BALKANJM 01 (2013) 16-27

Contents lists available at BALKANJM

BALKAN JOURNAL OF MATHEMATICS

journal homepage: www.balkanjm.com

A New Method for Fast Computation of Factorials of Numbers

Fikret Cihan[a], Fatih Aydin[b], Adnan Fatih Kocamaz[*c]

[a] *Vocational School of Technical Sciences, Kirklareli University, Kirklareli, Turkey*

[b]*Department of Computer Programming, Kirklareli University, Kirklareli, Turkey*

[c] *Department of Software Engineering, Kirklareli University, Kirklareli, Turkey*

ARTICLE INFO

Article history:
Received 23 December 2012
Accepted 23 January 2013
Available online 6 February 2013

Keywords:
Analyze of algorithm Factorial
Binary tree
Fast factorial computation

ABSTRACT

This study introduces a newly developed algorithm for fast computation of the factorial of big numbers. The algorithm reduces by half the number of multiplications required to compute the factorial of a number. Then, to speed up the multiplication process, the numbers to be multiplied are converted into binary trees. Following the conversion, the products for the left and right branches of the tree are computed synchronically, and the multiplication of the two values yields the result of the factorial.
In computing the factorial of numbers rapidly, 11 non-prime-number based algorithms are used, which are compared to the method developed in this study. Analyses show that the newly developed method, in addition to being simpler and easy to use, computes the factorial of big numbers much faster compared to the other methods.

1. Introduction

The notation $n!$ is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1, \; if\,(n=0) \\ n.(n-1)!, \; if\,(n>0) \end{cases} \qquad (1)$$

Thus, $n! = 1.2.3 \ldots n$.

*Corresponding author: *E-mail*: fatih.kocamaz@kirklareli.edu.tr (A., F., Kocamaz)

A weak upper bound on the factorial function is $n! \leq n^n$, since each of the n terms in the factorial product is at most n [1]. Stirling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^{\pi} e^{a_n} \tag{2}$$

where

$$\frac{1}{12n+1} < a^n < \frac{1}{12n} \tag{3}$$

Stirling's approximation yields a result very close to the real result, though the result of the number whose factorial is to be computed is not absolute. However, the higher the value of n, the closer one gets to the real result. This is illustrated in Equation (4).

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^{\pi}} = 1 \tag{4}$$

The factorial function is used in mathematics, the number theory, the probability theory, exponential, trigonometric and hyperbolic functions, series expansions, permutations and combinations. As such, fast computation of factorial is highly significant.

Most basic occurrence of the factorial operation is the fact that there are $n!$ ways to arrange n distinct objects into a sequence. This fact was known at least as early as the 12th century, to Indian scholars [2]. The notation $n!$ was introduced by Christian Kramp in 1808 [3].

The asymptotically best efficiency is obtained by computing $n!$ from its prime factorization. Prime factorization allows $n!$ to be computed in time $O(n(\log n \log\log n)^2)$, provided that a fast multiplication algorithm is used [4].

The method that computes the factorial as a product of the numbers $1-n$ is known as the naive product [1, 5]. Considering function call overhead, the recursive method is even slower than the naive product. Computing factorials has long been a tedious task, requiring many multiplications and working with numbers that grow exponentially by the number of digits of their values [6]. Such cases greatly increase the time required to compute the factorial of big numbers. This has led researchers to develop faster factorial computation methods, including BoitenSplit, Split, Swing Simple, Swing, Square Difference etc. BoitenSplit algorithm transforms the multiplication of even numbers into shift operations and applies multiplication only to odd numbers [7]. Swing Simple is a simple Swing algorithm. Square Difference is an algorithm based on the differences of squares. A benchmark program developed by Peter Luschny features most of fast factorial computation algorithms [8]. The 11 fast factorial computation algorithms used in this study can be found in that benchmark program.

