

Recursive Algorithms in Computer Science Courses: Fibonacci Numbers and Binomial Coefficients

Ivan Stojmenovic

Abstract—We observe that the computational inefficiency of branched recursive functions was not appropriately covered in almost all textbooks for computer science courses in the first three years of the curriculum. Fibonacci numbers and binomial coefficients were frequently used as examples of branched recursive functions. However, their exponential time complexity was rarely claimed and never completely proved in the textbooks. Alternative linear time iterative solutions were rarely mentioned. We give very simple proofs that these recursive functions have exponential time complexity. The proofs are appropriate for coverage in the first computer science course.

Index Terms—Binomial coefficients, computer science, Fibonacci numbers, recursion.

I. INTRODUCTION

RECURSION is an important problem solving and programming technique and there is no doubt that it should be covered in the first year introductory computer science courses, in the second year data structure course, and in the third year design and analysis of algorithms course. While the advantages of using recursion are well taught and discussed in textbooks, we discovered that its potential pitfalls are often neglected and never fully discussed in literature.

For the purpose of our discussion, we shall divide recursive functions into linear and branched ones. Linear recursive functions make only one recursive call to itself. Note that a function's making only one recursive call to itself is not at all the same as having the recursive call made one place in the function, since this place might be inside a loop. It is also possible to have two places that issue a recursive call (such as both the *then* and *else* clauses of an *if* statement) where only one call can actually occur. The recursion tree of a linear recursive function has a very simple form of a chain, where each vertex has only one child. This child corresponds to the single recursive call that occurs. Such a simple tree is easy to comprehend, such as in the well known factorial function. By reading the recursion tree from bottom to top, we immediately obtain the iterative program for the recursive one. Thus the transformation from linear recursion to iteration is easy, and will likely save both space and time. However, these savings are only in the constant of linear time complexity for both recursive and iterative solutions, and can be easily disregarded.

Branched recursive functions make more than one call to themselves. All examples of branched recursive functions seen

in the textbooks make two calls to themselves, and may be referred to as examples of double recursion. The recursion tree of a double recursive function is a binary one. There are applications of double recursion which produce optimal (up to constant) solutions. Typical examples are the preorder, inorder and postorder traversals of a binary tree, which are simple recursive functions used in any data structure course. Their time complexity is proportional to the size of a traversed tree. Another famous example is the towers of Hanoi problem, which has simple but exponential time recursive solution, which is optimal due to the nature of the problem. It is well known that any recursive function can be rewritten to produce an equivalent iterative implementation, using stack data structure and a scheme that is straightforward but is difficult to grasp and apply. The corresponding iterative solution, produced by the scheme, has equal time complexity. An iterative solution, often more elegant than the recursive one, can be produced for a given problem directly, without the scheme. For example, [1] gave a surprisingly simple iterative algorithm for the towers of Hanoi problem.

Most textbooks use, however, Fibonacci numbers or binomial coefficients as the first examples of branched recursive functions, in the first year computer science courses. These examples often appear as apparently recommended programs in some textbooks, as observed in [2]. Although most textbooks mention the inefficiency of these solutions, the degree of inefficiency is rarely discussed and never fully proved. Only a few textbooks include alternative linear time (for Fibonacci numbers) or quadratic time (for binomial coefficients) algorithms.

In the next section we shall review some recent textbooks for first, second and third year computer science courses regarding branched recursive functions and their iterative counterparts. In Section III we give a simple proof that the recursive algorithm for computing Fibonacci numbers has exponential time complexity. A similar proof for recursive binomial coefficient computation is given in Section IV. All proofs are appropriate for the students in first computer science courses. They were taught successfully by the author at the University of Ottawa for five consecutive years. We conclude that recursion should be covered completely, including drawbacks in addition to advantages. It may be covered as a problem solving technique in the first computer science course. In our judgement, recursion as a programming technique should be taught after iterative algorithms are sufficiently exercised (in the later part of the first course or at the beginning of the second one). The success of actual programming exercises may depend on the accumulated programming skills and prior mastering of stacks as data structures.

