AVANT-PROPOS

This is a small excerpt from *Swing, divide and conquer the factorial*, a manuscript I wrote around 2008, for use on my website Fast Factorial Functions.

I would like to thank Bernardo Meurer who inquired some more information about the swing-algorithm and then encouraged me to translate the relevant sections (which were written in German) to English. I also thank Bernardo Meurer and Sander Hartkamp for their help in the translation.

Peter Luschny

CONTENTS

1 A NEW KIND OF FACTORIAL FUNCTION 2
- 1.1 A decomposition of the factorial function 2
- 1.2 The prime factors of the swinging factorial. 3
- 1.3 The dsc–algorithm for computing n! 5
- 1.4 Swing-time: the time complexity of $n\wr$. 6
- 1.5 Factorial-time: The time complexity of n! 7
- 1.6 A recurrence for $n\wr$ 8
- 1.7 Implementation of the dsc-factorial 8

2 APPENDIX: LISTINGS 9
- 2.1 dsc-factorial with prime factorization (pseudo code) 9
- 2.2 dsc-factorial with recursion (SageMath) 10
- 2.3 dsc-factorial with prime factorization (SageMath) 11

1 A NEW KIND OF FACTORIAL FUNCTION

1.1 A DECOMPOSITION OF THE FACTORIAL FUNCTION

„Unfortunately, we can't compute factorials efficiently" Graham, Knuth and Patashnik deplore in their standard reference *Concrete Mathematics* [3, p.133]. But how good can we be? Arnold Schönhage even mentions a competition to calculate the factorial faster and faster [5]. Here we will give a *divide-and-conquer* recurrence for $n!$, which, by making use of prime factorization, can be turned into a fast algorithm. We will investigate its time complexity and provide example implementations.

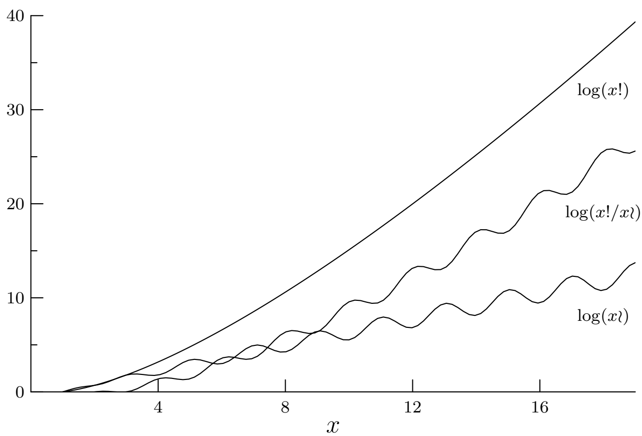

Figure 1 – The dragon's mouth: A decomposition of the factorial

Starting point is the *dragon*-representation of the factorial function, a multiplicative decomposition of the factorial into two oscillating functions, the name of which derives from their graphical presentation (fig. 1). We start by exploring the lower 'row of teeth' in the picture, the *swinging factorial* $x\wr$.

n	0	1	2	3	4	5	6	7	8	9	10	11
$n\wr$	1	1	2	6	6	30	20	140	70	630	252	2772

Table 1 – The swinging factorial

1.2 THE PRIME FACTORS OF THE SWINGING FACTORIAL.

The plot in figure 1 leaps ahead of our development. We start with the discrete case. Let $n \in \mathbb{N} = \{0, 1, 2, \ldots\}$ be a natural number. By the *(monotonic) factorial* n we understand the product $n! = 1 \cdot 2 \cdot \ldots \cdot n$ and by the *swinging factorial* of n we understand the ratio of $n!$ to $\lfloor n/2 \rfloor!^2$ and denote it by

$$n\wr = \frac{n!}{\lfloor n/2 \rfloor !^2} \qquad (n \geqslant 0)\,. \tag{1.1}$$

$\lfloor x \rfloor$ is the *floor*-function giving the largest integer not greater than x. The first few values of $n\wr$ are displayed in table 1.

We write $\binom{n}{k} = n!/(k!(n-k)!)$ for the binomial coefficient. Let $\mu_n = \binom{n}{\lfloor n/2 \rfloor}$ denote the *middle binomial coefficient*. Then $n\wr = \mu_n$ if n is even, otherwise $n\wr = \mu_n((n+1)/2)$. Thus $n\wr$ is always an integer.