2. Description and Complexity of the fast factorial computation algorithm

The algorithm introduced in this study uses the following approximation in computing the factorial of numbers. This approximation reduces by half the number of multiplications in computing the factorial. In this approximation, the number whose factorial is to be calculated is factorized. After that, the first term is multiplied by the last term; then the second term is multiplied by the second-from-last term etc., thus multiplying all the terms. Equation (7) illustrates this.

$$n! = 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20\cdots n \tag{5}$$

$$n! = n.2(n-1).3(n-2).4(n-3).5(n-4)\cdots\frac{n}{2}\left(\frac{n}{2}+1\right) \tag{6}$$

$$n! = n.(2n-2).(3n-6).(4n-12).(5n-20)\cdots\left(\frac{n^2+2n}{4}\right) \tag{7}$$

Afterwards, the difference between terms is considered. (8) shows the difference between terms. As can be seen, the difference between terms is 2, which is a constant. (8) shows further that the difference between the last two terms is 2.

$$\underbrace{(2n-2)-n}_{(n-2)}.\underbrace{(3n-6)-(2n-2)}_{(n-4)}.\underbrace{(4n-12)-(3n-6)}_{(n-6)}\cdots\underbrace{\left(\frac{n^2+2n}{4}\right)-\left(\frac{n^2+2n-8}{4}\right)}_{(2)} \tag{8}$$

Thus, the factors required for 20! are as in (9). An analysis of the difference between factors for 20! shows that the first difference is 18. In other words, it is always the number whose factorial is to be calculated minus 2. Then, the difference decreases by two, ultimately reducing down to 2. Thus, the value of the second difference becomes 16, and that of the last difference becomes 2. Crucially, the number of multiplications required for 20! is reduced from 20 to 9.

$$20! = 20.38.54.68.90.98.104.108.110 \tag{9}$$

This rule can be used with odd numbers as well as even numbers, though with a little trick. Assuming n to be odd, its factorial can be considered as $n.(n-1)!$. Thus, in computing $n!$, one can first calculate $(n-1)!$ using the above method, and then multiply the result by the value of n.

The method developed to compute the factorial of even numbers is formulated in (10). The formula is explained next.

2.1 Theorem The formula developed to compute the factorial of even numbers is as follows: when $n>1$ and n is even;

$$n! = [2+4+\cdots+(n-2)+n].[4+6+\cdots+(n-2)+n][(n-2)+n].n \tag{10}$$

Proof If we choose the method of induction to prove the theorem.

Let us assume that;

$$for \quad n=2, \quad 2!=2 \tag{11}$$

$$for \quad n=4, \quad 4!=(2+4).4=6.4=24 \tag{12}$$

$$for \quad n=k, \quad k!=[2+4+\cdots+(k-2)+k].[4+6+\cdots+(k-2)+k].k \tag{13}$$

And show that;

$$for \quad n=k+2, \quad (k+2)!=[2+4+\cdots+k+(k+2)].\cdots.[k+(k+2)].(k+2) \tag{14}$$

Then, we get

$$(k+2)! = (k+2).(k+1).k!$$

$$= (k+2).(k+1).[2+4+\cdots+(k-2)+k].[4+6+\cdots+(k-2)+k].k$$

$$= (k+2).(k+1).\left(\frac{k}{2}\right).\left(\frac{k+2}{2}\right).\left(\frac{k-2}{2}\right).\left(\frac{k+4}{2}\right).\left(\frac{k-4}{2}\right).\cdots.\frac{4}{2}.\left(\frac{k-2}{2}\right).k$$

$$= \left(\frac{k+2}{2}\right).\left(\frac{k+4}{2}\right).\left(\frac{k}{2}\right).\left(\frac{k+6}{2}\right).\left(\frac{k-2}{2}\right).\left(\frac{k+8}{2}\right).\cdots.\frac{4}{2}.\left(\frac{2k+2}{2}\right).(k+2)$$

$$= [2+4+\cdots+k+(k+2)].[4+6+\cdots+k+(k+2)].\cdots.[k+(k+2)].(k+2)$$

Which proves the theorem.

The method developed for fast factorial computation is shown to greatly reduce the number of multiplications required. (15) gives the number of multiplications required to compute the factorial of a number. Here, n is the number whose factorial is to be computed, and $f(n)$ is the number of multiplications required to compute the factorial of n.

$$f(n)=\begin{cases}\frac{n}{2}-1, & \text{if } n \text{ is even}\\ \left|\frac{n}{2}\right|, & \text{if } n \text{ is odd}\end{cases} \tag{15}$$

The number is first factorized and then the factors are multiplied with one another, which yields the result. However, multiplying these numbers in natural methods takes a much longer time than multiplying the factors in the binary tree structure, hence our preference of the binary tree structure. Due to the use of the binary tree structure, the multiplication function is recursive. Figure 1 illustrates this point.

