Comments on Factorial Programs

Richard J. Fateman
Computer Science Division, EECS
University of California, Berkeley

April 11, 2006

Abstract

Programs to compute the factorial function are used as classic illustrations of iteration and recursion. While such programs can be written in any programming language, many of the example solutions are inefficient, pointless, or both.

1 Introduction

Factorial programs can be written in any programming language, but they are especially compact in languages that are functional-style. Here we can see them in Lisp:

```
;; The obvious recursive one

(defun fact1 (x)(if (< x 1) 1 (* x (fact1 (1- x)))))

;; Here's another recursive function, called as  (fact2 x 1)
;; Fact2, properly compiled, runs as an ``iterative process''
;; and does not need call-stack space proportional to n.

(defun fact2(x n)(if (< x 1) n (fact2 (1- x)(* n x))))

;; Here are two programs written in a textually iterative style
;; using Common Lisp iterators dotimes and do.

(defun fact3 (x)(let ((ans 1))(dotimes (i x ans)(setq ans (* ans (1+ i))))))

(defun fact4 (x)
 (do ((i x (1- i))  (ans 1 (* ans i)))
     ((= i 0) ans)))
```

Factorial programs we see in introductory texts for Algol-like languages including C, Basic, Java, Pascal, and Fortran use those languages' default arithmetic semantics, and therefore use a limited-size integer data type. Since factorials grow so fast, such implementations are silly: Only a few inputs can be handled without overflow. Common Lisp (and other higher-level languages) have integer data types of arbitrary precision; most implementations can handle huge numbers, generally subject only to total memory- or address-space limits.

2 Trivializing a Factorial Program

13! is 6,227,020,800, which exceeds $2^{31} - 1$ or 2,147,483,647, so on a 32-bit machine using ordinary 32-bit integers, you cannot compute more than 12 different factorials, and out of those, 0!=1, 1!=1, and maybe 2!=2 are not very interesting. At most we are talking about these pairs:

```
(0 1)
(1 1)
(2 2)
(3 6)
(4 24)
(5 120)
(6 720)
(7 5040)
(8 40320)
(9 362880)
(10 3628800)
(11 39916800)
(12 479001600)
```

These can easily be stored in an array. If you are at all concerned about speed or simplicity, the factorial program could be something like this (yes, Lisp has arrays, accessed with `aref`):

```
(defun fact5 (x)
  (if (typep x '(integer 0 12))
      (aref #(1 1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600)
            x)
      (error "~S is not a valid argument to FACT5" x)))
```

Say you are willing to use floating-point numbers. You can compute up to 170! = about 7.3d+306, but 171 will overflow. These too could be stored in an array, and the answers will be exactly correct up to the factorial of 22. Beyond that the truncation error makes the answers inexact.

You might object that you are trying to compute factorials, and storing them in a table is unfair. Well, are you trying to compute them, or just supply them as fast as possible? Many programs store auxiliary data: a cosine program might have $\pi/8$ precomputed, as well as data consisting of coefficients of an approximating polynomial. A short table of factorials, if indeed that helps get the result fast, seems plausible from that perspective.

3 Non-trivial Factorials

Let us say you want to reconsider this calculation so that factorial of more numbers is possible, using "bignumber" arithmetic. As an example, compute 20,000! which has over 77,300 decimal digits.

How should you compute such huge numbers?

If your objective is to compare different implementations of bignumber operations, it probably doesn't matter if you have the most efficient factorial. The question then is: whatever algorithm you use runs a particular sequence of bignumber operations. Is that what you want to time? You can choose, depending on the algorithm, to arrange the testing so that operations are mostly or entirely small X big numbers, or (medium-big) X (medium-big). The latter situation occurs if you write a program to clump together (by multiplication) some grouping of the operands, instead of accumulating the product by successive multiplications by the next integer.

Any of the Lisp programs above will compute factorial, even for rather large numbers. The particular programs shown above will spend most of their time multiplying small X big.

