A Simple Algorithm to Calculate $S(n)$

by John C. McCarthy

Introduction

This short paper first outlines an "obvious" algorithm for calculating $S(n)$ (the smallest integer $m$ such that $m!$ is divisible by $n$). Doubtless, there exist more subtle and efficient algorithms. I hope some readers will devise these and enlighten me concerning them through this journal.

This is followed by a small scale investigation of the efficiency of the algorithm.

Then there is a short discussion of a simple way of reducing the space required for storage of all $S(n)$ for ranges of $n$. The storage space required for $S(n)$ for all $n$ which my routines can handle is considered.

Heavily commented listings of an implementation of the algorithm in "C", sample output and timing data are included to help illustrate the algorithm.

The Algorithm

The algorithm is described in detail at the start of the header file "S(n).H". Together with "S(n).C", this forms all the code necessary to implement the algorithm. Note that, for the $S(n)$ function to work correctly, the function make_primes() must first be called from the main program.

The code for printing $S(n)$ and timing the routines has been omitted. These activities are both implementation specific and easily done. They are therefore left as an exercise for the interested reader.

The algorithm hinges on finding the prime factors of $n$. Improvements on how this is done will most benefit its efficiency.

To be practical, the given implementation of the algorithm only works for $0<n<2^{12}$. However, the algorithm is generally applicable to any non-null integer.

Tables of $S(n)$, constructed using the routines of "S(n).C", for the largest 2000 permitted $n$ are included. My paging routines are rather elaborate. Using them (without printing!), it took 2.4 hours to discover that 3,745,708 pages, as tightly packed as those shown, would be required to print $S(n)$ for all $0<n<2^{12}$.

Efficiency of the Algorithm

In a letter to R. Muller (about computing the Smarandache Function, July 19, 1993), Ian Parberry (editor of <SIGACT News>,
Denton, Texas) expressed that one can immediately find an algorithm that computes $S(n)$ in $O(n\log n/\log\log n)$ time ("A Brief History of the "Smarandache Function" by Dr. Constantin Dumitrescu, Department of Mathematics, University of Craiova, Romania). Disappointingly, a little analysis of the accompanying timing data on my TI85 advanced scientific calculator reveals that my algorithm is somewhat worse than this.

Trying to fit the version 2 timing data to various $O(f(n))$, I obtained the following results ($x=3355443200$ and $10(O(x+99)-O(x-100))$ is calculated for comparison with the last entry of the version 1 timing data):

<table>
<thead>
<tr>
<th>$O(f(n))$</th>
<th>Correlation</th>
<th>$O(2^{x^2}-1)$ (years)</th>
<th>$10(O(x+99)-O(x-100))$ (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>0.9928879</td>
<td>0.6092</td>
<td>8909</td>
</tr>
<tr>
<td>$O(n\log n/\log\log n)$</td>
<td>0.9944006</td>
<td>0.7906</td>
<td>11827</td>
</tr>
<tr>
<td>$O(n/n)$</td>
<td>0.9997756</td>
<td>24.2</td>
<td>469178</td>
</tr>
</tbody>
</table>

$O(n/n)$ fits the version 2 timing data best, although the time it predicts for the last entry of the version 1 timing data is almost 3 times too large. Hence, I assume the time complexity of my algorithm is a little better than $O(n/n)$.

As a rough upper limit on the time my program (on my 20MHz 368DX PC) would take to calculate $S(n)$ for all $0<n<2^{32}$, let us assume that every value of $n$ requires as much time as each $n$ in the range of the last entry of the version 1 timing data ($=159111/199/10=79.9553$ ms). In this "worst case", it would take 10.882 years. $O(n/n)$ time complexity predicts more than twice this value, which is a measure of how pessimistic it is.

I would welcome a more rigorous analysis of the time complexity of my algorithm as I presently lack the necessary expertise.

**Simple Compression of Stored $S(n)$**

Without compression, each $S(n)$ would be stored as a 32-bit (= 4 bytes) value. Hence $2^{32}$ bytes (= 16 Gigabytes) would be required to store $S(n)$ for all $0<n<2^{32}$.

