J Supercomput (2006) 38:73–82
DOI 10.1007/s11227-006-7285-5

The *p*-sized partitioning algorithm for fast computation of factorials of numbers

Ahmet Ugur · Henry Thompson

Abstract Computing products of large numbers has always been a challenging task in the field of computing. One such example would be the factorial function. Several methods have been implemented to compute this function including naive product, recursive product, Boiten split, and prime factorization, and linear difference. The method presented here is unique in the sense that it exploits finite order differences to reduce the number of multiplications necessary to compute the factorial. The differences generated are regrouped into a new sequence of numbers, which have at most half as many elements of the original sequence. When the terms of this new sequence are multiplied together, the factorial value is obtained. The cardinality of the new sequence can further be reduced by partitioning. The sequence is computed by using several difference tables that assist in establishing the pattern that determines the sequence. An analysis of the algorithm is presented. The analysis shows that the execution time can be reduced significantly by the algorithm presented.

Keywords Fast factorial computation · Partitioning

1 Introduction

Factorials naturally arise in many problems in mathematics and computer science from the series expansion of basic mathematical functions such as exponential, and trigonometric functions to counting problems like permutation and combination calculations. 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. One standard method of computing a factorial, say $n!$, is the naive product method that multiplies all numbers from 1 to n using an iterative loop [1, 2]. This requires n or $n - 1$ multiplications depending upon implementation. As a result, for larger values of n, computing $n!$ takes a considerable amount of time on a digital computer. Another standard method of computing a factorial is

A. Ugur (✉) · H. Thompson
Department of Computer Science, Central Michigan University, Mt. Pleasant, MI 48859 USA
e-mail: ahmet.ugur@cmich.edu

the recursive product method. For $n!$ this requires n recursive calls and $n-1$ multiplications [1, 2]. Considering function call overhead, the recursive method is even slower than the naive product.

Researchers have developed more complex algorithms for fast factorial computation primarily by manipulating binary representation sometimes with recursive splitting, prime factorization and other advanced techniques [3, 4]. Boiten split algorithm transforms the multiplication of even numbers into shift operations and apply multiplication only to odd numbers [5]. Linear difference algorithm partitions the numbers into four groups only and calculate intermediate results using linear differences (the initial differences is set in advance), and multiply the intermediate results to obtain the final product [3]. Algorithms that utilize prime factorization are quite complex and also use division in addition to multiplication [3, 4].

We have looked at the factorial problem from a different perspective. Mainly, we have focused on how to reduce the number of multiplications to compute $n!$ since multiplication is an expensive operations for larger numbers. In machine language level, addition can be considered as a constant time operation from the algorithm analysis point of view. However, multiplication is not constant time operation like addition [6, 7]. To multiply two numbers represented in two n-bit words, the operation requires a maximum of $2n$ shifts and m additions, where m is the number of nonzero bits of the multiplicand, $0 \leq m \leq n$. Thus, it is important to reduce the number of multiplications for designing a faster algorithm [8]. The algorithm we present reduces the number of multiplications to a much smaller number than n multiplications necessary for the naive product version. The algorithm does so by using partitioning similar to the linear difference algorithm, but in more general way (i.e., it does not use fixed numbers for initial differences, rather calculates them, and does not use a fixed partition size). The idea of partitioning has been used in many algorithms to effectively solve the problem of interest [1, 8].

The algorithm presented below partitions the sequence of numbers to be multiplied together. The relationship of these partition elements is observed and exploited by constructing a finite difference table. Then the factorial is computed using the difference table by replacing at least half of the required multiplications by additions. This leads to a faster computation of factorials. The algorithm is implemented and tested. The algorithm is found to be significantly faster than the standard factorial algorithm [9]. An analysis of the algorithm including optimum number of partitions is presented. The complexity of the optimal algorithm is found to be $O(n^{2/3})$ in terms of number of multiplications necessary.