Manuscript received January 11, 1999; revised November 22, 1999.

The authors are with the Department of Computer Science, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: ivan@site.uottawa.ca).

Publisher Item Identifier S 0018-9359(00)03986-8.

Fibonacci numbers are defined as follows:

$$f(n) = 1 \quad \text{for } n = 1 \quad \text{and} \\ n = 2, \quad \text{and} \quad f(n) = f(n-1) + f(n-2) \\ \text{for } n > 2.$$

Thus, the sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 377, 610, 987, 1597, 2584, ...

The well-known recursive algorithms for computing $f(n)$ can be written as follows:

```
f(n):
  if n = 1 or n = 2 then the answer is 1;
  if n > 2 then {compute FN1 = f(n-1);
                compute FN2 = f(n-2);
                the answer is FN1 + FN2}.
```

Binomial coefficient $C(n, k)$ (n chooses k) can be recursively computed as follows (assume that k and n are nonnegative integers):

```
C(n, k):
  if k > n then the answer is 0;
  if k = 0 or k = n then the answer is 1;
  if k > 0 and k < n then {compute Cn1k = C(n-1, k);
                          compute Cn1k1 = C(n-1, k-1);
                          the answer is Cn1k + Cn1k1}.
```

Binomial coefficients $C(n, k)$ form Pascal's triangle as follows (each entry is the sum of the 2 entries above it, one directly above it, and the other to the left):

$n \backslash k$	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

II. REVIEW OF TEXTBOOKS

We reviewed three recent textbooks for introductory first year computer science courses. The textbook [3] presents the recursive Fibonacci numbers function, without mentioning its inefficiency. Similarly, the textbook [4] describes the recursive computation of " N choose K ," again without discussing its inefficiency. Finally, the textbook [5] observes that "although easy to write, the Fibonacci function is not very efficient, because each recursive step generates two calls to function Fibonacci." No further explanations were offered.

We also reviewed four second year textbooks in data structures. The textbook [6] does not have examples of inefficient recursive calls and does not discuss possible recursion inefficiency. The textbook [7] repeats the statement from [5] and continues by noting that the "recursive Fibonacci function repeats the same calculation several times during the evaluation of the expression. This does not happen in an iterative implementa-

tion of the Fibonacci function." The degree of inefficiency is not elaborated. Maximal element in a list has been computed in [8] by using "double" recursion, and its inefficiency was discussed. However, exponential time complexity was not mentioned. Finally, [2] offers a good discussion about linear and branched recursion. "A far more wasteful example than factorials is the computation of the Fibonacci numbers. This example is not "divide-and-conquer" but "divide-and-complicate." It turns out that the amount of time used by the recursive function to calculate $f(n)$ grows exponentially with n [2]. However, [2] stops short on proving the statement. The same book also observes that the recursion tree for calculating Fibonacci numbers is not a chain, but contains a great many vertices signifying duplicate tasks. When a recursive program is run, it sets up a stack to use while traversing the tree, but if the results stored in the stack are discarded rather than kept in some other data structure for future use, then a great deal of duplication of work may occur, as in the recursive calculation of Fibonacci numbers [2]. In such cases, it is preferable to substitute another data structure for the stack, one that allows references to locations other than the top. For the Fibonacci numbers only two additional temporary variables were needed to hold the information required for calculating the current number [2].