This requirement can be reduced considerably if we use the high bit of each each byte of each value to indicate if it is the last byte of the value. If the bit is set it means that further byte(s) are required and if it is reset it means that the byte is the last byte of the current value. This means that only 7 bits of each byte are used to form the numerical part of the value. Assuming that, as with Intel format, the values are stored low-'byte' (actually 7 bits) first, here are some examples:

i) 127 requires seven bits and so just one byte (with high bit reset to indicate no further bytes).

ii) 16,000 requires 14 bits. So it is stored as two bytes. The
first is 0 (16,000 mod 128) + 128 (to set the high bit indicating there is more to come). The second is 125 (16,000 div 128) (with high bit reset to indicate no further bytes). This reads simply as 0 (with more to follow) + 128*125 (no more to follow).

iii) A number stored as the three bytes 57+128, 93+128 and 125+0 would similarly represent:

\[ 57 + 93*128 + 125*128*128 = 2,059,961. \]

The largest numbers that can be represented by a given number of bytes is thus as follows:

- 1 byte can code up to \(2^7-1\) = 127.
- 2 bytes can code up to \(2^{14}-1\) = 16,383.
- 3 bytes can code up to \(2^{21}-1\) = 2,097,151.
- 4 bytes can code up to \(2^{28}-1\) = 268,435,455.
- 5 bytes can code up to \(2^{35}-1\) = 34,359,738,355 (or 8 times the largest unsigned long).

For small values of \(n\), the savings are considerable (400%). However, even large \(n\) often have small \(S(n)\).

Using this technique to compress all \(S(n)\) calculated for some ranges of \(n\) (each range was also stored), I obtained the following results:

<table>
<thead>
<tr>
<th>range of n</th>
<th>compression</th>
<th>time taken (seconds)</th>
<th>size</th>
<th>size after pkzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>without</td>
<td>4.5</td>
<td>40,008</td>
<td>19,836</td>
</tr>
<tr>
<td>-10,000</td>
<td>with</td>
<td>4.7</td>
<td>15,749</td>
<td>15,267</td>
</tr>
<tr>
<td>2,147,478,648</td>
<td>without</td>
<td>827.3</td>
<td>40,008</td>
<td>33,729</td>
</tr>
<tr>
<td>-2,147,488,647</td>
<td>with</td>
<td>842.4</td>
<td>33,541</td>
<td>30,836</td>
</tr>
<tr>
<td>4,294,957,296</td>
<td>without</td>
<td>1,066.2</td>
<td>40,008</td>
<td>34,320</td>
</tr>
<tr>
<td>-4,294,967,295</td>
<td>with</td>
<td>1,085.1</td>
<td>34,330</td>
<td>31,634</td>
</tr>
</tbody>
</table>

The results indicate that this compression is a little better than pkzip's (a commercial file compression utility). Application of pkzip to a pre-compressed file also gives a slight improvement.

Assuming that the savings shown for the middle range of 10,000 \(n\) are the average of all ranges of 10,000 \(n\), using my compression together with that of pkzip would permit storage of \(S(n)\) for all \(0 < n < 2^{32}\) in about \(3.0836*2^{32} = 12.3344\) Gigabytes. So look out for sets of 19 CD-ROMs with all your favourite numbers on them!

21st November 1993
Example Implementation of A Simple Algorithm to Calculate $S(n)$, The Smarandache Function:

Because there are more people familiar with C than with C++, this module has been written entirely in C (apart from "//" style comments). The module was compiled using Borland C++ version 3.1.

For efficiency, $n$ is constrained to the limits of an unsigned long. Hence, $0 \leq n < 2^{32} - 1$ (4,294,967,295). ("^" represents exponentiation).

Although catering for $n$ of vast magnitude is possible, it imposes heavy storage and processing overheads. The range of an unsigned long therefore seems a reasonable compromise.

The algorithm depends on the most elementary properties of $S(n)$:

1) Calculate the STANDARD FORM (SF) of $n$:
   In SF: $n = +/-(p_1^{a_1})(p_2^{a_2})\ldots(p_r^{a_r})$ where $p_1, p_2, \ldots p_r$ denote the distinct prime factors of $n$ and $a_1, a_2, \ldots a_r$ are their respective multiplicities.

