up with a general formula $1+2+\ldots+n = n(n+1)/2$. The trick that he used to design this formula – grouping together the first and the last term, the second and the second from last, etc. – can actually help to add up an arbitrary arithmetic progression. It is less known that Gauss kept an interest in similar sums for his entire life. He found many interesting formulas for *specific* sums $\sum\limits_k t(k)$ – and he was also looking for *general* formulas of this type.

The two most well-known cases in which we have an explicit formula for the sum are *arithmetic* and *geometric* progressions. A geometric progression $t(k) = \text{cosnt} \cdot q^k$ can be described by the condition that the ratio of every two consecutive terms is a constant: $t(k+1)/t(k) = q$. For an arithmetic progression $t(k) = a + b \cdot k$, a similar ratio is a rational function:

$$\frac{t(k+1)}{t(k)} = \frac{a + b \cdot (k+1)}{a + b \cdot k}.$$

It is therefore natural to define a *generalization* of geometric and arithmetic progressions by considering series in which the ratio $t(k+1)/t(k)$ is a rational function:

$$\frac{t(k+1)}{t(k)} = \frac{P(k)}{Q(k)}$$

for some polynomials $P(k)$ and $Q(k)$. Gauss called such series *hypergeometric*. Example of hypergeometric series include Taylor series for most elementary functions: e.g., $\exp(x) =$

$\sum_k t(k)$, where

$$t(k) = \frac{x^k}{k!},$$

and the ratio $t(k+1)/t(k) = x/(k+1)$ is a rational function of k. The usefulness of this notion for combinatorial sums comes from the fact that binomial coefficients are hypergeometric series: if $t(k) = \binom{n}{k}$, then $t(k+1)/t(k) = (n-k)/(k+1)$ is a rational function of k.

How can we describe the sum of such terms? In the two above cases in which we have an explicit formula for the "indefinite" sum $s(n) = t(1) + \ldots + t(n)$, the sum is either itself hypergeometric – for an arithmetic progression, or a hypergeometric term $z(n)$ plus a constant c ($s(n) = z(n) + c$) – for a geometric progression. It turns out that such a term $z(n)$ can be found in many other cases. The question of when the sum $s(n) = \sum_{i=0}^{n} t(i)$ can be represented as $z(n) + c$ for some hypergeometric term $z(n)$ was analyzed, in the late 1970s, by R. W. Gosper Jr. who was working on the first symbolic computation program MACSYMA. Gosper developed an algorithm which, given a hypergeometric term $t(n)$, checks whether the sum is hypergeometric, and if it is, produces the corresponding hypergeometric expression for the sum. This algorithm was actually implemented in *Macsyma*. From the *mathematical* viewpoint, whenever we can find such an expression $z(n)$, it is great. But do we gain anything from the *computational* viewpoint?

At first glance, if we consider *algebraic complexity* (number of arithmetic operations needed to compute a term), we do not gain anything at all. Indeed, what we do gain computationally when we know that a term $t(n)$ is hypergeometric, i.e., that the ratio $t(k+1)/t(k)$ is a rational function of k? Computing the value of a rational function requires a finite number C of arithmetic operations, so we need C operations to compute $t(1)$ from $t(0)$, C operations to compute $t(2)$ from $t(1)$, etc., until we compute $t(n)$. So, totally, we need $O(n)$ steps to compute $t(n)$. Similarly, if we know that the sum $s(n)$ is hypergeometric, we can compute it in $O(n)$ steps.

On the other hand, even if the sum $s(n)$ is *not* hypergeometric, we can still compute it in $O(n)$ steps: indeed, we start with $t(0) = s(0)$, and then, for $i = 1, \ldots, n$, compute $t(i+1) = t(i) \cdot (P(i)/Q(i))$ (constant number of operations) and $s(i+1) = s(i) + t(i+1)$ (one more operation). The multiplicative constants in these two $O(n)$'s may be different, but it is not even clear which one is faster: on one hand, we need an extra addition to compute $s(n)$ as a sum, but, on the other hand, $s(n)$ can be hypergeometric with a more complex polynomials P and Q.