Computer algebra systems (CAS) and libraries intended to be used for computational number theory are not only prepared to use bignumbers, but must be prepared, generally, to compute large results, including those which, in the ordinary course of events, don't seem particulary useful. Nevertheless, these questions should be answered as fast as practicable– perhaps to maintain a certain level of dignity.

Let's turn to much faster programs. For example consider the one below, even though it also does n multiplies. (It is a good puzzle to figure out why, and the reader might wish to figure this out. Hint: it is faster to multiply equal-size numbers especially when they are small.)

```
(defun fact6(x)(k x 1))

;; k(n,m) is a kind of factorial, n*(n-m)*(n-2m)* .....

(defun k (n m) ;; (k n 1) is n!
  (if (<= n m) n
    (* (k n (* 2 m))  (k (- n m)(* 2 m)))))
```

That's all that's needed to be fairly good.

We can try to gild the lily and use some other facts about factorials such as $k(2n, 2) = 2^n k(n, 1)$. Here's how

```
;; (kg n) is n!.
;;  kg uses ash, arithmetic shift. (ash m 1) is 2n.
;; (ash n -1) is n/2.

(defun kg (n)
  (let ((shift 0))
    (labels
        ((kg1 (n m)
           (cond ((and (evenp n)(> m 1))
                  (incf shift (ash n -1))
                  (kg1 (ash n -1) (ash m -1)))
                 ((<= n m) n)
                 (t (* (kg1 n (ash m 1))
                       (kg1 (- n m)(ash m 1)))))))
      (ash (kg1 n 1) shift))))
```

This `kg` program is faster than the `k` program immediately above it because it defers multiplications by factors of two until the end, and then does it as one big shift. For example, computing 20,000! ends with a whopping arithmetic left shift of 19,995 bits.

Earlier we suggested storing the first 12 factorials in a table. We can't realistically save "all" factorials in a table, but we can remember some of them; pre-computing 100! makes it fast to compute 101!. We can also save ones that were just previously computed, foiling any timing tests that compute the same value repeatedly in a loop.

```
;; kk computes the product of only the terms down to min, as needed.
;; (* (kk 100 1 50)(kk 49 1 1))  is the same as (kk 100 1 1).
;; (kk n 1 n) is simply n.
```

```
(defun kk (n m min) ;; (kk n 1 1) is n!
  (declare (fixnum n m min))
  (cond ((< n min) 1)
        ((<= n m) n)
        (t (* (kk n (ash m 1) min)
              (kk (- n m)(ash m 1) min)))))

(defun kmemfac(n)           ;; a factorial with memory
  (let ((z (lookupfact n)))
         ;; find z, the largest integer <= n such that we know z!
  (if (= (car z) n)(cdr z) ;; return the answer if we found it.
    (rememberfact n        ;; otherwise compute and remember this.
                  (* (kk n 1 (1+ (car z)))
                     (cdr z))))))

;; create a table for factorials
(defvar oldfacts nil) ;oldfacts is a global name

;; if you want to re-initialize the memory, call (clearfact)

(defun clearfact()(setf oldfacts (make-array 0 :adjustable t :fill-pointer t))
                  (vector-push-extend (cons 0 1) oldfacts)) ;install one answer.

(clearfact)  ; clear it now.

;; we could precompute factorials of 100, 1000, 5000,  by
;; (progn (kmemfac 100)(kmemfac 1000)(kmemfac 5000) nil)

(defun rememberfact(n f)
  (vector-push-extend (cons n f) oldfacts)
  (setf oldfacts (sort oldfacts #'> :key #'car))
   f)

(defun lookupfact(n)
  (loop for i from 0 to (length oldfacts)
      do (if
             (<=  (car (aref oldfacts i)) n)
             (return (aref oldfacts i)))))
```

We have run instrumented versions of these programs to figure out where they are spending most of their time. Say you wish to compute 20,000 factorial. How many "bignum" multiplications do you need? If you use one of the simple factorials, you compute about 19,987 products in which exactly one of the arguments is a bignum. Using the split-recursive version, only about 11,800 include a bignum, saving considerable time.