Equivalently the swinging factorial can be defined as a *trinomial coefficient*.

$$n\wr = \binom{n}{\lfloor n/2 \rfloor, [n \text{ odd}], \lfloor n/2 \rfloor} \tag{1.2}$$

Here we make use of the Iverson brackets $[\cdot]$ defined as $[b] = 1$ if the statement b is true and 0 otherwise. Thus $[n \text{ odd}]$ is 1 if n is odd and 0 if n is even.

The starting point of our considerations is the prime factorization of $n\wr$. We make use of the mod-operation defined by $x \bmod m = x - m \lfloor x/m \rfloor$ for $m \neq 0$, x otherwise.

Theorem 1 *Let $\ell_p(n\wr)$ denote the exponent of the prime p in the prime factorization of $n\wr$. Then*

$$\ell_p(n\wr) = \sum_{k \geqslant 1} \left\lfloor \frac{n}{p^k} \right\rfloor \bmod 2\,. \tag{1.3}$$

In consequence $\ell_p(n\wr) \leqslant \log_p(n)$ *and* $p^{\ell_p(n\wr)} \leqslant n$. *If* p *is an odd prime then* $\ell_p p^a\wr = a$. *Special cases of (1.3) are:*

$$
\begin{array}{llcllcl}
(a) & \lfloor n/2 \rfloor & < p \leqslant & n & \Rightarrow & \ell_p(n\wr) = 1 \\
(b) & \lfloor n/3 \rfloor & < p \leqslant & \lfloor n/2 \rfloor & \Rightarrow & \ell_p(n\wr) = 0 \\
(c) & \sqrt{n} & < p \leqslant & \lfloor n/3 \rfloor & \Rightarrow & \ell_p(n\wr) = \lfloor n/p \rfloor \bmod 2 \\
(d) & 2 & < p \leqslant & \sqrt{n} & \Rightarrow & \ell_p(n\wr) < \log_2(n) \\
(e) & & p = & 2 & \Rightarrow & \ell_p(n\wr) = \sigma_2(\lfloor n/2 \rfloor)
\end{array}
$$

Here $\sigma_2(n)$ *is the number of 1's in the binary representation of* n.

Proof: From Legendre's theorem on the prime factorization of $n!$ (see [3, 4.4]) we get

$$
\begin{aligned}
\ell_p(n!/\lfloor n/2 \rfloor!^2) &= \ell_p(n!) - 2\ell_p(\lfloor n/2 \rfloor!) \\
&= \sum_{k \geqslant 1} \lfloor n/p^k \rfloor - 2 \sum_{k \geqslant 1} \lfloor \lfloor n/2 \rfloor / p^k \rfloor \qquad (1.4) \\
&= \sum_{k \geqslant 1} \Big(\lfloor n/p^k \rfloor - 2 \lfloor \lfloor n/p^k \rfloor /2 \rfloor \Big)
\end{aligned}
$$

Since $j - 2\lfloor j/2 \rfloor = j \bmod 2$ by definition (1.3) follows. In the case of (a), (b) and (c) the summation range reduces to $k = 1$. In case (a) we have $\lfloor n/p \rfloor = 1$, in case (b) $\lfloor n/p \rfloor = 2$. Thus these three cases are implied by (1.3). (d) is a consequence of $\log_b(n) < \log_2(n)$ for $2 < b$ and (e) follows from (1.4) and from the identities $\ell_2(n!) = n - \sigma_2(n)$ and $\sigma_2(\lfloor n/2 \rfloor) = \sigma_2(n) - n \bmod 2$. Finally for p an odd prime

$$
\ell_p(p^a\wr) = \sum_{k \geqslant 1} \lfloor p^a/p^k \rfloor \bmod 2 = \sum_{1 \leqslant k \leqslant a} p^{a-k} \bmod 2 = \sum_{1 \leqslant k \leqslant a} 1 = a\,.
$$

The properties of $n\wr$ described in theorem 1 show that $n\wr$ is a hybrid between the factorial function and the binomial coefficient. And they put the swinging factorial in the kernel of a *divide-and-conquer* algorithm for the computation of the monotone factorial.

In consequence of theorem (1) the logarithm of the swinging factorial can be written as

$$
\log(n\wr) = \sum_{k \geqslant 1} \sum_{p \ \text{prim}} \left[\left\lfloor \frac{n}{p^k} \right\rfloor \ \text{odd} \right] \log p\,. \qquad (1.5)
$$

This is a finite sum with $k \leqslant \log_2 n$ and $p \leqslant n$. Note that we use here again the Iverson brackets. The simplicity of the representation (1.5) let us expect interesting number theoretical properties of $n\wr$.

