ALGORITHMS FOR DETERMINING INTEGER COMPLEXITY

J. ARIAS DE REYNA AND J. VAN DE LUNE

INTRODUCTION.

The complexity $\|n\|$ of a natural number n is defined as the least number of 1's needed to express the positive integer n using only the two operations $+$ and $\times$ and parentheses. At present there is a growing literature about this topic. For details we refer to [9], [4], [10], [2], [1], [6].

We present three algorithms to compute the complexity $\|n\|$ of all natural numbers $n \leq N$. The first of them is a brute force algorithm, computing all these complexities in time $\mathcal{O}(N^2)$ and space $\mathcal{O}(N \log^2 N)$. The main problem of this algorithm is the time needed for the computation. In 2008 there appeared three independent solutions to this problem: [11], [7], [3]. All three are very similar. Only [11] gives an estimation of the performance of its algorithm, proving that the algorithm computes the complexities in time $\mathcal{O}(N^{1+\beta})$, where $1 + \beta = \log 3/\log 2 \approx 1.584963$. The other two algorithms, presented in [7] and [3], were very similar but both superior to the one in [11]. In Section 2 we present a version of these algorithms and in Section 4 it is shown that they run in time $\mathcal{O}(N^\alpha)$ and space $\mathcal{O}(N \log \log N)$. (Here $\alpha = 1.230175$).

The authors of the present paper, at the moment of sending [3] to a journal (Oct. 2008) saw Martin N. Fuller's program [7] posted in the OEIS. Recognizing the similarity of our own approach in [3] we contacted Fuller and proposed to publish our results jointly. This started a very active collaboration, improving the results in [7] and [3]. The main problem with the algorithm presented in Section 2 concerned the space requirement. In [7] Fuller has given the idea of how to reduce the space requirements. He then programmed this improved algorithm and used it to compute the complexities for all $n \leq 10^{11}$.

For some reason (unknown to us) there was an interruption of this collaboration after which we could not contact Fuller any more. The present paper is the result of the above collaboration and we both think that Fuller should have been the main author of this paper. Of course he is not responsible for any errors contained in it.

In Section 2 we present the algorithm of [7] and [3]. The main advantage of this algorithm with respect to that in [11] is the definition of kMax in Section 2.7. This explains the difference in performance from $\mathcal{O}(N^{1+\beta})$ to $\mathcal{O}(N^\alpha)$.

In Section 3 we present a detailed description of this space-improved algorithm of Fuller and in Section 5 we prove that it runs in time $\mathcal{O}(N^\alpha)$ and space $\mathcal{O}(N^{(1+\beta)/2} \log \log N)$, where $\alpha = 1.230175$ and $(1 + \beta)/2 \approx 0.792481$.

We have delayed the publication of the present paper in the hope of again getting in contact with M. Fuller. In the mean time it has been published [6] where the computation of the complexities of all $n \leq 10^{12}$ is announced. In [6] it is said that they have used an algorithm inspired by the description in [7] of the space-improved algorithm of Fuller, but do not give a detailed description of their algorithm. The results proved in this paper about the performance of the algorithms are not trivial and are still missing.

There are many known results about the complexity of natural numbers. We will use the inequality $\|n\| \leq 3 \log_2 n$ for $n > 1$, which appeared first in [4].

Since we try to explain two algorithms, we have tried to concentrate on explaining things to human beings how to proceed instead of trying to explain to a computer. This is the definition of *Literate programming*. We follow the recommendations of Knuth in [8].

The full power of these programs will be apparent when implemented in a language such as C or Pascal, but to explain the algorithm we have preferred Python as a language. In this way we can concentrate on the computations needed and not on the manipulation of data.

To one not used to Python it is a very simple language, the main difficulty possibly being that the scopes of the for, while, etc. loops are indicated by indentations. We also note that for integers a and b Python defines a/b as $\lfloor \frac{a}{b} \rfloor$.

1. Brute force algorithm.

This is the algorithm used in [2] to compute the values of $\|n\|$ for $1 \leq n \leq \text{nMax}$ with $\text{nMax} = 200\,000$.

It is based on the fact that $\|1\| = 1$, and (also see Section 2.1)

$$(1) \qquad \|n\| = \min_{\substack{a,b < n \in \mathbf{N} \\ a+b=n \text{ or } ab=n}} \|a\| + \|b\|.$$

```
<1 Brute force algorithm>  ≡
nMax = 20000
Compl={}
s = range(nMax+1)
divisors = [[1] for n in s]
for k in range(2, nMax+1):
    j=1
    m = k
    while m <= nMax:
        divisors[m].append(k)
        m = m+k
Compl[1]=1
for n in range(2, nMax+1):
    S=nMax+1
    for k in range(1,1+n/2):
        a = Compl[k]+Compl[n-k]
        if a < S:
            S=a
    div_n = divisors[n]
    tau_n = len(div_n)
```

```
P = nMax+1
for k in range(1,tau_n-1):
    d = div_n[k]
    a = Compl[d]+Compl[n/d]
    if a < P:
        P = a
Compl[n] = min(S,P)
```