2) $S(n) = \max\{S(p_1^{a_1}), \ldots, S(p_r^{a_r})\}$.

3) $S(p^{a})$, where $p$ is prime, is given by:
   3.1) $a=p \implies S(p^{a}) = p^{a}$.
   3.2) $a>p \implies S(p^{a}) = x < p^{a}$. In this case, fortunately rare, $x$ is the smallest integer such that $p$ appears as a factor in the list of all integers > 1 and $\leq x$ at least $a$ times. Let the no. of times $p$ appears as a factor in the list of all integers > 1 and $\leq y$ be $f(y, p)$. Then:
   $f(y, p) = \sum\{\text{int}(y/(p^i))\}$ for $i>0$ while $y>(p^i)$.
   Hence, $x$ is the smallest integer such that $f(x, p) = a$. Note that between successive integer multiples of $p$ there are no integers which have $p$ as a factor. The trick here is to look for the largest multiple of $p$ (call it $c$), such that $f(p*c, p) = a$ (so that $x = p*c$, if $f(p*c, p) = a$, else $x = p*(c+1)$: 3.2.1) $c = a-2$ (largest possibility for $c$ since $f(p*(a-1), p) = a$ when $a>p$ (Note: $f(p*(a-1), p) = a$ is not sought for slight performance gain)).
   3.2.2) $z = f(p*c, p)$.
   3.2.3) While($z > a$):
      3.2.3.1) $d = \text{no. of times } p \text{ appears as a factor of } p*c$
      \quad = (\text{no. of times } p \text{ appears as a factor of } c) + 1.
      3.2.3.2) $c = c-1$ (next largest possibility for $c$).
      3.2.3.3) $z = z-d$ ($= f(p*c, p)$).
   3.2.4) If($z < a$), $x = p*(c+1)$.
   3.2.5) Else $x = p*c$.

To calculate the prime factors of all 32-bit $n$ requires use only of primes < $2^{16}$ (i.e. all primes expressible as an unsigned short integer). This is because any factor of $n$ remaining after division of $n$ by all its prime factors < $2^{16}$ is simply a prime. Since there are only 6542 16-bit primes, the program first creates a list of these (which only takes about 4 seconds on my 20 MHz 386DX PC) so that they never have to be recalculated, thus saving much time.

*/
```c
#define PRIMES16 6542 // The number of 16-bit primes
#define MAX_SFK 9 /* max. distinct primes in the SF of n. The smallest number with more than 9 distinct primes is the product of the 10 smallest primes (= 6,469,693,230), which is substantially more than the largest integer expressible as an unsigned long. Hence, 9 distinct primes are more than ample. */

typedef unsigned long u_long;
typedef unsigned int u_int;
typedef enum {false, true} boolean;

struct SF_struct {
    int sfk;    // no. of distinct primes
    u_long sfp[MAX_SFK]; // the distinct primes
    int sfa[MAX_SFK];    // respective multiplicities
};

extern u_int prime[PRIMES16+1]; // list of all 16-bit primes
    // plus terminating zero.

void make_primes(void); // construct list of all 16-bit primes (prime[]).
    // Must be called before calls to getSF() or S().

void getSF(u_long n, struct SF_struct *SF); // calc. SF of n and store in SF
u_long S(u_long n); // calc. S(n)
u_long Spa(u_long p, int a); // calc. S(p-a) where p is prime

int f(int x, int p); /* the number of times the prime p appears as a factor in the integers from 1 to x inclusive. This function is only called from Spa(p, a) when a>p with x=p*(a-2) (refer to item (3) of algorithm outline above). Max value of (a) occurs when p is a minimum, n is a maximum and (p-a)=n. So, (2^max(a))=max(n)=(2^32)-1. Hence max(a)<32. So, x<60 when (a) is at its max. Max value of p (and x) occurs when a=p+1 and (p-a)=max(n). So, max(p)=(max(p)+1)=(2^32)-1. The upshot is that max(p)=9 when a=10. Hence, max(x)=72. This explains why it is safe for x, p and the return value of f(x,p) to be passed as ints. */

```
Example Implementation of A Simple Algorithm to Calculate S(n), The Smarandache Function:

