Binomial Coefficient Computation: Recursion or Iteration?

Yannis Manolopoulos
Data Engineering Laboratory
Department of Informatics
Aristotle University
Thessaloniki, 54006 Greece
<manolopo@delab.csd.auth.gr>

Abstract

Binomial coefficient computation, i.e. the calculation of the number of combinations of n objects taken k at a time, C(n,k), can be performed either by using recursion or by iteration. Here, we elaborate on a previous report [6], which presented recursive methods on binomial coefficient calculation and propose alternative efficient iterative methods for this purpose.

1. Introduction

Students become familiar with the topic of recursion versus iteration in several courses as in computer programming, algorithms, and data structures. When comparing the advantages and disadvantages of recursion against iteration, we mention the simplicity in writing and understanding a recursive program (especially if it based on a recursive mathematical function). On the other hand, we emphasize the fact that recursive programs are not as efficient as iterative ones due to the extra time cost for function calls and extra space cost for stack bookkeeping. Thus, the epilogue in such a lecture is that if we need to simply have a correct program, then we can use recursion; however, if time is a critical issue, then we have to use iteration.

Motivation of the present report is the article of Timothy Rolfe in the 2001 June issue of *inroads* [6], where he examined a number of tips for efficient calculation of binomial coefficients by using recursion. The author presented a recurrence relation based on the Pascal triangle, then he derived a more efficient recurrence relation by using some algebra and binomial coefficient properties, and finally he suggested using the greatest common divisor to further improve the latter method.

2. Iterative Approaches

When lecturing on recursion and recursive programs, at the same time we spend quite some time to compare with equivalent iterative programs. Moreover, we comment on all these methods and try to show how we can improve by elaborating gradually each method. Thus, students capitalize from this *engineering* approach. In this respect, classical are the books by Bentley [1,2,3]. For specific examples that build on such a progressive approach, see Section 5.3 on Minimum Spanning Trees in the book by Moret and Shapiro [5]), or Column 7 on the Maximum Subsequence Problem in [2].

In the classroom, we can show several examples for such a purpose. For instance, the calculation of powers, factorials, greatest common divisors, and Fibonacci numbers are classic simple cases. In contrast, well-known algorithms exist for both recursive and iterative versions of binary search, quicksort, and mergesort among others.

In the sequel, first we present a recursive and then an iterative Pascal variation for the calculation of factorials. We will use these functions for the calculation of binomial coefficients. From the theoretical point of view, both variants are equivalent since they perform $O(n)$ operations to calculate n! However, from the practical point of view, the previous comment holds: function calls have a cost that is more significant in comparison to the cost spent for the control structures and other assignment operations. Also, have in mind that in this case, the performed operations are multiplications, but we use the general term *operations* instead of *multiplications*, since the number of divisions will be equally important in the subsequent efforts. In other words, multiplications *and* divisions are our *barometer* metrics in the course of estimating the algorithmic complexity [4].

```
FUNCTION Factorial1(n: INTEGER): INTEGER;
BEGIN
  IF n=0
    THEN Factorial1:=1
    ELSE Factorial1:=n*Factorial1(n-1)
END;

FUNCTION Factorial2(n: INTEGER): INTEGER;
VAR i,product: INTEGER;
BEGIN
  I:=n; product:=1;
  WHILE i<>0 DO
  BEGIN
    product:=i*product;
    i:=i-1
  END;
  Factorial2:=product
END;
```

Reviewed Papers

The following code fragment depicts a Pascal implementation of the first recursive solution for the binomial coefficient calculation, which appears in [6]. Apparently, this implementation is very inefficient since its complexity is *O*(C(n,k)). More specifically, it performs 2C(n,k)-1 multiplications to calculate C(n,k).

```
FUNCTION Comb1(n,k: INTEGER): INTEGER;
BEGIN
  IF (k=0) OR (k=n)
    THEN Comb1:=1
    ELSE Comb1:=Comb1(n-1,k-1)+Comb1(n-1,k)
END;
```

From the above starting point, we will move gradually to smarter solutions. First effort is to get rid of the recursion. We can achieve this by calling any of the previous two Factorial functions. Although, the following straightforward fragment has a *O*(n) computation complexity, it has to be noted that there is a hidden constant equal to 2.

```
FUNCTION Comb2(n,k:INTEGER): INTEGER;
VAR t1,t2,t3: INTEGER;
BEGIN
  t1:=Factorial2(n);
  t2:=Factorial2(k);
  t3:=Factorial2(n-k);
  Comb2:=t1/(t2*t3)
END;
```