We will not explain this simple minded algorithm. It is not very useful. At the end Compl[n] will give us the value of $\|n\|$ for $1 \leq n \leq$ nMax. Observe that for each n we compute

$$S := \min_{\substack{a,b<n\in\mathbf{N} \\ a+b=n}} \|a\| + \|b\| \quad \text{and} \quad P := \min_{\substack{a,b<n\in\mathbf{N} \\ ab=n}} \|a\| + \|b\|.$$

The complexity of n is then the minimum of these two quantities: $\|n\| = \min(S, P)$.

It is easy to prove the following

Proposition 1. *The above brute force algorithm computes the values of $\|n\|$ for $1 \leq n \leq N$ in time $\mathcal{O}(N^2)$ and space $\mathcal{O}(N \log^2 N)$.*

To store the complexities we need only $\mathcal{O}(N \log \log N)$ space. But our program stores the divisors of all $1 \leq n \leq N$ requiring $\mathcal{O}(N \log^2 N)$ space.

2. TIME IMPROVED ALGORITHM.

2.1. **Idea of the algorithm.** The main disadvantage of the brute force algorithm is its running time $\mathcal{O}(N^2)$. This time is spent mainly on checking all the instances of $\|n - k\| + \|k\|$ for $1 \leq k \leq n/2$. We will improve the time by finding a suitable kMax such that

$$(2) \qquad S = \min_{1 \leq k \leq n/2} \|k\| + \|n - k\| = \min_{1 \leq k \leq \text{kMax}} \|k\| + \|n - k\|.$$

In this new program we will define a function: 'complexity($\cdot$)'. Given an integer nMax, the output of $B = $ complexity(nMax) will be a list B containing the values of the complexities of the numbers $1 \leq n \leq$ nMax. More precisely we will have $B[n] = \|n\|$.

2.2. **Plan of the second algorithm.** After some initializations the program will consist of only one main loop. In this loop we will compute successively the complexities of the natural numbers $2 \leq n \leq$ nMax. The first part of this computation will consist of the computation of an adequate kMax for the corresponding n. The computation of kMax uses an auxiliary function E.

We import the module *math* because we are using the mathematical function log in our program.

⟨2.2 Time improved algorithm ⟩

```
<2.3 definition of E>
from math import *
def complexity(nMax):
    <2.4 Initializing Compl>
    <2.5 Main Loop>
    return Compl
```

2.3. **Definition of** E**.** For each natural number k let $E(k)$ be the largest number with complexity k (see [1]). Then $E(1) = 1$, and for $j \geq 1$, $E(3j) = 3^j$, $E(3j + 1) = 4 \cdot 3^{j-1}$, $E(3j - 1) = 2 \cdot 3^{j-1}$. We extend this definition by $E(0) := 1$. In this way E coincides with the sequence A000792 in the OEIS [13]. We have

$$3^{(n-1)/3} \leq E(n) \leq 3^{n/3} \qquad n \geq 0. \tag{3}$$

Note that E is a non decreasing function. The following algorithm computes E:

```
<2.3 definition of E>
def E(n):
    if n == 0:
        return 1
    result = 1
    while n > 4:
        result *= 3
        n -= 3
    return result * n
```

For later application we state the following easily proved fact.

Proposition 2. *The cost of computing $E(n)$ is $\mathcal{O}(n)$ operations and $\mathcal{O}(n)$ space.*

Proof. Observe that to write $3^{n/3}$ we need $\mathcal{O}(n)$ space. $\qquad\square$

2.4. **Initializing** Compl**.** The complexity function outputs a list $\text{Compl}[\cdot]$. We first fill this list with a value which is an upper bound for all the true complexities. This upper bound of $\text{Compl}[n]$ will be denoted by cMax. Since $\|n\| \leq \frac{3}{\log 2} \log n$, we may take this bound equal to $\lfloor 3 \log \text{nMax} / \log 2 \rfloor$. We take a unit more because in case nMax is a power of 2 we are near the singularity of the $\lfloor \cdot \rfloor$ function.

We also initialize $\text{Compl}[1] = 1$.

```
<2.4 Initializing Compl>  ≡
cMax = int(3*log(nMax)/log(2))+1
Compl =[cMax for n in range(0,nMax+1)]
Compl[1] = 1
```

2.5. **Main Loop.** Now we have the correct value of $\text{Compl}[1]$. Dynamically we compute $\text{Compl}[n]$ for $2 \leq n \leq \text{nMax}$. The pertinent loop consists of the following parts:

```
<2.5 Main Loop>  ≡
for n in range(2,nMax+1):
    <2.6 usual best value>
    <2.7 computing kMax>
    <2.8 testing the sums>
    <2.9 testing the products>
```

2.6. **Usual best value.** When computing $\|n\|$ we already have $\mathrm{Compl}[j] = \|j\|$ for all $1 \le j \le n-1$, and for $k \ge n$ we have $\mathrm{Compl}[k] \ge \|k\|$. We will proceed to check all sums and products to replace $\mathrm{Compl}[k]$ by better values, so that eventually we will obtain the correct values.