In short, from the *algebraic* complexity viewpoint, there may be no advantage in using this mathematically interesting result. However, from the viewpoint of *actual* computation time, there is a serious advantage, because actual computation can use *special functions* in addition to arithmetic operations. Specifically, if we factorize both polynomials $P(k)$ and $Q(k)$, we get the expression

$$\frac{t(k+1)}{t(k)} = \frac{(k+a_1) \cdot \ldots \cdot (k+a_p)}{(k+b_1) \cdot \ldots \cdot (k+b_q)} \cdot x$$

for some (generally complex) numbers a_i, b_j, and x. Thus, by using a gamma function $\Gamma(x)$, for which $\Gamma(x+1) = x \cdot \Gamma(x)$ and $\Gamma(n+1) = n!$ for integer n, we get

$$t(n) = c \cdot \frac{\Gamma(n+a_1) \cdot \ldots \cdot \Gamma(n+a_p)}{\Gamma(n+b_1) \cdot \ldots \cdot \Gamma(n+b_q)} \cdot x^n, \tag{2}$$

where

$$c = t(0) \cdot \frac{\Gamma(b_1) \cdot \ldots \cdot \Gamma(b_q)}{\Gamma(a_1) \cdot \ldots \cdot \Gamma(a_p)}.$$

We can also use the formula $x^n = \exp(n \cdot \ln(x))$ to compute x^n. So, if we count the application of exp and Γ functions as one step each in our description of a generalized algebraic complexity, then we can conclude that we can compute $t(n)$ in finitely many computational steps. In this case, finding an explicit hypergeometric expression for the sum $\sum t(i)$ is extremely computationally useful: it brings the complexity down from $O(n)$ to $O(1)$ operations.

One thing deserves mentioning here: from the viewpoint of *practical* computations, the formula (2) is not very useful, because it requires us to divide two extremely large numbers (of orders $n!$) to get a number of reasonable size. This division (similarly to a more well known case of subtracting two large almost equal numbers) is not a good way to computing, because for the result to come out correct, we need to compute both the numerator and the denominator with a very high accuracy. A much better formula can be obtained if we use, instead of the actual (fast growing) gamma functions, auxiliary functions $\Gamma'(n, a_i) = \Gamma(n + a_i)/n! = \Gamma(n + a_i)/\Gamma(n + 1)$. In terms of these auxiliary functions, the expression (2) takes the easier-to-compute form

$$t(n) = c \cdot \frac{\Gamma'(n,a_1) \cdot \ldots \cdot \Gamma'(n,a_p)}{\Gamma'(n,b_1) \cdot \ldots \cdot \Gamma'(n,b_q)} \cdot n!^{q-p} \cdot \exp(n \cdot \ln(x)). \tag{3}$$

This expression was actually proposed (in slightly different notations) by Gauss himself. Interestingly, *mathematicians* (and even the authors of the book under review) consider the introduction of the $\Gamma(n+1)$ terms in Gauss's formulas as an unnecessary complication – because it does make formulas (slightly) longer, but, as we have just seen, it make perfect *computational* sense.

In most combinatorial sums used in the analysis of algorithm complexity, the sum goes over all the values of the index for which the corresponding binomial coefficients are different from 0. We can therefore describe the desired sums as *definite* sums $\sum\limits_k t(k)$, where k runs over all the integers. If the corresponding indefinite sum $s(n) = \sum\limits_{k=0}^{n} t(k)$ is hypergeometric, then we can also compute the corresponding definite sum. But what if the indefinite sum is *not* hypergeometric (and often it is not)? In many of such cases, the definite sum is either hypergeometric itself, or is a sum of several hypergeometric term. Chapter 8 describes an algorithm which, given an integer r, checks whether a definite sum $\sum\limits_k t(k)$ can be represented as a sum of r (or fewer) hypergeometric terms (and explicitly produces these terms if they exist). This algorithm, first proposed in Petkovšek's 1991 Ph.D. dissertation, capitalizes on the techniques previously proposed by the two other authors of the book.

In some cases, the desired sum $\sum t(k)$ is not hypergeometric, so we need a larger class of functions to describe such sums. The main reason why such a class is needed is that the class of hypergeometric functions is not algebraically closed: while the *product* of two hypergeometric functions is (trivially) also hypergeometric, the *sum* is often not: one can easily check that even the sum $t(n) = a_1^n + a_2^n$ of two geometric progressions is not hypergeometric.

In this particular non-hypergeometric sum, the first term $t_1(n) = a^n$ is a solution to the following first-order difference equation with constant coefficients: $(S - a_1)t_1 = 0$, where $(St)(n) \stackrel{\text{def}}{=} t(n + 1)$; similarly, the second term $t_2(n) = a_2^n$ is a solution to $(S - a_2)t_2 = 0$, hence the sum is a solution to the following *second-order* difference equation with constant coefficients: $(S - a_1)(S - a_2)t = 0$, i.e., $t(n + 2) = (a_1 + a_2) \cdot t(n + 1) - a_1 \cdot a_2 \cdot t(n)$.

In general, each hypergeometric term $t_i(k)$ is, by definition, a solution to the first-order difference equation with *rational* coefficients $t_i(k+1) = (P_i(k)/Q_i(k)) \cdot t_i(k)$. Therefore, the sum $t(k) = t_1(k) + \ldots + t_r(k)$ of such terms satisfies a higher-order difference equation with rational coefficients:

$$t(k + r) = \frac{P_1'(k)}{Q_1'(k)} \cdot t(k + r - 1) + \ldots + \frac{P_r'(k)}{Q_r'(k)} \cdot t(k). \tag{4}$$

Functions satisfying such equations (4) are called *holonomic.* The set of holonomic functions is closed under addition, multiplication, indefinite addition, etc. A natural algorithmic definition of such a function starts with functions which are solutions of equations (4), and then allows addition, multiplication, etc.