The previous iterative function has made an impressive improvement in efficiency in comparison to the Comb1 function. However, we can achieve further improvement by avoiding performing a certain number of multiplications on the numerator for a term that simplifies by a same term of the denominator. The following fragment `Comb3` is a Pascal implementation in place of the second recursive formula that appeared in [6]. Here, we remark that a division operation comes into play. The computation complexity of `Comb3` is *O*(k), i.e. irrelevant of n, with a hidden constant equal to 2. We derive this by simply remarking that after simplifications, both the numerator and denominator contain k terms.

```
FUNCTION Comb3(n,k: INTEGER): INTEGER;
BEGIN
  IF (k=0)
    THEN Comb3:=1
    ELSE Comb3:=Comb3(n-1,k-1)*n/k
END;
```

As mentioned in [6], the latter method can be improved by using the binomial coefficient property that C(n,k)=C(n,n-k).

Next, we present a more efficient iterative version based on this remark. Thus, the following `Comb4` fragment first calculates the maximum between k and n-k, in order to save a number of operations. Although Comb4 fragment is longer, apparently its computation complexity is *O*(min(k,n-k)), also with a hidden constant equal to 2.

```
FUNCTION Comb4(n,k:INTEGER): INTEGER;
VAR t1,t2: INTEGER;
BEGIN
  IF k<n-k THEN DO
    BEGIN
      t1:=1;
      FOR i:=n DOWNTO n-k+1 DO t1:=t1*i;
      t2:=Factorial2(k); Comb4:=t1/t2
    END
  ELSE
    BEGIN
      t1:=1;
      FOR i:=n DOWNTO k+1 DO t1:=t1*i;
      t2:=Factorial2(n-k); Comb4:=t1/t2
    END
END;
```

As noted in [6], the `Comb3` method has the disadvantage that intermediate results are larger in magnitude than the final number. Our`Comb4` method has the same disadvantage. For large values of n and k, this may lead to overflows due to the inadequacy of the used data types. For this reason, a third recursive solution was provided in [6], where divisions are performed before multiplications to prevent such a situation. This is achieved by first calculating the greatest common divisor (gcd) of n and k, a task that can be solved by the well-known Euclidean algorithm of *O*(logn) complexity [4]. Assuming that d:=Gcd(n,k) and q:=k/d, the following fragment `Comb5` is easy to follow.

```
FUNCTION Comb5(n,k: INTEGER): INTEGER;
VAR d,q: INTEGER;
BEGIN
  IF (k=0)
    THEN Comb5:=1
    ELSE Comb5:=(Comb5(n-1,k-1)/q)*n/d
END;
```

The above fragment `Comb5` has the same computation complexity as the previous `Comb3` function. The advantage of `Comb5` over `Comb3` is that the former is more robust for various n and k values, whereas the disadvantage is the cost of the gcd calculation. Next, we give a final iterative function `Comb6`, which is more efficient from the theoretical and the practical point of view.

```
FUNCTION Comb6(n,k:INTEGER): INTEGER;
VAR t: INTEGER;
BEGIN
  t:=1
  IF k<n-k
    THEN FOR i:=n DOWNTO n-k+1 DO
      t:=t*i/(n-i+1)
    ELSE FOR i:=n DOWNTO k+1 DO
      t:=t*i/(n-i+1);
  Comb6:=t
END;
```

Thus, with `Comb6` we have reached our final word. This method has three advantages:

1. It is iterative, thus avoiding time overhead for function calls and space overhead for stacks,
2. It has optimal complexity, that is *O*(min(k,n-k)),
3. It is robust, as it performs multiplications and division alternatively, thus avoiding data type overflows.

3. Conclusions

Motivation for this paper was the article by Rolfe [6] on binomial coefficient calculation by using recurrence relations. Here, we make a step further and argue on alternative iterative methods. We presented a number of Pascal fragments that evolve from the less efficient to variants that are more efficient. Such an approach shows that programming is a science (i.e. methodology) *and* an art.

References

[1] Bentley J.L.: *Writing Efficient Programs*, Prentice Hall, 1982.

[2] Bentley J.L.: *Programming Pearls*, Addison Wesley, 1986.

[3] Bentley J.L.: *More Programming Pearls - Confessions of a Coder*, Addison Wesley, 1988.

[4] Brassard G. and Bratley P.: *Fundamentals of Algorithmics*, Prentice Hall, 1996.

[5] Moret B.M.E. and Shapiro H.D.: *Algorithms from P to NP, Volume I: Design and Efficiency*, Benjamin/Cummings, 1991.

[6] Rolfe T.: Binomial Coefficient Recursion: the Good, and the Bad and Ugly, *ACM SIGCSE Bulletin Inroads*, Vol.33, No.2, pp.35-36, 2001.