Journal of Information & Communication Technology
Vol. 1, No. 1, (Spring 2007) 34-41

A logarithmic time hybrid solution of Fibonacci numbers using dynamic programming technique

H.Mehta*
D.Abhyankar*
S. Tanwani*
A. K. Ramani*
School of Computer Science, Devi Ahilya University, Indore, INDIA

ABSTRACT

Leonardo of Pisa (1175-1250) in 1202 introduced Fibonacci numbers. Gabriel lame used the Fibonacci sequence in the analysis of the efficiency of the Euclidean algorithm (the first algorithm of the world). Lucas who popularized the Towers of Hanoi puzzle derived many properties of this sequence. Lucas was first to call these numbers the Fibonacci sequence. Despite a long history, very limited literature is available on the efficient solution of the Fibonacci sequence. In this paper, we propose an algorithm that efficiently computes Fibonacci numbers. The proposed algorithm is an hybrid version of two existing algorithms: one based on memroization Mehta (2006) and the other based recursive squaring method Knuth and designed to deliver best space-time tradeoff. The implementation of our algorithm and the experimental results prove that the suggested algorithm outperforms the other known algorithms.

Inspec Classification: C4140; C4240P; C6110F

Keywords : Dynamic Programming, memorization, Fibonacci number, recursive solution, Recursive squaring, complexity

1) INTRODUCTION

An algorithm is a sequence of computational steps that transforms the input into output. Algorithms and their analysis is an activity that amuses the intellectuals and at the same time there are huge payoffs in terms of time and money. The practical application of algorithms are ubiquitous and varies from the Human Genome Project, which identifies approximately 100,000 genes in human DNA, determines three billion chemical base pairs that makeup human DNA to the recently emerging E-commerce applications. Optimal tools and sophisticated algorithms are needed for searching, data analysis and efficient information retrieval Introduction to algorithms. The next section gives a brief overview of the existing algorithms in the area of Fibonacci computation.

* The material presented by the authors does not necessarily portray the viewpoint of the editors and the management of the Institute of Business and Technology (BIZTEK) and as well as the School of Computer Science, Devi Ahilya University, Indore, INDIA

*H.Mehta: hmehta.scs@dauniv.ac.in,
*D.Abhyankar: dabhyankar.scs@dauniv.ac.in,
*S. Tanwani: stanwani.scs@dauniv.ac.in
*A. K. Ramani: head.scs@dauniv.ac.in

Fibonacci numbers were first given by Leonardo of Pisa (1175-1250) in 1202. Fibonacci numbers can be given by following recurrence relation.

F0 = 0
F1 = 1
Fn = Fn-1 + Fn-2 for n > 1

Abraham DeMoivre solved this recurrence relation using generating functions in 1718. Kepler also used these numbers in his studies. The mathematical writings of Fibonacci can be traced way back to 1202 by Fibonacci's book Liber Abbaci describing his mathematical experiences arising from the contacts he made on his Mediterranean travels was completed in Pisa. After this Practica Geometriae (1220) (The Practice of Geometry), this substantial, well-written book contains several chapters of mainly Euclidean theorems which represent "a considerable advance over the Geometry of Boethius and Gerbert (Pope Sylvester II)". Later a book named Liber Quadratorum (1225), was written by Fibonacci after his Liber Abbaci. In it, Fibonacci shows his mathematical prowess in solving Diophantine problems. At the same time, Flos (The Flower) (1925), in this short work Fibonacci describes inter alia two of the Diophantine problems he worked on at the court of the Emperor Frederick. Next, we describe the Fibonacci solution using recursive squaring method.

Recursive squaring method is based on following equation:

$$\begin{bmatrix} Fn+1 & Fn \\ Fn & Fn-1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n}$$

This requires exponentiation using binary method which will require 2 log n matrix multiplications in the worst case. These matrices are 2X2 and their multiplication will need 8 multiplications and 4 additions each. These 8 multiplications and 4 additions can be reduced in 4 multiplications and 2 additions because Fn is being computed twice in above mentioned equation.