2 The p-sized partitioning algorithm

2.1 Overview

We will explain the algorithm through an example. Suppose we are calculating 10!. For the naive or recursive product algorithm, this requires at least 9 multiplications to compute. Before we do any multiplication, let us examine the numbers we use to get our answer.

$$1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$$

Now let us see what happens when we partition numbers by pairs of two. Suppose that we multiply each pair together, we get a new sequence, g_n. When the elements of g_n are multiplied together, we obtain the result, 10!. Here, we have to mention that our algorithm

Table 1 The complete difference table after pairwise partitioning of the number sequences to be multiplied

$1 * 2$	$3 * 4$	$5 * 6$	$7 * 8$	$9 * 10$	The two partitioning of the original sequence h_n
2	12	30	56	90	Elements of g_n 0th order difference)
10	18	26	34		1st order difference of g_n
8	8	8			2nd order difference of g_n

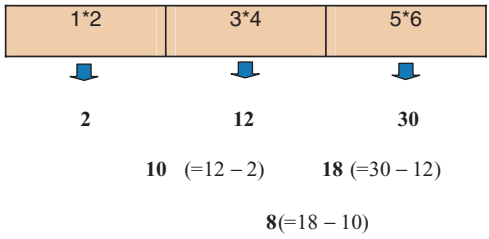

Fig. 1 Initial difference table for pairwise partitioning to compute 10!

does not use the pairwise multiplication to obtain the elements of g_n rather, it uses additions, the details of which will be explained below.

Table 1 shows the first-order and second-order differences after partitioning and multiplying. Note that the second-order differences are equal and constant. Any further differences will all be equal to zero. This implies that there must exist a polynomial that describes the sequence of multiplied terms on the second row [10]. The polynomial is $g_n = 2n(2n - 1)$, for $n \geq 1$, and its second order difference is a constant. The polynomial g_n represents the 0th order difference. Using the polynomial g_n directly is equivalent to the standard factorial algorithm and is not the optimal way of calculating 10! since a polynomial of degree two requires three multiplications and one addition to compute each term of the new sequence.

However, there is an alternative way to use additions to produce terms of a polynomial sequence using first and second order differences. An initial difference table needs to be constructed and utilized to compute elements of g_n successively. For the above example, we know the second-order difference of g_n is always 8. If we start with the initial value and add 8 each time, we would be generating the sequence of numbers that describe the first order difference of g_n whose first term is the initial value (e.g., $10 + 8 = 18$, $18 + 8 = 26$, etc.). Similarly, if we start with the initial value and add the first-order differences together to create the sequence g_n whose first term is the initial value (e.g., $2 + 10 = 12$, $12 + 18 = 30$, etc.). All that is necessary now is to multiply the elements of g_n (i.e., $10! = 2 * 12 * 30 * 56 * 90$). Figure 1 shows the initial difference table for this example. Note that at least 3 pairs of numbers need to be multiplied in order to generate the first element of the second order difference. Figure 2 shows how the elements of g_n is obtained from the initial table.

How much work has been saved? For the example above, in total, we have used 8 additions and $4 + 3 = 7$ multiplications to compute 10!, which is faster than the simple naive product

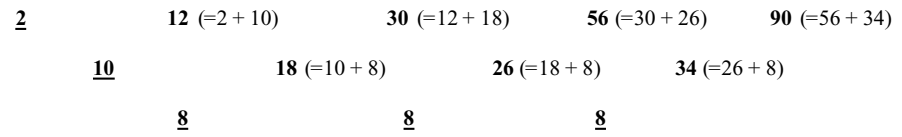

Fig. 2 Elements of the pairwise-multiplication sequence g_n obtained from the initial difference table. Elements of the initial difference table are underlined

method that requires 9 multiplications since each multiplication may require a number of additions.

The above example illustrates the pairwise partitioning fast factorial algorithm. This algorithm can be generalized to an algorithm that uses partititions of size p, where $1 \leq p \leq n$. The detailed algorithm is described below.

2.2 Algorithm details

The algorithm first creates the first elements of each set of differences of different degree from 0 to p since any further differences will all be equal to zero [10]. Then it uses this set to compute the first order differences whose values will yield the main intermediate result when multiplied together. If n is not divisible by p, then the remaining n mod p numbers are multiplied with this intermediate result to complete factorial computation. Figure 3 shows the details of *initializeDifferenceTable* procedure that will generate the initial sequence for the algorithm. Figure 4 shows a pseudo code description of the p-size partitioning fast factorial algorithm.

```
        procedure initializeDifferenceTable(p: integer, X: array of integers, size p + 1,
        the first index is 0)
        { computes the initial difference table using p-sized partitions, 1 ≤ p ≤ n }

        begin
            products: array of integers, size p + 1, the first index is 0
            q, r, s, t: integers

            { initialization }
1:          q = p+1
2:          r = q*p
3:          s = 1

            { Section 1:  compute initial p+1 products of size p }
4:          for (i := 0 ; i < q ; i := i + 1)
            begin
5:              t = 1
6:              for (j := 0 ; j < p ; j := j + 1)
                begin
7:                  t = t * (s + j)
                end

8:              products[i] = t
9               s = s + p
            end

            { Section 2:  compute first elements of each of p differences }
10          X[0] = products[0]
11          for (i := 0 ; i < p ; i := i + 1)
            begin
12              q = q – 1
13              for (j := 0 ; j < q ; j := j + 1)
                begin
14                  products[j] = products[j +1] – products[j]
                end

15              X[i + 1] = products[0]
            end
        end
```