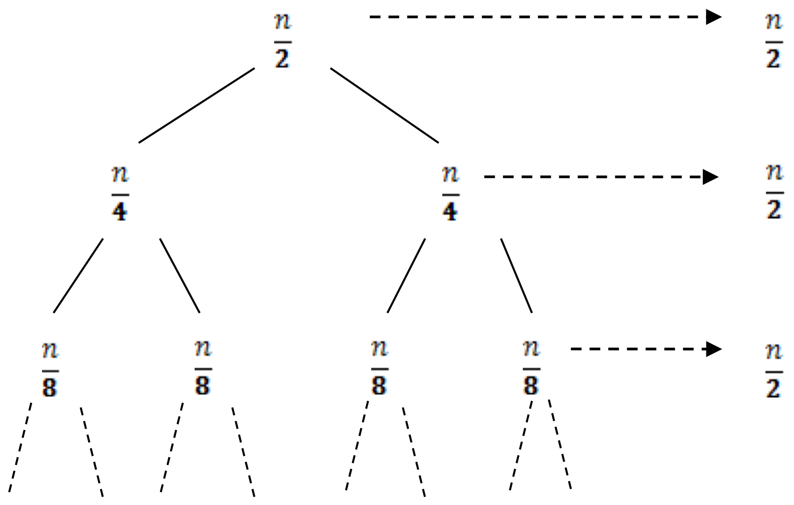

Figure 1. A recursion tree for the recurrence $T(n/2)=2T(n/4)+(n/2)$

The general form for the time complexity of a recursion tree has been shown in (16) equation. Here "a" and "b" rates are arbitrary substantive. $f(n)$ function is a function of n. According to this equation the tree in (n/b) dimension is divided into sub problem till a. The reunification of these sub problems gives the rate of $f(n)$ function. The time complexity of level 1 which is the sub-level of the root node of tree has been shown in (17) equation. This situation goes on until it is reached to the leaves of tree.

$$T(n)=a.T\left(n/b\right)+f(n) \tag{16}$$

$$T\left(n/b\right) = a.T\left(n/b^2\right) + f\left(n/b\right) \tag{17}$$

The basic situation in the leaves is related to $1 \le n \le b$. By this way tree has $\log_b n$ level. The total number of the leaves is the same as it is shown in (18) equation.

$$a^{\log_b n} = n^{\log_b a} \tag{18}$$

According to this, the total of consumed time is the total of the consumed time in every level. Then, the consumed time in i level is $a^i f(n/b^i)$. The level of the tree is $\log_b n$ then i interval is from 0 to $\log_b n - 1$. In this situation, the total consumed time with the consumed time in every level is as the same as the total number of consumed time in the leaves. This is shown in (19) equation.

$$T(n) = \sum_{i=0}^{\log_b(n-1)} a^i.f\left(\frac{n}{b^i}\right) + O\left(n^{\log_b a}\right) \tag{19}$$

Since the used tree structure in developed algorithm is a complete tree and the cost of every leaf is fixed the total cost of the entire leaves is as it is in (20). However, if we want to show the top limit of the cost for the worst situation, we show it as it is in (21) equation.

$$T(n)_{\text{only all leaves}} = O\left(n^{\log_b a}\right) = \omega\left(\log_b n\right) \tag{20}$$

$$T(n)_{\text{only all leaves}} = O\left(n \log_b a\right) \tag{21}$$

The asymptotic growth of $f(n)$ function located in (19) equation is compared with the asymptotic growth of the number of leaves. If the growth of leaves is slower than f function, the time cost is as it is shown in (22) term. If the growth of leaves and f function is the same, the cost of time is as it is shown in (23) term. But if the growth of f function is faster than the growth of leaves, the cost of time is as it is shown in (24) term.

$$T(n) = O\left(n^{\log_b a}\right) \tag{22}$$

$$T(n) = \Theta\left(n^{\log_b a} \log n\right) \tag{23}$$

$$T(n) = \Theta\left(f\left(n\right)\right) \tag{24}$$

Since the time complexity of the developed algorithm is $T(n/2) = 2T(n/4) + (n/2)$, $f(n)$ equates $(n/2)$. The rate is 2 and b rate is 4. The total number of leaves is as it is shown in (18) equation. According to this the number of leaves is $n^{1/2}$. According to this, $f(n)$ function is growing faster than the number of leaves. As a result of this, a time complexity that is shown in (24) equation occurs. That's to say, time complexity is as it is below:

$$T(n) = \Theta(f(n)) = \Theta(n) \tag{25}$$

For the tree shown in Figure 1, the calculations have been done simultaneously from level 1. Thus, time complexity is done as it is in (26) equation. As it is seen in this equation, the time complexity of algorithm has decreased.

$$T\left(\tfrac{n}{4}\right) = 2.T\left(\tfrac{n}{8}\right) + f\left(\tfrac{n}{4}\right) \tag{26}$$

$$T(n) = \Theta(n/4) \tag{27}$$

Consequently, the time cost of algorithm is as it is shown (27) equation. This situation shows that algorithm has increased the multiplier number performance and other methods have provided additional contribution to this situation.