Using the table-based factorial, assume you wish to compute 10,000!. This uses about 5888 bignum multiplies, so the other 4,112 are done "on the cheap."

Now suppose you wish to compute 20,000!. It requires about 5,900 additional bignum multiplies.

4 What's Out There Already?

For some people the first and perhaps only experience with computing with arbitrary precision integers comes through using a CAS such as Mathematica or Maple. One way of testing them is to compute factorial of large numbers. So even if the computation is inherently not very useful it serves as an easily-typed benchmark.

How hard do the programmers of these systems work on making this task fast? We looked at the Macsyma CAS factorial program (actually, the open-source Maxima version, code unchanged since 1982.) Restated somewhat, it looks like this:

```
(defun maxfact (n &aux (ans 1)  )
  (let* ((vec (make-array (if (< n 100) 1 20) :initial-element 1))
         (m (length vec))
         (j 0))
 (declare (fixnum j n m j))
  (loop for  i from 1 to n
         do (setq j (mod i m))
         (setf (aref vec j) (* (aref vec j) i)))
  (dotimes (v m ans)
         (setq ans (* ans (aref vec v))))))
```

This routine, except for smallish input, works by setting up an array of 20 values and tries to accumulate the product in 20 pieces, and then multiplies them together. It occurred to us to use a circular list instead of an array with modular indexing, saving some time.

Here what it looks like

```
;; make a circular list of numbers 1,2, ... n, 1, 2, ...
(defun cloop(n) ;;   (cloop 5) is #1=(5 4 3 2 1 . #1#)
  (let*((start (list 1))
        (end start))
    (dotimes (i (1- n) (setf (cdr end) start)) ;loop around
      (setq start (cons (+ 2 i) start)))))

(setf  *print-circle* t) ;; helpful if you want to debug this...

(defun loopprod (l)
  ;; efficient product of all the approximately equal
  ;; numbers in the loop trying to keep the sizes of inputs
  ;; approximately balanced.  the circular loop l is destroyed in the
  ;; process.
  (cond ((eq (cdr l) l)(car l))
        (t (setf (car l)(* (car l)(cadr l)))
           (setf (cdr l)(cddr l))
           (loopprod (cdr l))))) ;tail recursion

;; Here is the factorial based on maxima's  but using a circular list for storage
(defun maxfact (n)
  (let* ((z (if (< n 100) 5 100))
         (aloop (cloop z))) ; heuristic: choose 5 or 100.
    (loop for  i from (1+ z)  to n
```

```
        do  (setf (car aloop) (* (car aloop) i))
            (setf aloop (cdr aloop)))
    (loopprod aloop)))
```

A simple profiling of factorial of 20,000 shows that most of these algorithms use 98 percent or more of their time in bignumber multiplication. Speeding this up by using GMP[1] especially with appropriate assembler code can make any of these algorithms considerably faster on this benchmark, if the Common Lisp arithmetic is relatively low tech. We observed a difference of 16X in Allegro Common Lisp (older version 7.0) when using GMP instead. The code below illustrates the set up for GMP. (support code available from the author.) Some versions of Lisp use GMP by default, and so such rewriting may not be necessary. The changed lines are indicated by ***

```
(defun gmaxfact (n) ;maxima version of factorial, in GMP
  (let* ((z (if (< n 100) 5 100))
         (aloop (gcloop z))) ; heuristic
    (loop for  i from (1+ z)  to n
        do (mpz_mul_si (car aloop) (car aloop) i) ;*** mult by signed int
           (setf aloop (cdr aloop)))
    (gloopprod aloop)))

(defun gcloop(n)
  (let*((start (list (gmpz-z (into 1))))         ;***
        (end start))
    (dotimes (i (1- n) (setf (cdr end) start))
      (setq start (cons (gmpz-z (into (+ 2 i))) start))))) ;***

(defun gloopprod (l)
  (cond ((eq(cdr l) l)(make-gmpz :z (car l))) ;***
        (t (mpz_mul (car l)(car l)(cadr l))   ;***
           (setf (cdr l)(cddr l))
           (gloopprod (cdr l)))))
```