The third year textbook on the design and analysis of algorithms were expected to provide full answer about branched recursive functions. The textbook [9] derives $f(n)$ using a sophisticated argument involving calculus and quadratic functions. For the recursive solution, [9] notes that "this is a bad way to compute the Fibonacci numbers since many numbers will be computed many times. How much work does the recursive algorithm do? This is a surprisingly *difficult* question, considering how simple the algorithm is" [9]. The exponential time complexity was not mentioned. In the next section, a very *simple* proof that the amount of work is exponential shall be provided. The following linear time algorithm for computing $f(n)$ was given in [9] (the algorithm computes $f(n) = fn$ by using only four variables):

```
fn2 := 1; fn1 := 1;
for i := 3 to n do {fn := fn2 + fn1;
                  fn2 := fn1;
                  fn1 := fn}.
```

The textbook [10] explains how to compute $f(n)$ in $\log(n)$ time by using a 2×2 matrix. It offers an exercise to readers to show themselves that the recursive computation of $f(n)$ takes exponential time, and to find an iterative linear time solution. A few textbooks, not cited below, did not discuss recursive algorithms at all. Finally, textbook [11] presents a linear time algorithm for computing $f(n)$ by using arrays, as follows"

```
f[1] := 1; f[2] := 1;
for i := 3 to n do f[i] := f[i-1] + f[i-2].
```

In addition to being linear, the algorithm calculates $f(i)$ for all values of $i = 1, 2, \dots, n$, while the corresponding recursive algorithms returns only $f(n)$. The same book also makes

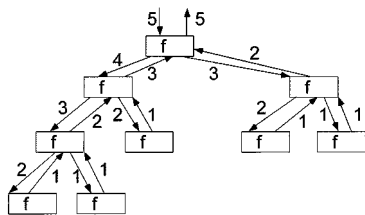


Fig. 1. Execution tree for $f(5)$.

a step toward providing a full answer by observing that the ‘exact number of calls on the procedure Fibonacci is equal to the number of calls to compute $f(n - 1)$ plus the number of calls to compute $f(n - 2)$, that is exactly $f(n)$ ’. The statement is not formally proven and is somewhat imprecise, as discussed below. Moreover, the size of $f(n)$ is not discussed.

III. FIBONACCI NUMBERS

Fig. 1 shows the execution tree for computing $f(5) = 5$ by using a recursive algorithm. The execution tree is a binary tree. The order of execution (recursive calls) follows the outer boundary of the execution tree and makes a cycle with input $n = 5$ and output $f(n) = 5$, as indicated by arrows (more precisely, the order of the execution is the preorder traversal of the execution tree). Repeated computations of $f(3), f(2)$ and $f(1)$ are easily observed. The main observation is that the recursive calls stop when the corresponding value for Fibonacci number (that is, $f(1)$ or $f(2)$) becomes equal to 1. These 1’s are at the leaves of the execution tree. How many 1’s are at the bottom of the execution tree, that is at its leaves? The answer is surprisingly simple: $f(n)$, because recursive calls only stop when the value drops to 1, and thus $f(n)$ is actually computed as $f(n) = 1 + 1 + \dots + 1$. Therefore the recursive solution must “decompose” $f(n)$ into $f(n)$ 1’s and sum them up!

With this observation, it is easy to see that the recursive solution needs $f(n) - 1$ times to apply the summation (operation “+”) in order to compute $f(n)$. Again, this is because $f(n)$ is computed as the sum of 1’s only! The number of recursive calls is also $f(n) - 1$, since each such recursive call can be paired with a “+” operation (here, the leaves are not counted as producing recursive calls; otherwise the answer is $2f(n) - 1$). Note that the proof can be easily presented even without the execution tree; that is, it is appropriate for the first computer science course that typically does not cover the notion of trees.

An alternate proof may be given as an exercise for mathematical induction. The base case is true (i.e. $f(1)$ and $f(2)$). In the inductive step, one can observe that, since the computation of $f(n)$ recursively compute $f(n - 1)$ and $f(n - 2)$, the number of recursive calls is $(f(n - 1) - 1) + (f(n - 2)) + 1 = f(n) - 1$.