Fig. 3 The *initializeDifferenceTable* procedure that will generate the initial sequence for the p-sized partitioning fast factorial algorithm

```
function factorial_p(n: integer, p: integer)
{computes n! using p-sized partitions, 1 ≤ p ≤ n}
        begin
            X: array of integers, size p + 1, the first index is 0
            iterates, fact: integers

            {Section 1: set the initial elements of X which stores the difference table}
1:          initializeDifferenceTable(p, X)

            {Section 2: obtain elements of difference table successively and multiply
              elements of the sequence g_n}
2:          iterates := n – p
3:          fact := X[0]
4:          while ( iterates ≥ p )
            begin
                {create the first order difference}
5:              for (i := 0 ; i < p ; i = i + 1)
6:                  X[ i ] := X[ i ] + X[ i + 1 ]
7               fact := fact * X[0]
8:              iterates = iterates – p
            end

            {Section 3: multiply remaining numbers to complete the factorial}
9:          for (i := 0 ; i < iterates ; i = i + 1)
            begin
10:             fact := fact * n
11:             n = n – 1
            end

12:         return fact; {final factorial value}
        end
```

Fig. 4 The p-sized partitioning fast factorial algorithm. The algorithm uses $\lfloor \frac{n}{p} \rfloor$ partitions

In Figure 3, lines 1 through 3 represent initialization, lines 4 through 9 represent Section 1 which is responsible for generating the initial $p + 1$ products of size p, and lines 10 through 15 represent Section 2, where each row of differences of order 0 through p are calculated. After each row of difference values is calculated, the first element of each difference sequence is stored in an array of size $p + 1$ (called X in Figure 3). The fast factorial algorithm then uses this array to successively compute elements of the difference table.

In Figure 4, line 1 represents Section 1, which is a call to the procedure *initialize Difference Table* to obtain the initial values of differences of order 0 to p. Lines 2 through 8 represent Section 2 which generates the 0^{th} order difference sequence using the initial values set by section 1. This section also multiplies the 0^{th} order differences to compute the factorial, which may be complete. Lines 9 through 11 represent Section 3 which will complete the factorial computation. If n is not divisible by p, then the remaining n mod p numbers are multiplied with the intermediate result of Section 2, otherwise this Section will be skipped.

The complexity of the algorithm is dominated by operations performed in Section 2 which will be analyzed next.

3 The complexity of the p-sized partitioning algorithm

The complexity of the algorithm will be analyzed in terms of the number of multiplications necessary to compute factorial as a function of both n, whose factorial to be computed, and p,

the partition size. Our working hypothesis is that in a digital computer, addition is a constant operation, while multiplication is not as far as running time of hardware level instructions concerns [6, 7]. Each multiplication requires m additions, where m is the number of nonzero bits of the multiplicand, $0 \leq m \leq k$ where k is the maximum number of bits necessary to represent n.

Let n be number whose factorial is to be computed. The naive product factorial algorithm in general requires n successive multiplication with a running time $T(n) = n$, and therefore is $O(n)$ (i.e., its running time is proportional to the value of n). The total number of multiplications required by the p-sized partitioning fast factorial algorithm presented the above section is given by the following equation:

$$T(n, p) = \left\lfloor \frac{n}{p} \right\rfloor + p^2 + n \bmod p. \tag{1}$$

This is because the initialization procedure (Section 1, Figure 3) requires p^2 multiplications due to nested for loops, and the computation of elements of the 0^{th} order differences (Section 2, Figure 4) requires $\lfloor \frac{n}{p} \rfloor$ multiplications since the step size in the while loop is p, and completing the factorial (Section 3, Figure 4) requires n mod p multiplications when n is not divisible by p. If we take a look at Eq. (1), in terms of the big-Oh notation, running time of the algorithm presented is $O(\max\{\lfloor \frac{n}{p} \rfloor, p^2\})$. If p is equal to 1 (i.e., the smallest partition) or is equal to $\sqrt[2]{n}$, the complexity is $O(n)$, which is the complexity of the naive product algorithm. If p is greater than $\sqrt[2]{n}$, the complexity is greater than $O(n)$. If p is equal to n, the complexity is $O(n^2)$. This is the case when there is a single partition, which means no partitioning is applied.