This is the code for the module. Refer to "S(n).h" for details.

```c
#include "S(n).h"

u_int prime[PRIMES16+1]; // allocate storage for list of all 16-bit primes
    // plus terminating zero.

void make_primes(void)
{
    u_int *pp;  // ptr to last prime so far of prime list
    u_int *tp;  // ptr to current test prime
    u_int p;   // number being tested for primality

    pp=prime;  // point to start of prime list
    *pp=2;     // set first prime to 2
    *++pp=3;   // set second prime to 3
    p=5;       // next possible prime. N.B. p is kept odd so that trial
                // division by 2 is unnecessary.

    while(true) { // infinite loop!:
        tp=prime+1; // point to first odd test prime
        whilst test prime <= sqrt(p):
            while((long)*tp)*(*tp)<=p) {
                if(!(p%*tp)) { // If current test prime divides (is factor of) p:
                    p+=2;      // try next odd number
                    if(p<*pp) { // done when p overflows:
                        *++pp=0;  // terminate list
                        return;
                    }
                    tp=prime+1; // point to first odd test prime
                }
                else ++tp;   // Else point to next test prime
            }
        if(p<*pp) { // no prime <= sqrt(p divides p so p must be prime:
            *++pp=p;   // so store it next in the list
            p+=2;     // try next odd number
            if(p<*pp) { // done when p overflows:
                *++pp=0;  // terminate list
                return;
            }
        }
    }
}```
S(n).C

void getSF(u_long n, struct SF_struct *SF) {
    u_int *pp; // ptr to current prime
    u_long r; // 'residue' of n remaining for factoring

    SF->sfk=0; // no. of distinct prime factors discovered
    r=n;
    pp=prime; // point to start of prime list

    // whilst current prime <= \sqrt{r} and prime list not exhausted:
    while((long) *pp)*(*pp)<=r && *pp) {
        if(! (r%*pp) { // if current prime is a factor of r:
            SF->sfp[SF->sfkJ=*pp; // store current prime as next prime of SF
            SF->sfa[SF->sfkJ]=1; // set its multiplicity to 1
            r/=*pp; // 'divide out' current prime
            while(! (r%*pp)) { // while current prime factors r:
                SF->sfa[SF->sfkJ]++; // increment multiplicity
                r/=*pp; // 'divide out' current prime
            }
            SF->sfk++; // increment count of distinct prime factors
        }
        ++pp; // next prime
    }

    if (r>1) { // If n contains prime > 2^16:
        SF->sfp[SF->sfkJ]=r; // store it as last prime of SF
        SF->sfa[SF->sfkJ]=1; // set its multiplicity to 1
        SF->sfk++; // increment count of distinct prime factors
    }
}
S(n).C

u_long S(u_long n)
{
    struct SF_struct SF; // to store SF of n
    int sfi; // index of current term of SF of n
    u_long Sn; // current guess at S(n)
    u_long x; // S(current term of SF of n) where it might exceed
                // current value of Sn.

    if(n==1) return 0; // special case

    getSF(n, &SF); // calc. and store SF of n

    // First guess at S(n) is S(p^a), where p is the largest prime in the SF
    // of n and a is its multiplicity. This pre-empts the calculation of S(p^a)