2) LITERATURE SURVEY

After an initial survey on the algorithms related to Fibonacci, It was difficult to find recent research papers in this area.   Fibonacci computation algorithms can be broadly classified into three categories: The first algorithm describes the Fibonacci function with a Fibonacci recurrence relation and is given below:
F (0) = 0
F (1) = 1
For n> 1 F (n) = F (n-1)+ F (n-2)
A recursive function on the basis of above mentioned recurrence relation takes exponential arithmetic operations. Another algorithm uses a table to store partial results in the recurrence relation computation and therefore results in linear arithmetic operations. The third category is of algorithms, those are able to compute the function in logarithmic time. Two algorithms under this category are introduced in the literature. The first algorithm taking logarithmic time is based on following formula given by Abraham DeMoivre:

Fn = (1/v5) (1n-2n) ---------------------------- (A)
Fn = (1/v5) (1n) Approximately
1 =((1+v (5))/2) and 2 = (((1 -v (5))/2)

The algorithm computes the Fibonacci function in logarithmic arithmetic operations. However, the computation involves extensive floating point arithmetic. The second algorithm in the category of taking logarithmic time is Recursive Squaring MIT.

An algorithm that is free of floating point arithmetic and matrix multiplications and solves the problem in logarithmic time using integer operations ONLY was introduced in Mehta (2006). The mathematical analysis and the empirical results proved that the proposed algorithm Mehta (2006) was superior to the best-known techniques in the literature. The idea of the algorithm is based on dynamic programming technique David B. Wagner. . A 1995, where the partial results are stored in a table. The next section briefly describes the dynamic programming technique.

3) INTRODUCTION TO DYNAMIC PROGRAMMING

In computer science, dynamic programming is a method for reducing the runtime of algorithms exhibiting the properties of overlapping subproblems and optimal substructure. Optimal substructure means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem. For example, the shortest path to a goal from a vertex in an acyclic graph can be found by first computing the shortest path to the goal from all adjacent vertices, and then using this to pick the best overall path. In general, we can solve a problem with optimal substructure using a three-step process:

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The subproblems are, themselves, solved by dividing them into sub-subproblems, and so on, until we reach some simple case that is easy to solve.

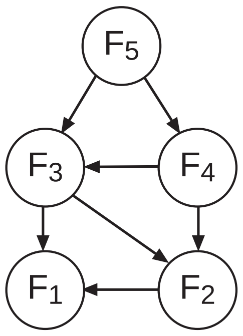


Figure 2: The above is a subproblem graph for the Fibonacci sequence. That it is not a but a indicates overlapping subproblems.

To say that a problem has overlapping subproblems is to say that the same subproblems are used to solve many different larger problems. For example, in the , F3 = F1 + F2 and F4 = F2 + F3 — computing each number involves computing F2. Because both F3 and F4 are needed to compute F5, a naïve approach to computing F5 may end up computing F2 twice or more. This applies whenever overlapping subproblems are present: a naïve approach may waste time recomputing optimal solutions to subproblems it has already solved.

In order to avoid this, we instead save the solutions to problems we have already solved. Then, if we need to solve the same problem later, we can retrieve and reuse our already-computed solution. This approach is called memoization (not memorization, although this term also fits). If we are sure we won't need a particular solution anymore, we can throw it away to save space. In some cases, we can even compute the solutions to subproblems we know that we'll need in advance.

In summary, dynamic programming makes use of:

- Overlapping subproblems
- Optimal substructure
- Memoization

Dynamic programming usually takes one of two approaches:

- Top-down approach: The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again. This is recursion and memoization combined together.

- Bottom-up approach: All subproblems that might be needed are solved in advance and then used to build up solutions to larger problems. This approach is slightly better in stack space and number of function calls, but it is sometimes not intuitive to figure out all the subproblems needed for solving given problem.

Originally, the term dynamic programming only applied to solving certain kinds of operational problems outside the area of , just as did. In this context, it has no particular connection to at all; the name is a coincidence. The term was also used in the by , an American mathematician, to describe the process of solving problems where one needs to find the best decisions one after another.