3. Algorithm Details

Algorithm has been developed by using .NET 4.0 library and C# programming language in MS Visual Studio 2010. So as to calculate the factorial of bigger numbers, "System.Numerics.BigInteger" [9] data type has been used. The details of algorithm have been shown as Pseudo code below.

In Figure 2, the starting functions of algorithms were given. In this function, if n value takes a value under 0, the practice throws an exception. If the n value is higher than 0 and lowers than 7, the factorial of the number between these intervals is returned. The reason for making such kind of an operation is that for n value, one branch of the tree needs $n/4$ element. For each number lower than 7, this value will be 1. For such a situation, because a binary tree hasn't been prepared, the factorial of the number until 7 is kept in a line and returned.

```
Function: Factorial
Input: Integer n
Output: BigInteger
Algorithm:

1:     if n < 0
2:            then   throw Exception("n >= 0 required");

3:     else if n < 7
4:     then   fact: array of BigInteger ← {1, 1, 2, 6, 24, 120, 720},
                  size 7, the first index is 0

5:            return fact[n];

6:     return InitializeFactorial(n);
```

Figure 2. Main function

In Figure 3, to calculate the factorial of numbers higher than 6, a function named “Initialize Factorial” is shown. In this function, there are made operations according to whether the number whose factorial is to be calculated is uneven or even number. For, the multiplying number changes depending on if the number is uneven or even. The change of multiplying number (15) is shown in the expression. Besides these procedures, the factors of the number whose factorial is to be found are thrown into a set. The numbers put in the set are sent to the function called “Synchronized Binary Tree” to be multiplied. The result obtained from this function is then multiplied with $2^{n/2}$ for; its factors are two times of each one.

```
Function: InitializeFactorial
Input: Integer n
Output: BigInteger
Algorithm:

1:     eCount ← 0
2:     if n is odd
3:            then   eCount ← 1

4:     loop ← n / 2
5:     f: create array of BigInteger, size (loop + eCount), the first index is 0
6:     f[0] ← loop
7:     if (eCount = 1)
8:            then   f[loop] ← n

9:     i ← 1
10:    s ← loop
11:    r ← 1

12:    for inc ← loop - 1 downto 0
13:           do     f[i] ← s ← s + inc

14:                  while f[i] is even
15:                  do     loop ← loop + 1
16:                         f[i] ← f[i] / 2;
17:                  i ← i + 1

18:    return SynchronizedBinaryTree(f, length of f) * (2 ^ loop)
```

Figure 3. Initialize Factorial function

In Figure 4, the factors of the number whose factorial is to be found are turned into binary tree structure. For the calculation times of the left and right branches of the tree to be same, the left and right branches are calculated simultaneously. Thus, the calculation speed is raised.

```
Function: SynchronizedBinaryTree
Input: array of BigInteger: f and len: Integer
Output: BigInteger
Algorithm:

1:    parallel
2:          do    right ← RecursiveBinaryTree(f, (len - 1) / 2 + 1, len - 1)

3:    left ← RecursiveBinaryTree(f, 0, (len – 1) / 2)

4:    return left * right
```

Figure 4. Synchronized Binary Tree function

In Figure 5, the factors of the number whose factorial is to be found start to be multiplied with each other. On such a tree, the leaves have not been designed in the way to send the value of each number. On the leaves, at least two numbers are multiplied with each other and a result is returned. Hence, the depth of the tree doesn't always have a changeable structure. Besides, the tree is enabled to be a full tree. Such a structure runs with a better performance comparing with another structure.

```
Function: RecursiveBinaryTree
Input: array of BigInteger: f and n: Integer and m: Integer
Output: BigInteger
Algorithm:

1:    if m = n + 1
2:          then   return f[n] * f[m]
3:    if m = n + 2
4:          then   return f[n] * f[n + 1] * f[m]

5:    k ← (n + m) / 2

6:    return RecursiveBinaryTree(f, n, k) * RecursiveBinaryTree(f, k + 1, m)
```