The exponential time complexity can only be argued if $f(n)$ is proven to be a number that is exponential in n . The proofs in literature refer to the exact formula for $f(n)$

$$f(n) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

$$\cong \frac{1}{\sqrt{5}} 1.618^n$$

which can be proved by mathematical induction (and can be used as a basis for yet another iterative algorithm). The proof can be understood (with difficulties, according to personal experience) by first year students. For a large n

$$\frac{F(n)}{F(n - 1)} \cong 1.618 \text{ is referred to as the golden ratio.}$$

Here a much simpler proof that $f(n)$ is exponential in n is given. Observe that $f(n) = f(n - 1) + f(n - 2) = f(n - 2) + f(n - 3) + f(n - 2) = 2f(n - 2) + f(n - 3) > 2f(n - 2)$. Thus $f(n) > 2f(n - 2) > 2(2f(n - 2 - 2)) = 4f(n - 4) > 4(2f(n - 4 - 2)) = 8f(n - 6) > \dots > 2^k f(n - 2k)$ for $n - 2k > 0$. If n is even stop at $n - 2k = 2$; otherwise stop at $n - 2k = 1$. In both cases $k = (n - 1)/2$ where the division is integer division (that is, the ratio is truncated). Since $f(1) = f(2) = 1, f(n) > 2^{(n-1)/2}$ is obtained. Therefore $f(n)$ is exponential in n .

IV. BINOMIAL COEFFICIENTS

We did not find the iterative algorithm for computing binomial coefficients in the textbook reviewed, but believe that it is well known. The rows $0 \dots m$ of Pascal’s triangle can be found iteratively if $C[n, k]$ is treated as a two-dimensional array (matrix) as follows:

```

for i := 0 to m do {C[i, 0] := 1; C[i, i] := 1};
for n := 2 to m do
  for k := 1 to n - 1 do
    C[n, k] := C[n - 1, k] + C[n - 1, k - 1];

```

The second for loop is iterated approximately m times, each time with approximately n iterations of the internal nested for loop. Since $n \leq m$, the time complexity is proportional to m^2 . More precisely, the solution requires about $m^2/2$ additions and is therefore quadratic. Note that each $C(i, j)$ is computed only once in the algorithm, and that all of them are computed, while the recursive algorithm returns only one requested value $C(n, k)$.

Consider now the time complexity of the recursive algorithm for computing binomial coefficients. Fig. 2 illustrates the execution tree for computing $C(4, 2) = 6$. One can again observe that recursive calls stop when the value of the binomial coefficient drops to 1, and that $C(n, k)$ has been decomposed as $C(n, k) = 1 + 1 + \dots + 1$. Therefore there are $C(n, k)$ leaves at the bottom of the execution tree, and the recursive algorithm makes $C(n, k) - 1$ additions to compute $C(n, k)$. To prove that the algorithm has exponential time complexity, one should show that $C(n, k)$ is exponential in size. This is not always true, but is true when k is approximately $n/2$. This can be proven in an elegant way as follows. It is well known that $C(n, k)$ can be expressed as follows (the expression can be used as a basis for another iterative algorithm):

$$C(n, k) = \frac{n(n - 1)(n - 2) \dots (n - k + 1)}{k!}$$

$$= \frac{n}{k} \frac{n - 1}{k - 1} \frac{n - 2}{k - 2} \dots \frac{n - k + 1}{1}$$

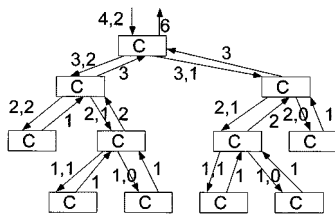


Fig. 2. Execution tree for computing $C(4, 2)$.

We shall prove that $n/k < (n-1/k-1) < (n-2/k-2) < \dots < (n-k+1/1)$. One can show that $a/b < (a-1/b-1)$ whenever $b < a$ (this is equivalent to $a(b-1) < b(a-1)$, i.e. $ab-a < ba-b$, that is $b < a$). This suffices for our claim. Thus $C(n, k) > (n/k)^k$. For $k = n/2$ this implies $C(n, k) > 2^{n/2}$, which is exponential in n .