4) ANALYSIS OF FIBONACCI COMPUTING USING DYNAMIC PROGRAMMING TECHNIQUE [9]

This algorithm is based on the equation no. 7 and 8. The Fibonacci recurrence relation is:

F (0) = 0
F (1) = 1
For n> 1 F (n) = F (n-1)+ F (n-2)
$$F(n) = F(n-1) + F(n-2) \quad \text{----(1)}$$
$$F(n-1) = F(n-2) + F(n-3) \quad \text{----(2)}$$
Using (1) and (2)
$$F(n) = 2*F(n-2) + F(n-3) \quad \text{----(3)}$$
Similarly
$$F(n) = 3*F(n-3) + 2*F(n-4) \quad \text{----(4)}$$
In generalize form
$$F(n) = F(a)*F(n-a+1) + F(a-1)*F(n-a) \quad \text{----(5)}$$
If n is even then, take a=n/2
$$F(n) = F(n/2)*F(n/2+1) + F(n/2-1)*F(n/2) \quad \text{----(6)}$$
By taking F (n/2) common,
$$F(n) = F(n/2)[F(n/2+1) + F(n/2-1)] \quad \text{----(7)}$$
If n is odd then, take a = (n+1)/2
$$F(n) = F((n+1)/2)*F((n+1)/2) + F((n-1)/2)*F((n-1)/2) \quad \text{----(8)}$$

These identities are special cases of identities given in Knuth's Art of Computer programming Volume 1. Knuth.

Equation (7) and (8) are key to the success of proposed algorithm. First let us have a close observation of equation (7). At first, it looks as if it needs 3 recursive calls one each for F(n/2), F(n/2+1) and F(n/2-1). But a close look at equation (7) reveals that a recursive function call to compute F((n/2)+1) is redundant. In equation (7), the computed values of F(n/2-1) and F(n/2) are stored in a table, it need not call a recursive function to compute F(n/2+1) because F(n/2+1) = F(n/2-1) + F(n/2). So, it needs two calls to compute F((n/2)-1) and F(n/2). The key characteristics of the proposed algorithm is that the repeated computations of F(n) can be eliminated by storing the values in a table and using them later, whenever needed. This is because dynamic programming technique is applied here. Through the suggested approach, the growth of the tree is restricted. Similarly in Equation (8), it needs to make two recursive calls F((n+1)/2) and F((n-1)/2) out of which only one will be expanded.

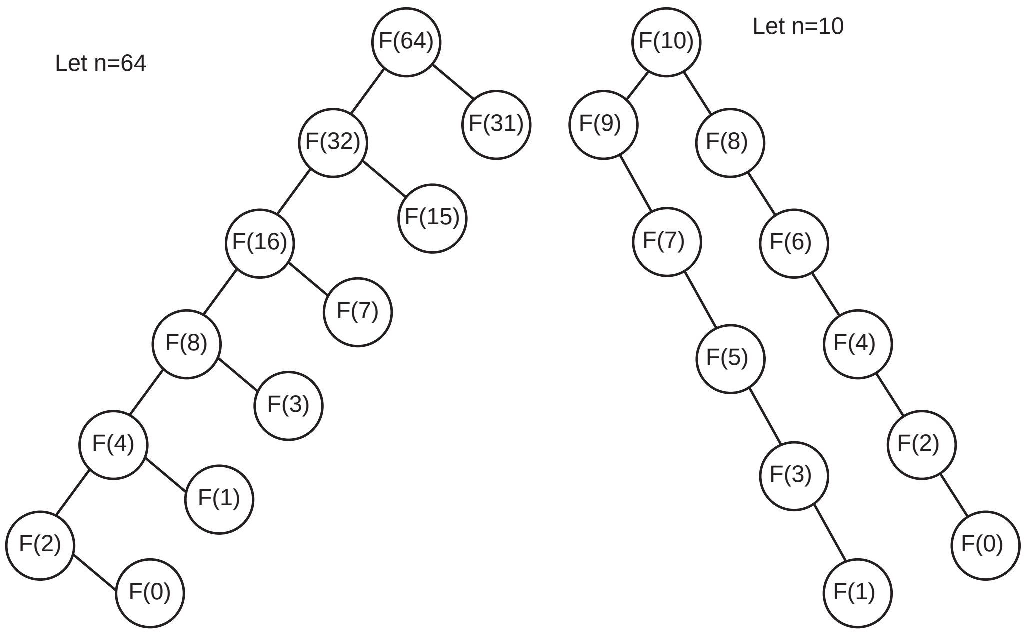
Algorithm: Fib(n as integer values)
Comment: This is a recursive algorithm. Here Result is an array of integer values, used to store the calculated results.

```
              Begin
                  If (n=0) then
                          return 0
                  if (n=1) then
                          return 1
                  if (n=2) then
                          return 1
                  if (Result[n]!=0) then
                          return Result[n]
                  if ((n%2)==0) then
                                  if(Result[n/2]!=0) then
                                          return Result[n/2]
                                  a =  fib(n/2)
                                  if(Result[n/2-1]!=0) then
                                          return Result[n/2-1]
                                  b =  fib((n/2)-1)
                                  c = a+b
                                  Result[n]=a*(b+c)
                  else
                          if (Result[(n+1)/2]!=0) then
                                  return Result[(n+1)/2]
                                  a = fib((n+1)/2)
                                  if(Result[(n-1)/2]!=0) then
                                  return Result[(n-1)/2]
                                  b = fib((n-1)/2)
                                  Result[n]=a*a + b*b
                          end if;
                          return Result[n]
end of fib(n);
```