There exist algorithms for checking whether two given holonomic functions (of one or several variables) coincide or not.

The programs in MAPLE and MATHEMATICA implementing all above algorithms can be downloaded from the book's website http://www.cis.upenn.edu/ wilf/AeqB.html. Moreover, the entire book can be downloaded for free (the visionary publisher agreed to it!). For those who want to study the book seriously: there are several minor typos – which are all mentioned on the book's website.

It is worth mentioning, that for several variables, the problem of checking whether $A = B$ quickly becomes computationally intensive. This is not surprising since one can show that satisfiability can be described in these terms. Namely, if we have a propositional formula with v Boolean variables $z_1, \ldots, z_v$, then we can define n auxiliary holonomic functions $a_i(n_i)$, $1 \le i \le v$, as follows: $a_i(0) = 1$ and $a_i(n_i + 1) = -a_i(n_i)$ (hence, $a_i(n_i) = (-1)^{n_i}$). Constants are holonomic functions, and sums and products of holonomic functions are holonomic, so for every i, the function

$$s_i(n_i) = \frac{1}{2} \cdot (a_i(n_i) + 1)$$

is also holonomic. We assign, to every Boolean variable z_i, the corresponding function $f^{[z_i]} = s_i(n_i)$, to its negation, the function $f^{[\bar{z}_i]} = 1 - f^{[z_i]}$; to a disjunction, the function $f^{[a \vee b]} = f^{[a]} + f^{[b]} - f^{[a]} \cdot f^{[b]}$, and to conjunction, $f^{[F_1 \& F_2]} = f^{[F_1]} \cdot f^{[F_2]}$. As a result, to the original Boolean formula F, we assign a holonomic function $f^{[F]}(n_1, \ldots, n_v)$ such that this function f is identically 0 if and only is not satisfiable. Thus, checking equalities for thus defined holonomic functions is *NP-hard.*

Moreover, we can add $\sum\limits_{n_i=0}^{1}$ instead of $\exists z_i$ and $\prod\limits_{n_i=0}^{1}$ instead of $\forall z_i$ and thus get a holonomic function equivalent to an arbitrary *quantified satisfiability* problem – hence, we can prove that the corresponding problem is *PSPACE-hard.*

In this review, I have only mentioned the computational complexity of the problems and algorithms, and I said nothing about a beautiful mathematical theory behind them, a theory for which Herbert Wilf and Doron Zeilberger received a prestigious Steele prize from the American Mathematical Society. I just want to mention that a large part of the algorithms for computing a definite sum $f(n) = \sum\limits_k F(n,k)$, where $F(n,k)$ is a hypergeometric function of both variables, is based on the possibility to find a "dual" hypergeometric expression $G(n,k)$ for which either

$$F(n+1,k) - F(n,k) = G(n,k+1) - G(n,k) \tag{5}$$

– then $f(n) = \text{const}$ – or a more sophisticated equality hold for some polynomials $a_j(n)$

$$\sum_{j=0}^{J} a_j(n) \cdot F(n+j,k) = G(n,k+1) - G(n,k) \tag{6}$$

– in this case $f(n)$ is a holonomic function satisfying the difference equation

$$\sum_{j=0}^{J} a_j(n) f(n+j) = 0.$$

There is an algorithm which, given F, always returns G and a_i. The left-hand side of (6) can be unraveled via unraveling several consequent first-order difference operators – like a telescoping fishing rod. This is how Zeilberger called his algorithm – *creative telescoping.*

The equality (5) is as simple and as beautiful as its continuous analog

$$\frac{\partial F}{\partial x} = \frac{\partial G}{\partial y}$$

which is a part of a definition of a complex analytical function $F(x,y) + \mathrm{i}G(x,y)$, and the consequences of the simple equality (5) are as deep, unexpected, beautiful (and as computationally helpful) as those of complex analysis.

The book also contains an interesting history, from the pioneer 1945 algorithm by Sister Mary Celine Fasenmyer (who happened to die, at the age of 90, in the exact same year in which this book was published), to algorithms and results of Gosper, Zeilberger, Wilf, and Petkovšek.

Although the book is very technical, it is written in a very popular and understandable way. It starts with the basics, it gently guides the reader through the programs, through the formulas, and through the numerous examples. Because of the book's unusual level of understandability, when it was published in 1996, it was selected as a Main Selection of the Library of Science – an honor rarely reserved for technical books.

I just love this book, and I hope y'all will too.