1.3 THE DSC–ALGORITHM FOR COMPUTING n!

Let us return to the question posed at the beginning: how to calculate the factorial of $n!$ efficiently. We now know that the factorial can be calculated by means of the square recurrence $n! = \lfloor n/2 \rfloor!^2 \, n\wr$ with base case $0! = 1$. It is important to understand that the swinging factorial $n\wr$ will be determined by prime factorization. By theorem 1 the prime factors of $n\wr$ can be computed easily and found quickly using the sieve of Eratosthenes (or any other prime number sieve).

Let's call this algorithm the *dscFactorial* (*dsc* is formed from *divide, swing and conquer*). As an example we show the calculation of 62!.

$$\begin{aligned}
3\wr &= 2 \cdot 3 \\
7\wr &= 2^2 \cdot 5 \cdot 7 \\
15\wr &= 2^3 \cdot 3^2 \cdot 5 \cdot 11 \cdot 13 \\
31\wr &= 2^4 \cdot 3^2 \cdot 5 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \\
62\wr &= 2^5 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 \cdot 61 \\
\\
62! &= (((((1\wr)^2\, 3\wr)^2\, 7\wr)^2\, 15\wr)^2\, 31\wr)^2\, 62\wr
\end{aligned}$$

First one determines the prime exponents of $\lfloor n/2^k \rfloor \wr$ for $k \geqslant 0$. In doing so one takes advantage of the relation $\lfloor \lfloor n/2 \rfloor / p^k \rfloor = \lfloor \lfloor n/p^k \rfloor / 2 \rfloor$ which allows one to determine these exponents using only simple shift operations on $\lfloor n/p^k \rfloor$.

Computationally valuable is also the relation $p^{e_p(n\wr)} \leqslant n$ as it allows to calculate the part of $n\wr$ which is based on p in a computer register as long as n fits into the register. In this case no book–keeping of the prime exponents is needed.

After putting all prime parts of $\lfloor n/2^k \rfloor$ on the k-th of $\log_2 n$ lists a *divide-and-conquer* product is applied to these lists. Assuming that this product, in turn, is building on a *divide-and-conquer* multiplication algorithm, say that of Karatsuba, the calculation passes through three levels of *divide-and-conquer* strategies before actually a multiplication is performed.

Finally we take advantage of the fact that multiplication by powers of 2 on binary computers is very fast. We therefore calculate only the odd part of the swinging factorial and then multiply the result of the recurrence with $2^{n-\sigma_2(n)}$ where $\sigma_2(n)$ is the number of digits of n in the binary base.

One of the first questions that arise in the implementation of the *dsc-Factorial* is how much memory for the lists of factors must be allocated. This is equivalent to ask for the number of prime factors of $n\wr$. Helpful for practical applications are the following simple bounds on the *number of all prime factors* of the swinging factorial $\Omega(n\wr)$. The bounds were verified numerically in the specified range. We conjecture that they hold for all $n \geqslant 25$.

In the range $25 \leqslant n \leqslant 10^6$ *the following bounds hold:*

$$\left\lfloor \frac{n}{\log_2(n/2)} + n^{1/6} \right\rfloor \leqslant \Omega(n\wr) \leqslant \left\lfloor \frac{n}{\log_2(n/2)} + n^{1/2} \right\rfloor$$

1.4 SWING-TIME: THE TIME COMPLEXITY OF $n\wr$.

Let us consider now how much time the calculation of the product of these prime factor lists takes when using the recursive divide-and-conquer method. We understand this as the calculation of $n\wr$ using the recurrence $n\wr = P(1, \Omega(n\wr))$ and

$$P(k,n) = P(k,l) \cdot P(l+1,n), \quad (k < n, \ \ l = \lfloor (k+n)/2 \rfloor)$$

with $P(n,n) = P(n)$, where $P(n)$ is the n-th prime factor from the list, which we will denote by $F(n\wr)$. Since the bit length of $n\wr$ is asymptotically n the recursion traverses $\log n$ stages, where in the last stage two factors of bit length $n/2$ are multiplied which were recovered from 4 factors with the bit length $n/4$, and so on. Now assume that we have two binary integers with bit length n which we can multiply in $M_{asy}(n) = \beta n \log(\alpha n)(1 + \log\log(\alpha n))$ $(\alpha, \beta > 0)$ time units (this is asymptotically achieved with the Schönhage-Strassen multiplication [5, S.208,6.1.33]), then we arrive at the asymptotic estimate of the time complexity of the product:

$$T_{\text{prod}} F(n\wr) \simeq \beta n \log\log(n) \sum_{1 \leqslant i \leqslant \lfloor \lg n \rfloor} \log(\alpha n 2^{-i})$$

$$\simeq \frac{\alpha}{\beta} n\,(\log n)^2 \log\log n$$

If one uses the Sieve of Eratosthenes to create the prime factor lists it can be shown that for sufficiently large n the time $T_{\text{prim}} F(n\wr)$ for determining the prime factors of $n\wr$ can be neglected compared with the time to compute the product (see for instance [1, S. 297]). Thus we come to the conclusion that the time $T_{\text{swing}}(n)$ for the calculation of the swinging factorial of n is asymptotically limited by the order $n(\log n)^2 \log\log n$.

$$T_{\text{swing}}(n) = T_{\text{prod}} F(n\wr) + T_{\text{prim}} F(n\wr) = O(n(\log n)^2 \log\log n)$$

1.5 FACTORIAL-TIME: THE TIME COMPLEXITY OF $n!$

Let $T_{\text{mult}}(n, m)$ the time a multiplication of two factors with bit lengths n and m respectively requires, and $T_{\text{quad}}(k)$ the time the squaring of a number with bit length k requires. Since the bit length of $n!$ asymptotically is $n \log n$ we can estimate the time $T_{\text{dsc}}(n)$ for calculating $n!$ using the *dsc-algorithm* for big n as follows: With $\lambda_n = \lfloor \log_2(n) \rfloor$

$$\begin{aligned}
T_{\text{dsc}} \sim & \sum_{0 \leqslant l < \lambda_n} T_{\text{mult}}(2\lfloor n/2^{l+1} \rfloor \log\lfloor n/2^{l+1} \rfloor, \lfloor n/2^{l} \rfloor) \\
& + \sum_{0 \leqslant l < \lambda_n} T_{\text{quad}}(\lfloor n/2^{l+1} \rfloor \log\lfloor n/2^{l+1} \rfloor) \\
& + \sum_{0 \leqslant l < \lambda_n} T_{\text{swing}}(\lfloor n/2^{l} \rfloor).
\end{aligned}$$

If we insert in $T_{\text{mult}}(n, n)$ and $T_{\text{quad}}(n)$ the asymptotic time complexity of the Schönhage-Strassen multiplication $M_{\text{asy}}(n)$ it is easily seen that the first two sums are limited by the asymptotic order $n(\log n)^2 \log\log n$.

Next the sum of calculation times of the swinging factorials can be estimated by a constant multiple of $T_{\text{swing}}(n)$. Therefore, from the result of the last section, we conclude that

$$T_{\text{dsc}}(n) = O(n(\log n)^2 \log\log n).$$

Thus the time complexity of the *dscFactorial* is asymptotically limited by the same order as the multiplication of binary numbers of the length

$\log(n!)$. The fastest previously known algorithm for calculation of $n!$ is based on the prime factorization of $n!$ which is evaluated with the method of nested squaring. So described by A. Schönhage et alia [5, S.225] (and in a similar form by P. B. Borwein [2]). Based on the computational model of a multi-band Turing machine Schönhage gives for the time complexity of his algorithm the same asymptotic order as we have found for T_{dsc}.

1.6 A RECURRENCE FOR $n≀$

The idea of calculating the factorial with the help of the swinging factorial can also be implemented without using prime factorization. In this scenario the swinging factorial is calculated using a recurrence. This leads to an efficient algorithm which we give in the appendix as an Sage-Python implementation. Benchmarks indicate that this might be the fastest known algorithm in this category [4].

The most surprising feature of this algorithm is that strong use of division is made – something that does not come to one's mind when approaching the factorial function naively.

1.7 IMPLEMENTATION OF THE DSC-FACTORIAL

Implementations of *dscFactorial* exist in various programming languages. They can be found for instance on the web-page *Fast Factorial Functions* [4] where a total of 21 different algorithms for the computation of the factorial are showcased and compared to each other; besides the dscFactorial also the methods of Arnold Schönhage, Peter Borwein and Ilan Vardi.