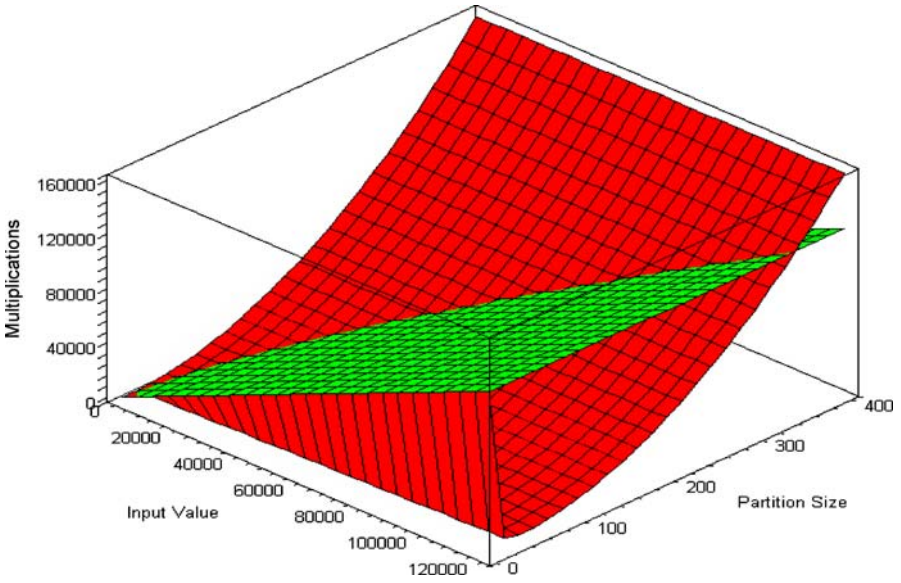

Fig. 5 Total number of multiplications necessary for computing factorials. The green plane represents the number of multiplications required for the naive product factorial algorithm. The red surface represents the number of multiplications required for the p-sized partitioning fast factorial algorithm and has a minima

Figure 5 shows a comparison of the number of multiplications necessary for computing factorials. A slight modification to Eq. 1 was necessary in order to graph the red surface that represents the number of multiplications needed for the fast factorial algorithm. A replacement of n mod p (which returns a value between 0 and $p-1$) with p is used as an approximation that allows a simplified analysis and smoothes out discontinuities caused by the mod function in the graph. The complexity described in the paragraph above correlates with Figure 5.

Comparing the two algorithms in terms of the big Oh-notation does not clearly show a difference in runtime performance. The fast factorial algorithm can guarantee fewer multiplications by choosing an optimal partition value p for a given input value n. Figure 5, the three dimensional plot of the running time in terms of multiplications necessary, indicates that there exists a value of p for a given n such that the fast factorial algorithm results in fewer multiplications. This can also be shown analytically by taking second derivatives. The next section covers the analysis for determining optimum values for p that shows (in terms of the actual big Oh-notation) that the fast factorial algorithm is quite efficient.

4 The analysis of the p-sized partitioning algorithm

In order to see the best performance of the fast factorial algorithm, we need to determine an optimal partitioning size algebraically. Considering the modification to Eq. (1) as described in the previous section, we have the following equation:

$$T_2(n,\, p) = \left\lfloor \frac{n}{p} \right\rfloor + p^2 + p. \tag{2}$$

Also, note that T($n,\, p$) $< T_2\ (n,\, p)$ holds, where, T($n,\, p$) is the number of multiplications required by the p-sized partitioning fast factorial algorithm, and $T_2\ (n, p)$ is an upper bound for T($n,\, p$). Differentiating Eq. (2) with respect to p, we get:

$$T_2'(n,\, p) = \mathrm{d}\left(T_2\,(n,\, p)\right)/\mathrm{d}p = \frac{-n}{p^2} + 2p + 1. \tag{3}$$