5) TIME COMPLEXITY AND SPACE REQUIREMENT FOR VARIABLES IN FIBONACCI COMPUTING USING MOMOIZATION

The total number of function calls will be of logarithmic order, as we are storing the numbers in a table. If h is the height of recursion tree, the total number of function calls will be approx. 2h. When the number (n) is a integral power of 2, the h will be log2 n otherwise it will be approx. log2 n. Consider the following recursion tree: Empty sub-trees will not be expanded because they are calculated and stored in Result Table in Right sub-tree.

This algorithm uses memoization so it has cost of an array of size n. Our algorithm needs to maintain an hash table to store 2 log n elements.



1 (a): Proposed method with n=64 Figure 1 (b) : Some Other method with n=10

Comparison of Fibonacci computing using memoization with Recursive Squaring method:

The results obtained with the proposed method are compared with the Recursive squaring method. The following table and graph compare the performance of our proposed method and recursive squaring method in terms of number of operations. Number of multiplication operations is taken as a criterion of comparison. The following graph is depicting the comparison between the input parameter "n" and required multiplication operation to obtain the Fibonacci number. Now as in case if the input parameter is even, it requires less number of multiplications. For odd numbers it requires more multiplication. This is why the graph contains less multiplication in case of even number and more for the odd numbers. The following data in Table 1 is used to create the Graph in figure 2. The results prove that the memoization method Mehta (2006) is better than recursive squaring method in terms of number of operations.

Number	Proposed	Rec. Sqr.	Number	Proposed	Rec. Sqr.	Number	Proposed	Rec. Sqr.	Number	Proposed	Rec. Sqr.
1	0	0	13	9	40	25	14	48	37	15	56
2	1	8	14	8	40	26	13	48	38	14	56
3	3	16	15	9	48	27	12	56	39	14	64
4	2	16	16	8	32	28	11	48	40	13	48
5	5	24	17	11	40	29	13	56	41	17	56
6	4	24	18	10	40	30	12	56	42	16	56
7	6	24	19	11	48	31	12	64	43	15	64
8	5	24	20	10	40	32	11	40	44	14	56
9	8	32	21	12	48	33	16	48	45	16	64
10	7	32	22	11	48	34	15	48	46	15	64
11	8	40	23	11	56	35	14	56	47	14	72
12	7	32	24	10	40	36	13	48			

Table 1: Number of operation with the Input Parameter.