V. CONCLUSION

Recursion is an efficient technique for definitions and algorithms that make only one recursive call, but can be extremely inefficient if it makes two or more recursive calls. Thus the recursive approach is frequently more useful as a *conceptual tool* rather than as an efficient *computational tool*. The proofs presented in this paper were successfully taught (over a five-year period) to first year students at the University of Ottawa. It is suggested that recursion as a problem solving and defining tool be covered in the second part of the first computer science course. However, recursive programming should be postponed for the end of the course (or perhaps better at the beginning of the second computer science course), after iterative programs are well mastered and stack operation well understood.

Our method for proving the exponential time complexity of some branched recursive functions can be generalized for other similar functions. If t is the maximum value for a function $g(n)$ that does not invoke a recursive call (in our examples $t = 1$) and addition is applied on the results of recursive calls to compute $g(n)$, then the number of additions is at least $g(n)/t$. Thus such recursive functions have complexity $O(g(n))$ since t is a constant. Addition may be replaced here with some other operations to obtain similar arguments.

The discussion here refers only to the standard recursive solutions to Fibonacci number and binomial coefficients problems. It may not apply if a linear recursive function for any of them is described. However, the author believes that the problem is fundamental, meaning that there is no general scheme for converting inefficient branched recursive functions into efficient linear recursive ones. This will remain a problem for future study.

REFERENCES

- [1] F. B. Chedid and T. Mogi, "A simple iterative algorithm for the towers of Hanoi problem," *IEEE Trans. Educ.*, vol. 39, pp. 274-275, May 1996.
- [2] R. L. Kruse, C. L. Tondo, and B. P. Leung, *Data Structures and Program Design in C*. Englewood Cliffs, NJ: Prentice-Hall, 1997.
- [3] A. B. Tucker, A. P. Bernat, W. J. Bradley, R. B. Cupper, and G. W. Scragg, *Fundamentals of Computing I*. New York: McGraw-Hill, 1994.
- [4] T. L. Naps and D. W. Nance, *Introduction to Computer Science: Programming, Problem Solving, and Data Structures*. St. Paul, MN: West, 1995.
- [5] E. B. Koffman, *Pascal*. Reading, MA: Addison-Wesley, 1995.
- [6] T. A. Standish, *Data Structures, Algorithms, and Software Principles*. Reading, MA: Addison-Wesley, 1994.
- [7] E. B. Koffman and B. R. Maxim, *Software Design and Data Structures in Turbo Pascal*. Reading, MA: Addison-Wesley, 1994.
- [8] A. B. Tucker, W. J. Bradley, R. D. Cupper, and R. G. Epstein, *Fundamentals of Computing II*. New York: McGraw-Hill, 1994.
- [9] G. J. E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms*. Rockville, MD: Computer Science, 1992.
- [10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*: McGraw Hill, 1990.
- [11] R. Sedgwick, *Algorithms*. Reading, MA: Addison-Wesley, 1988.

Ivan Stojmenovic received the B.S. and M.S. degrees from the University of Novi Sad, Yugoslavia, in 1979 and 1983, respectively, and the Ph.D. degree in mathematics from the University of Zagreb, Croatia, in 1985.

In 1980, he joined the Institute of Mathematics, University of Novi Sad. In the fall of 1988, he joined the faculty in the Computer Science Department at the University of Ottawa, Ottawa, ON, Canada, where currently he holds the position of a Full Professor. He has published three books and over 140 different papers in journals and conferences. His research interests are wireless networks, parallel computing, multiple-valued logic, evolutionary computing, neural networks, combinatorial algorithms, computational geometry and graph theory. He is currently a Managing Editor of *Multiple-Valued Logic* and an editor of *Parallel Processing Letters*, *IASTED International Journal of Parallel and Distributed Systems*, *Discrete Mathematics and Theoretical Computer Science*, *Parallel Algorithms and Applications*, and *Tangenta*.