    // for the remaining terms where, as is likely, p^a for these terms is <=
    // this initial guess (since S(p^a) <= p^a always):
    sfi=SF.sfk-1;
    Sn=Spa(SF.sfp[sfi],SF.sfa[sfi]);

    while(sfi>0) { // while more term(s):
        sfi--; // next term
        if(SF.sfp[sfi]*SF.sfa[sfi]>Sn) { // if this term may have larger S(p^a):
            x=Spa(SF.sfp[sfi],SF.sfa[sfi]); // calc. it
            if(x>Sn) Sn=x; // if new max., update Sn with it
        }
    }
    return Sn; // That's all folks!
}

u_long Spa(u_long p, int a)
{
    // Refer to item 3) of the algorithm description in S(n).h.
    int c; // largest multiple of p such that f(p*c, p) <= a (eventually!)
    int z; // f(p*c, p)
    int m; // used to calc. no. of times p appears as factor of c

    if(a<=p) return p*a;

    c=a-2;
    z=f(p*c, p);
    while(z>a) {
        // d in items 3.2.3.1 and 3.2.3.3) of algorithm description is implicit
        // here:
        z--; //
        m=c--; //
        while(! (m%p)) { // while p divides m:
            z--; // 'divide out' factor of p from m
            m/=p;
        }
    }
    if(z<a) return P*(C+1);
    else return P*C;
}
int f(int x, int p)
{
    int k=0; // count of appearance of prime p as a factor in the integers
             // from 1 to x.
    int xdp; // successive divisions of x by p
    xdp=x/p;
    while(xdp>0) {
        k+=xdp;
        xdp/=p;
    }
    return k;
}
Timing of $S(n)$.c Module for Calculation of Smarandache Function, version 1

Time taken to calculate $S(n)$ depends on how easy it is to factor $n$. Less time is required if $n$ has "small" prime factors. So, in the following table, the values of $n$ shown are the mid-points of ranges $(n-99 \text{ thru } n+99)$. Times shown are for calculating $S(n)$ for all integers in each range 10 times over:

<table>
<thead>
<tr>
<th>$n$</th>
<th>time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>268</td>
</tr>
<tr>
<td>200</td>
<td>308</td>
</tr>
<tr>
<td>400</td>
<td>345</td>
</tr>
<tr>
<td>800</td>
<td>387</td>
</tr>
<tr>
<td>1600</td>
<td>432</td>
</tr>
<tr>
<td>3200</td>
<td>490</td>
</tr>
<tr>
<td>6400</td>
<td>571</td>
</tr>
<tr>
<td>12800</td>
<td>661</td>
</tr>
<tr>
<td>25600</td>
<td>766</td>
</tr>
<tr>
<td>51200</td>
<td>919</td>
</tr>
<tr>
<td>102400</td>
<td>2450</td>
</tr>
<tr>
<td>204800</td>
<td>4036</td>
</tr>
<tr>
<td>409600</td>
<td>5670</td>
</tr>
<tr>
<td>819200</td>
<td>7977</td>
</tr>
<tr>
<td>1638400</td>
<td>10423</td>
</tr>
<tr>
<td>3276800</td>
<td>13004</td>
</tr>
<tr>
<td>6553600</td>
<td>16302</td>
</tr>
<tr>
<td>13107200</td>
<td>23438</td>
</tr>
<tr>
<td>26214400</td>
<td>29642</td>
</tr>
<tr>
<td>52428800</td>
<td>37011</td>
</tr>
<tr>
<td>104857600</td>
<td>50330</td>
</tr>
<tr>
<td>209715200</td>
<td>62363</td>
</tr>
<tr>
<td>419430400</td>
<td>77888</td>
</tr>
<tr>
<td>838860800</td>
<td>108179</td>
</tr>
<tr>
<td>1677721600</td>
<td>158480</td>
</tr>
<tr>
<td>3355443200</td>
<td>159111</td>
</tr>
</tbody>
</table>
Timing of $S(n).c$ Module for Calculation of Smarandache Function, version 2

"Time to n" is the time taken to calculate $S(n)$ for all $n \leq$ that shown. 
"Time add." is the time taken to calculate $S(n)$ for all $n >$ previous $n$ and <= current $n$. All times are in milliseconds (as per version 1):