Figure 5. Recursive Binary Tree function

4. Results and discussion

The tests of algorithm have been realized in benchmark program called "Silver Factorial" that is developed by Peter Luschny. The algorithms chosen for testing are: SquaredDiff, ProductRecursive, Boiten Split, SwingDouble, SwingRational, SwingSimple, Hyper, SwingRationalDbl, Split, SquaredDiffProd and Swing. These algorithms were implemented in on the Windows 7 Ultimate operating system, with a Intel® Core™ i7 CPU Q 740 @ 1.73 GHz and 4 GB of RAM. According to the tests done, it has been seen that newly developed algorithms are faster than the other algorithms. The results of tests done are seen in Table 1. When the results are investigated, it is seen that the best result is given by the newly developed algorithms. Secondly, the fastest algorithm is Swing. Then is SquaredDiffProd algorithm. The developed algorithm has produced the same fast result as it is in Split algorithm when calculating the factorial of 5000 number. However, it has gone beyond in the other algorithms that have bigger worthies after this value.

The algorithms that are used in the tests are not the algorithms based on prime numbers. According to the tests done, the reason behind the fact why these algorithms have been chosen is the fact that new algorithm can do fast factorial calculation based on normal methods. Furthermore, the developed algorithm has an easy and simple applicable structure.

Table 1. The comparative results of calculations of fast factorial algorithms (in milliseconds)

n / **Algorithms**	1000	2500	5000	10000	25000	50000	100000	**Average**
Our	1	6	24	86	743	3285	14643	2684
Swing	0	5	26	119	900	4054	16576	3097
SquaredDiffProd	1	5	28	114	884	3827	15323	2883
Split	0	5	24	107	903	4128	18442	3372
SwingRationalDbl	1	7	33	128	1194	4905	19777	3720
Hyper	1	10	29	121	1070	4891	21685	3972
SwingSimple	1	10	40	155	1240	5106	21681	4033
SwingRational	1	8	35	134	1003	5106	22706	4141
SwingDouble	1	7	34	127	843	4502	19761	3610
BoitenSplit	1	8	37	146	1194	5172	25189	4535
ProductRecursive	1	7	33	133	1165	5113	22726	4168
SquaredDiff	1	11	43	170	1314	5675	30995	5458

5. Conclusion

As a result, the developed algorithm factorial, at worst situation, decreases the multiplication number, which is necessary for the number to be calculated, to the half. By this way, the calculations at high speed can be done. The tests have been done according to 11 algorithms and newly developed algorithm. 11 algorithms used for comparison are not the fast factorial calculation algorithms based on prime numbers. According to the tests done, newly developed algorithm factorial calculates the results of the numbers that will be calculated in a fastest way. If the complexity of algorithm is shown via Theta notation, it will be as $\Theta\left(n/4\right)$.

References

[1] Cormen, T. H., Leiserson, C.E., Rivest, R. L., Stein, C., “Introductions to algorithms”, 2nd Ed., The MIT Press, 2001, Printed and bound in the United States of America.ISBN 0-262-03293-7 (hc.: alk. Paper, MIT Press).ISBN 0-07-013151-1 (McGraw-Hill).
[2] N. L. Biggs, (1979) “The roots of combinatorics, Historia Math. 6”, pp. 109−136.
[3] Higgins, Peter (2008), “Number Story: From Counting to Cryptography”, New York: Copernicus, p.p 12, ISBN 978-1-84800-000-1 says Krempe though.
[4] Peter Borwein. "On the Complexity of Calculating Factorials". Journal of Algorithms 6,
[5] Sedgewick, R., (1992) “Algorithms in C++”. Addison-Wesley.
[6] Ugur, A.;Thompson, H., “The p-sized partitioning algorithm for fast computation of factorials of numbers”, The Journal of Supercomputing, 38(1)/October, 2006, pp. 73-82.
[7] BoitenA. (1992) “Factorisation of the factorial-An example of inverting the flow of computation”. Periodica Polytechnica Ser El Eng 35(2):pp. 77–99.
[8] Peter Luschny, Fast Factorial Functions, http://www.luschny.de/math/factorial/FastFactorial Functions.htm, [Accessed: 21.07.2011].
[9] Albahari, J, Albahari, B., “C# 4.0 In a Nutshell”, 4th Ed., O’Reilly Media, 2010, ISBN: 978-0-596-80095-6.