A version of `kg` using GMP arithmetic is somewhat more elaborate. To make this more efficient we try to avoid allocating many mostly-small GMPZ integers. Instead we allocate a resource: a short list of temporaries (size $\log(n)$ for $n!$) for GMPZ numbers, and re-use them. It looks like this:

```
(defun gkg(n &optional (aa (alloc-gmpz))) ;put answer in 2nd arg, if there
  (declare (fixnum n)(optimize (speed 3)(safety 0)(debug 0)))
  (let ((res nil) (shift 0)
        (L (ceiling (cl::log n 2)))
        (a (gmpz-z aa)))
    (declare (fixnum L shift n))
    (labels
        ((k (n m ans res)
              (declare (fixnum n m))
              (cond
               ((and (evenp n)(cl::> m 1))
```

[1] `http://www.swox.com/gmp`

```
             (cl::incf shift (ash n -1))
             (k (ash n -1) (ash m -1) ans res))
            ((cl::<= n m)(mpz_set_si ans n))
            (t(let ((b (car res)))
                (k (cl::- n m) (setq m (ash m 1)) b (cdr res))
                (k n m ans (cdr res))
                (mpz_mul ans ans b)
                ans)))))

    (dotimes (i L)(push (gmpz-z (alloc-gmpz)) res)) ;set up resource
    (k n 1 a res)
    (mpz_mul_2exp a a shift)
    aa)))
```

How does this compare to various CAS? The time to compute 20,000! using `gmaxfact` or `gkg` and pentium-4 GMP arithmetic is about 0.021 seconds. Mathematica 5.1 on the same machine uses 0.030 seconds. How much faster is this than "just doing the arithmetic"? Mathematica uses 0.437 seconds for the program

```
f[n_] := Block[{p = 1}, Do[p *= i, {i, n}]; p]
```

and 0.366 seconds for the program

```
k[n_, m_] := If[n <= m, n , k[n, 2m]*k[n - m, 2m]]
```

Clearly Mathematica's factorial function is cleverer than such a loop!

Maple version 7 uses 0.250 seconds. A later Maple version apparently calls the GMP library factorial, which is about as fast as the fastest version we have seen (0.021 sec.) Maple and Mathematica also seem to memoize their results, so timing them has to be done carefully.

A Java program for split-recursive provided by Luschny[2] is a kind of unrolled version of `gkg`. We wrote two versions: one simply converting it to Lisp, `sprfact` and then a revision `gsprfact` to use GMP. The `gsprfact` program uses the same trick as `gkg` for allocating a list of GMP numbers as a resource. Both versions below are notably obscure, considering what is being computed is just the factorial function.

```
(defun sprfact(n) ;split recursive, based on P. Luschny's code
  (let  ((p 1) (r 1) (NN 1) (log2n (floor (log n 2)))
         (h 0) (shift 0) (high 1) (len 0))
    (labels ((prod(n)
                  (declare (fixnum n))
                  (let ((m (ash n -1)))
                    (cond ((= m 0) (incf NN 2))
                          ((= n 2) (* (incf NN 2)(incf NN 2)))
                          (t  (* (prod (- n m)) (prod m)))))))

      (loop while (/= h n) do
            (incf shift h)
            (setf h (ash n (- log2n)))
            (decf log2n)
            (setf len high)
            (setf high (if (oddp h) h (1- h)))
```