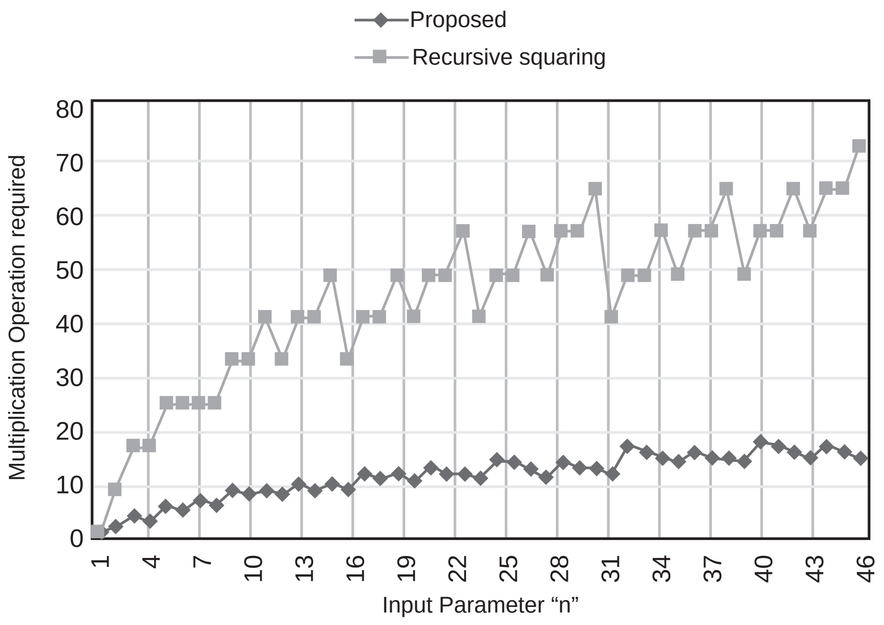


Figure 2: Operation required computing the Fibonacci number.

In case of Fibonacci computation (using memoization) storage cost will increase with increasing values of "n". There are two reasons. First and more obvious reason is that with increasing "n" more results will have to be stored. Second problem is that with increasing "n" results (fib (n)) will get bigger and bigger. And the Recursive squaring method doesn't stores any values, so for larger values of "n" the recursive squaring method is more space efficient.

6) PROPOSED HYBRID METHOD

One major problem with recursive squaring method is that for small values of n, it suffers high overhead. Recursive squaring requires multiplications of 2x2 matrices. A 2x2 matrix multiplication requires 4 multiplications and 2 additions. For example, F(3) requires 4 multiplications and 2 additions. When n is fairly low this overhead is simply too much. Therefore, we propose a hybrid method. Our idea is to use dynamic programming method for small values of n and when n grows sufficiently big we can use recursive squaring method. There is a space speed tradeoff between dynamic programming and recursive squaring method. Recursive squaring method does not require an array which we use in memoization (dynamic programming). As n grows, space requirement of memoization increases.

7) CONCLUSION

This paper proposes a hybrid algorithm for Fibonacci computation. Our hybrid is based on two algorithms. First method, based on memoization Mehta (2006) is best in terms of time. The second one, based on Recursive squaring method is best in terms of space. Hybrid method gives the best of both the methods. For small values of n, where space overhead suffered by dynamic programming is low memoization is used and when n is fairly high recursive squaring method is applied.

REFERENCES

LIBER ABBACI (The Book of Calculation), 1202 (1228).
LIBER QUADRATORUM (The Book of Square Numbers), 1225.
FLOS (The Flower), 1225.
PRACTICA GEOMETRIAE (The Practice of Geometry), 1220.
INTRODUCTION TO ALGORITHMS, Second Edition, PHI, Thomas H. Coreman, Charles E. Lieserso.
MIT Open Courseware.
KNUTH, Art of computer programming Volume 1.
DAVID B. WAGNER. . A 1995 introductory article on dynamic programming.
H. MEHTA, D. ABHYAKAR, S. TANWANI "A logarithmic time recursive solution of Fibonacci numbers using dynamic programming technique" Accepted by Varahmihir Journal of Computer and Information Science(VJCIS), Indian to be published in December 2006.