The second derivative in Eq. (3) yields:

$$T_2''(n,\, p) = \mathrm{d}\left(T_2'\,(n,\, p)\right)/\mathrm{d}p = \frac{2n}{p^3} + 2. \tag{4}$$

Since $p > 0$ and $n > 0$, then $T_2''(n,\, p) > 0$. This means that the surface defined by Eq. (2) has a minimum, where $T_2'(n,\, p) = 0$. Hence,

$$n = 2p^3 + p^2. \tag{5}$$

Finally, we have to solve Eq. (5) for p in terms on n, which is not trivial. However, by omitting p^2 since $2p^3$ is the dominating term, we obtain an easily solvable equation which happens to be a good estimate for the optimal value of p in the fast factorial algorithm. Thus, the

estimated optimal value for the partition size p is:

$$\tilde{p} \approx \sqrt[3]{\frac{n}{2}}. \tag{6a}$$

Since p is an integer number, then $\hat{p}$, the optimal integer value for p is:

$$\hat{p} \approx \left\lfloor \sqrt[3]{\frac{n}{2}} + 0.5 \right\rfloor. \tag{6b}$$

The algorithm is most efficient when it uses $\hat{p}$, the optimal integer value for the partition size p. The $\hat{p}$ value needs to be computed first, and passes as a parameter to the algorithm.

By substituting $\tilde{p}$ in Eq. (2), we obtain

$$\begin{aligned} T_2(n, \tilde{p}) &= 2^{\frac{1}{3}} n^{\frac{2}{3}} + (1/4)^{\frac{1}{3}} n^{\frac{2}{3}} + (1/2)^{\frac{1}{3}} n^{\frac{1}{3}} \\ &= 1.89 n^{\frac{2}{3}} + 0.79 n^{\frac{1}{3}}. \end{aligned} \tag{7}$$

Also, $T(n, p) < T_2(n, p)$ holds. Again, $T(n, p)$ is the number of multiplications required by the p-sized partitioning fast factorial algorithm. Clearly, $T_2(n, \tilde{p})$ is $O(n^{\frac{2}{3}})$. Since $T_2(n, \tilde{p})$ is an upper bound for $T(n, \tilde{p})$, by definition of the big-Oh notation, $T(n, \tilde{p})$ is also $O(n^{\frac{2}{3}})$ in terms of number of multiplications necessary. Now, for larger values of n, we have the number of multiplications reduced to $O(n^{\frac{2}{3}})$, which is significantly smaller than the number of multiplications needed for the linear difference, Boiten split, and naive product algorithms, which are all $O(n)$. The number of multiplications necessary for the linear difference algorithm is $T(n) = \frac{n}{4}$, and for the Boiten split algorithm is $T(n) = \frac{n}{4}$. For example, in order to compute 32000!, we would need 31,999 multiplications for the naive product, 16,014 for the Boiten split, 8,000 for the linear difference, and only 1,930 multiplications for the optimized p-sized partitioning fast factorial algorithm with optimal partition size $\hat{p} = 25$.

Table 2 shows actual execution times for several factorial algorithms to compute factorials of numbers, ranging from 250 to 128,000. Execution times in Table 2 were obtained by

Table 2 Actual execution times of different factorial algorithms (in milliseconds). The p-sized partitioning fast factorial algorithm uses the optimal partition size $\hat{p}$

n	p-sized partitioning		Linear difference	Boiten split	Naive product
250	($\hat{p} = 5$)	0.26	0.07	0.07	0.14
500	($\hat{p} = 6$)	0.55	0.23	0.20	0.47
1000	($\hat{p} = 8$)	1.29	0.85	0.72	1.61
2000	($\hat{p} = 10$)	3.85	3.36	2.63	6.07
4000	($\hat{p} = 13$)	11.98	16.26	9.82	23.26
8000	($\hat{p} = 16$)	39.10	73.61	36.64	94.65
16000	($\hat{p} = 20$)	132.85	321.89	140.99	402.80
32000	($\hat{p} = 25$)	431.81	1389.49	559.58	1673.59
64000	($\hat{p} = 32$)	1453.43	7059.77	3263.14	8752.17
128000	($\hat{p} = 40$)	4826.95	31410.53	14576.11	45459.05