[2] `http://www.luschny.de/math/factorial/index.html`

```
            (setf len (ash (- high len) -1))
            (cond ((> len 0)
                   (setf p (* p (prod len)))
                   (setf r (* r p)))))
      (ash r shift))))

;;; unrolled sparse-recursive Lisp using GMP arithmetic...

(defun gsprfact(n)
  (declare (fixnum n))
  (let*  ((pp (into 1)) (p (gmpz-z pp))
          (rr (into 1)) (r (gmpz-z rr))
          (NN 1)
          (mlog2n (-(floor (log n 2))))
          (h 0) (shift 0) (high 1) (len 0) (ans (gmpz-z (into 1)))
          (res nil))
    (declare (fixnum mlog2n log2n h shift high len NN))
    (labels ((gprod(n ans res)
               (declare (fixnum n))
               (let ((m (ash n -1)))
                 (cond ((cl::= m 0)
                        (mpz_set_si ans (incf NN 2)))
                       ((cl::= n 2)
                        (mpz_set_si ans (cl::* (cl::incf NN 2)(cl::incf NN 2))))
                       (t (let ((b (car res))) ;b is a temporary gmpz
                            (gprod m b (cdr res)) ;set b
                               (gprod (cl::- n m) ans (cdr res)) ;set ans
                               (mpz_mul ans b ans) ;set ans
                               )))
                     ans)))

        (dotimes (i  (1+ (- mlog2n)))
          (push  (gmpz-z (alloc-gmpz)) res)) ;initialize resource

        (loop while (cl::/= h n) do
              (cl::incf shift h)
              (setf h (ash n mlog2n))
              (cl::incf mlog2n)
              (setf len high)
              (setf high (if (oddp h) h (cl::1- h)))
              (setf len (ash (cl::- high len) -1))
              (cond ((cl::> len 0)
                        (mpz_mul p p (gprod len ans res))
                        (mpz_mul r r p)
                        )))
        (mpz_mul_2exp r r shift)
        rr)))
```

5 Can we do better yet?

With considerable additional programming (about 120 lines of Lisp code) we can reformulate the computation to one based on factoring the factorial, a variation described in Luschny's report. This kind of "advanced" algorithm will interleave factorial and a prime-number sieve, and produce, in effect, the prime factor decomposition of the factorial before multiplying it out. A detailed discussion of this would extend the length of this paper substantially, though, and so we refer the interested reader to Luschny's web site, directly.

By intertwining a prime-number sieve with the computation, we can beat the split-recursive idea as embodied in programs `kg` and `gkg` also using GMP, by about 25 percent, for numbers of the size of 20,000!. Contact the author for a version of this code.

6 Conclusions

Computing large factorials is a more interesting problem than we had first thought, suggesting much more elaborate program schemas than the familiar and naive recursive or iterative approach. There are practical issues that arise on the border between ordinary and bignum arithmetic whenever one can make a choice of using a mixture of numeric types. And we haven't even looked at such issues as the influence of cache-size on performance.

All the benchmark times depend substantially on the nature and relative efficiency of the underlying system's bignum arithmetic, and that in turn can depend not only on algorithms, but in the case of GMP, on the availability of machine-specific assembler and the quality of the C compiler used, as well as the particular version of the GMP library.

Our major conclusion is that you should probably not use the "naive" factorial programs for much of anything. If you have GMP available, you can save effort and just call the library GMP routine. It is likely to be at least as good as anything else, and may, from time to time, get even better, without any effort on your part.

If you wish to write your own program, or use one of ours, you can make your own benchmarking tests on data of interest to you.

The exact ranking of the algorithms differs depending even on the CPU: it appears that on a Pentium 3, the codes in (Lisp+GMP) for `gsprfact` and `gkg` are both faster by 20 percent than the GMP 4.1.14 library routine `mpz_fac`. The differences are negligible on a Pentium 4. The new GMP 4.2 library has an improved factorial, as well as an improved multiplication which should make it perhaps 25 percent faster again, if it is based on a prime-number sieve. We hope the GMP library will continue to be among the fastest: after all, if an improved algorithm can be expressed in a page of code in Lisp or Java, it can and should be written in C for the next GMP library!

Finally, depending on circumstances, it may pay to memoize the factorial, whatever version is used.

7 Acknowledgments

Thanks for comments on an earlier draft to participants in newsgroups `sci.math.symbolic` and `comp.lang.lisp`, including Peter Luschny, in April, 2006.