<table>
<thead>
<tr>
<th>$n$</th>
<th>Time to n</th>
<th>Time add.</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>18223</td>
<td>18223</td>
</tr>
<tr>
<td>100000</td>
<td>66763</td>
<td>48540</td>
</tr>
<tr>
<td>150000</td>
<td>139191</td>
<td>72428</td>
</tr>
<tr>
<td>200000</td>
<td>229634</td>
<td>90443</td>
</tr>
<tr>
<td>250000</td>
<td>335252</td>
<td>105618</td>
</tr>
<tr>
<td>300000</td>
<td>452539</td>
<td>117287</td>
</tr>
<tr>
<td>350000</td>
<td>579419</td>
<td>126880</td>
</tr>
<tr>
<td>400000</td>
<td>715146</td>
<td>135727</td>
</tr>
<tr>
<td>450000</td>
<td>859963</td>
<td>144816</td>
</tr>
<tr>
<td>500000</td>
<td>1012335</td>
<td>152372</td>
</tr>
<tr>
<td>550000</td>
<td>1171221</td>
<td>158886</td>
</tr>
<tr>
<td>600000</td>
<td>1336899</td>
<td>165678</td>
</tr>
<tr>
<td>650000</td>
<td>1508825</td>
<td>171927</td>
</tr>
<tr>
<td>700000</td>
<td>1686808</td>
<td>177983</td>
</tr>
<tr>
<td>750000</td>
<td>1870023</td>
<td>183215</td>
</tr>
<tr>
<td>800000</td>
<td>2058983</td>
<td>188961</td>
</tr>
<tr>
<td>850000</td>
<td>2252457</td>
<td>193473</td>
</tr>
<tr>
<td>900000</td>
<td>2450892</td>
<td>198435</td>
</tr>
<tr>
<td>950000</td>
<td>2653620</td>
<td>202728</td>
</tr>
<tr>
<td>1000000</td>
<td>2860734</td>
<td>207115</td>
</tr>
<tr>
<td>1050000</td>
<td>3072049</td>
<td>211314</td>
</tr>
<tr>
<td>1100000</td>
<td>3288502</td>
<td>216454</td>
</tr>
<tr>
<td>1150000</td>
<td>3509106</td>
<td>220603</td>
</tr>
<tr>
<td>1200000</td>
<td>3733965</td>
<td>224860</td>
</tr>
<tr>
<td>1250000</td>
<td>3962171</td>
<td>228206</td>
</tr>
<tr>
<td>1300000</td>
<td>4194158</td>
<td>231987</td>
</tr>
<tr>
<td>1350000</td>
<td>4429331</td>
<td>235173</td>
</tr>
<tr>
<td>1400000</td>
<td>4668560</td>
<td>239229</td>
</tr>
<tr>
<td>1450000</td>
<td>4910513</td>
<td>241953</td>
</tr>
<tr>
<td>1500000</td>
<td>5155601</td>
<td>245088</td>
</tr>
<tr>
<td>1550000</td>
<td>5404652</td>
<td>249051</td>
</tr>
<tr>
<td>1600000</td>
<td>5656512</td>
<td>251859</td>
</tr>
<tr>
<td>1650000</td>
<td>5911306</td>
<td>254794</td>
</tr>
<tr>
<td>1700000</td>
<td>6169686</td>
<td>258380</td>
</tr>
<tr>
<td>1750000</td>
<td>6431883</td>
<td>261697</td>
</tr>
<tr>
<td>1800000</td>
<td>6696172</td>
<td>264789</td>
</tr>
<tr>
<td>1850000</td>
<td>6963206</td>
<td>267034</td>
</tr>
<tr>
<td>1900000</td>
<td>7232974</td>
<td>269768</td>
</tr>
<tr>
<td>1950000</td>
<td>7505412</td>
<td>272438</td>
</tr>
<tr>
<td>2000000</td>
<td>7779763</td>
<td>274351</td>
</tr>
<tr>
<td>2050000</td>
<td>8056579</td>
<td>276816</td>
</tr>
<tr>
<td>2100000</td>
<td>8336442</td>
<td>279863</td>
</tr>
<tr>
<td>2150000</td>
<td>8620053</td>
<td>283611</td>
</tr>
<tr>
<td>2200000</td>
<td>8905641</td>
<td>285588</td>
</tr>
<tr>
<td>2250000</td>
<td>9194727</td>
<td>289086</td>
</tr>
<tr>
<td>2300000</td>
<td>9486449</td>
<td>291722</td>
</tr>
<tr>
<td>2350000</td>
<td>9780105</td>
<td>293655</td>
</tr>
<tr>
<td>2400000</td>
<td>10076920</td>
<td>296815</td>
</tr>
<tr>
<td>2450000</td>
<td>10375202</td>
<td>298282</td>
</tr>
<tr>
<td>2500000</td>
<td>10676383</td>
<td>301181</td>
</tr>
</tbody>
</table>

John McCarthy
17 Mount Street
Mansfield
Notts.
NG19 7AT
United Kingdom