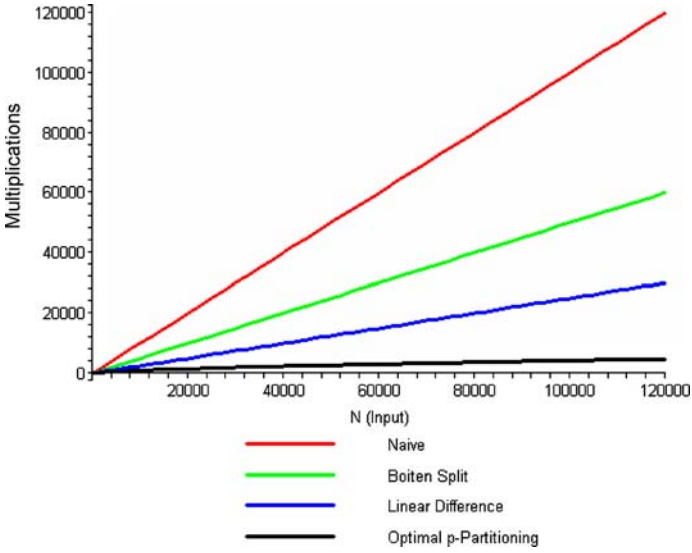

Fig. 6 Asymptotic comparison of total multiplications necessary for several factorial algorithms. The p-sized partitioning fast factorial algorithm is labeled as "Optimal p-Partitioning"

finding the minimal time of five repeated procedure calls to each algorithm. The algorithms chosen are the optimized p-sized partitioning, linear difference, Boiten split, and the naive product. These algorithms were implemented in Python (version 2.4.1) on the Windows XP operating system, with a Pentium 4 processor (3.2 GHz) and 504 MB of RAM. Python was chosen for its built-in support for large integers. For smaller integers, the other algorithms showed slightly smaller execution times. This is due to the fact that Python is an interpreted language (i.e., no potential for code optimization). The p-sized partitioning algorithm showed reasonably smaller execution times than other three algorithms for larger integers. This is due to the fact that the complexity of multiplication increases as numbers get larger. As a result, multiplication dominates the execution time. The Boiten split showed execution times that are about half of those of the linear difference algorithm. This is due to the overhead of the code being interpreted (more lines of code are being interpreted in the linear difference) and faster operations used by the Boiten split (i.e., shifts).

Figure 6 shows the asymptotic comparison of the number of multiplications necessary for the optimized p-sized partitioning, linear difference, Boiten split fast factorial algorithms, and the naive product algorithm. The graph for the optimized p-sized partitioning algorithm indicates that for larger numbers, the algorithm is quite effective. This is also consistent with the experimental results on physical execution time for larger numbers (see Table 2).

5 Conclusions

We have demonstrated an algorithm that significantly reduces the number of multiplications for computing factorials utilizing a difference table and partitioning. The reduction of multiplications is achieved by correctly transforming them into additions and choosing an optimal partition size. Since addition operations in a digital computer are much faster than multiplication operations [6, 7], the reduction of multiplications in computing factorials reduces

the execution time. This was shown both analytically and experimentally. The reduction in number of multiplications would naturally play a larger role in saving execution time in large number math packages. The algorithm has potential for further improvement as well as parallelization. For example, eliminating multiplication of even numbers by using appropriate bit shift operations would improve the actual speed. We are currently working on potential improvement and a parallel version of the algorithm for further reduction in execution time.

References

1. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edition. McGraw-Hill
2. Sedgewick R (1992) Algorithms in C++. Addison-Wesley
3. National Institute of Standards and Technology. 'Dictionary of Algorithms and Data Structures, http://www.nist.gov/dads/, 2005.
4. Borwein P (1985) On the complexity of computing factorials. J Algorithms 6:376–380
5. Boiten A (1992) Factorisation of the factorial—An example of inverting the flow of computation. Periodica Polytechnica Ser El Eng 35(2):77–99
6. Paul R (2000) SPARC architecture, assembly language programming, and c, 2nd edition. Prentice Hall
7. Chaudhuri R (2003) Do the arithmetic operations really execute in constant time? The SIGCSE Bulletin 35(2):43–44
8. Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms. Addison-Wesley.
9. Thompson H, Ugur A (2004) Fast computation of factorials of numbers. In: Proceedings of The 2004 International Conference on Algorithmic Mathematics and Computer Science (AMCS'04). CSREA Press, Vol. II, pp. 419–422
10. Brualdi RA (1999) Introductory combinatorics, 3rd edition. Prentice Hall