A greatly simplified implementation (compared to the description given in the last section) written in pseudo-code is given in the listing below. It consists of the three functions `Factorial`, `PrimeSwing` and `Product`. Note that we assume that the smallest index of a list is 0 and that `index` is a global variable. An optimized implementation with the computer algebra system *SageMath* can be found at the end of this chapter. These implementations are written in the Python dialect of SageMath. Algorithm 2 uses the Python library function `bisect_left` and the SageMath function `prime_range`.

2 APPENDIX: LISTINGS

2.1 DSC-FACTORIAL WITH PRIME FACTORIZATION (PSEUDO CODE)

```
Factorial(n)
   if n < 2 then return(1) end_if
   return(Factorial(⌊n/2⌋)^2 PrimeSwing(n))

PrimeSwing(n)
   count ← 0
   for prime in Primes(2...n) do
      q ← n;p ← 1
      repeat
        q ← ⌊q/prime⌋
        if q is odd then p ← p · prime end_if
      until q = 0
      if p > 1 then FactorList[count++] ← p end_if
   end_for
   index ← 0
   return(Product(FactorList, count))

Product(list, len)
   if len = 0 then return(1) end_if
   if len = 1 then return(list[index++]) end_if
   hlen ← ⌊len/2⌋
   return(Product(list, len − hlen) · Product(list, hlen))
```

2.2 DSC-FACTORIAL WITH RECURSION (SAGEMATH)

```
def factorial(n):

    def product(m, len):
        if len == 1: return m
        if len == 2: return m * (m - 2)
        hlen = len >> 1
        return product(m-hlen*2, len-hlen) * product(m, hlen)

    def odd_factorial(n):
        if n < 5:
            oddFact    = [1,1,1,3,3][n]
            sqrOddFact = [1,1,1,3,3][n//2]
        else:
            sqrOddFact, oldOddFact = odd_factorial(n//2)
            len = (n - 1) // 4
            if (n % 4) != 2: len += 1
            high = n - ((n + 1) & 1)
            oddSwing = product(high, len) // oldOddFact
            oddFact = (sqrOddFact**2) * oddSwing
        return (oddFact, sqrOddFact)

    def eval(n):
        if n < 10: return mul(range(2,n+1))
        bits = n - sum(n.digits(2))
        return odd_factorial(n)[0] * 2**bits

    return eval(n)
```

```
def factorial(n):

    def product(s, n, m):
        if n > m: return 1
        if n == m: return s[n]
        k = (n + m) // 2
        return product(s, n, k) * product(s, k + 1, m)

    def swing(m, primes):
        if m < 4: return [1,1,1,3][m]

        s = bisect_left(primes, 1 + isqrt(m))
        d = bisect_left(primes, 1 + m // 3)
        e = bisect_left(primes, 1 + m // 2)
        g = bisect_left(primes, 1 + m)

        factors = primes[e:g]
        factors += filter(lambda x: (m//x)&1 == 1, primes[s:d])
        for prime in primes[1:s]:
            p, q = 1, m
            while True:
                q //= prime
                if q == 0: break
                if q & 1 == 1:
                    p *= prime
            if p > 1: factors.append(p)

        return product(factors, 0, len(factors) - 1)

    def odd_factorial(n, primes):
        if n < 2: return 1
        return (odd_factorial(n//2,primes)**2)*swing(n,primes)

    def eval(n):
        if n < 10: return product(range(2, n + 1), 0, n-2)
        bits = n - sum(n.digits(2))
        primes = prime_range(2, n + 1)
        return odd_factorial(n, primes) * 2**bits

    return eval(n)
```

BIBLIOGRAPHY

[1] E. Bach and J. Shallit. *Algorithmic Number Theory*. MIT Press, 1996. (Cited on page 7.)

[2] P. B. Borwein. On the complexity of calculating factorials. *Journal of Algorithms*, 6:376–380, 1985. (Cited on page 8.)

[3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989. (Cited on pages 2 and 4.)

[4] Peter Luschny. The homepage of factorial algorithms. http://www.luschny.de/math/factorial/FastFactorialFunctions.htm, 2002. (Cited on page 8.)

[5] A. Schönhage, A. F. W. Grotefeld, and E. Vetter. *Fast Algorithms, A Multitape Turing Machine Implementation*. BI Wissenschaftsverlag, 1994. (Cited on pages 2, 6, and 8.)