We call $\|n-1\|+1$ the *usual best value* of $\|n\|$. At this stage we simply replace $\mathrm{Compl}[n]$ by this bound if needed.

```
<2.6 Usual best value>  ≡
a = Compl[n-1]+1
if a < Compl[n]:
    Compl[n] = a
```

2.7. **Computing** kMax**.** To compute kMax we use a bound s with $\|n\| \le s$. Here we always take $s = \|n-1\|+1$.

Here is our procedure to compute kMax. Subsequently we will prove that it has the desired properties.

```
<2.7 Computing kMax>  ≡
target =Compl[n-1]
t = target/2
while E(t)+E(target-t) < n:
    t = t-1
kMax = E(t)
```

Proposition 3. *For $n \ge 3$ the above procedure stops in finite time giving a value*

$$(4) \qquad 1 \le \mathrm{kMax} \le \frac{n}{2}\left(1 - \sqrt{1 - \frac{4}{n^2}3^{\|n-1\|/3}}\right).$$

Proof. Observe that in the second line we obtain $t = \lfloor \|n-1\|/2 \rfloor$ (by the convention of Python $m/2$ returns the integer part of the fraction $\frac{m}{2}$). Therefore, for $n \ge 3$ after the second line t will be a natural number or 0. In each while-loop of the while t is replaced by $t-1$, but if we arrive at $t = 0$ we will reach the condition of the while loop

$$E(0) + E(\|n-1\| - 0) \ge 1 + (n-1) = n$$

and the program will stop with $t \ge 0$. Hence at finite time.

For each natural number t we will have

$$E(t) + E(\|n-1\| - t) \le 3^{t/3} + 3^{(\|n-1\|-t)/3} < n$$

whenever $x^2 - nx + 3^{\|n-1\|/3} < 0$ with $x = 3^{t/3}$. For $n \ge 29$, the roots x_1, x_2 of this polynomial in x are real, because its discriminant is $n^2 - 4 \cdot 3^{\|n-1\|/3} > 0$. In fact

$$2 \cdot 3^{\|n-1\|/6} \le 2\exp\left(\frac{\log 3}{6}\frac{3}{\log 2}\log n\right) < n, \qquad n \ge 29.$$

We also have $0 < x_1 < x_2$ because $x_1 x_2 > 0$, so that x_1 and x_2 have the same sign and since $x_1 + x_2 = n$ they are positive.

The initial value of $t = \lfloor \|n - 1\|/2 \rfloor$, so that $t = \|n - 1\|/2 - \varepsilon$ with $\varepsilon = 0$ or $1/2$. For $n \geq 31$, we will have

$$3^{t/3} + 3^{(\|n-1\|-t)/3} = 3^{\|n-1\|/6-\varepsilon/3} + 3^{\|n-1\|/6+\varepsilon/3} \leq (3^{-\varepsilon/3} + 3^{\varepsilon/3}) \exp\left(\frac{\log 3}{\log 4} \log n\right) < n.$$

It follows that this initial value of t satisfies $x_1 < 3^{t/3} < x_2$ when $n \geq 31$. While $3^{t/3} > x_1$ we will get $E(t) + E(\|n - 1\| - t) < n$, so that we end the while loop with a value $t = t_e$ such that $3^{t_e} \leq x_1$. Then

$$(5) \qquad \text{kMax} = E(t_e) \leq 3^{t_e/3} \leq x_1 = \frac{n}{2}\left(1 - \sqrt{1 - 4 \cdot 3^{\|n-1\|/3}/n^2}\right).$$

While computing the needed complexities one may check that (4) is also true for $3 \leq n \leq 31$. $\qquad \square$

Corollary 4. *For $n \geq 2$ we have* $\text{kMax} \leq 2n^\beta$ *where* $\beta = \frac{\log 3}{\log 2} - 1 \approx 0.584963$.

Proof. We have $3^{\|n-1\|/3} < 3^{\log n/\log 2} = n^{1+\beta}$ and $4n^{\beta-1} < 1$ for $n \geq 29$, so that in this case

$$\sqrt{1 - \frac{4}{n^2}3^{\|n-1\|/3}} > \sqrt{1 - 4n^{\beta-1}} > 1 - 4n^{\beta-1}.$$

Therefore, by Proposition 3 we get

$$\text{kMax} \leq \frac{n}{2}\left(1 - \sqrt{1 - \frac{4}{n^2}3^{\|n-1\|/3}}\right) < 2n^\beta.$$

For $2 \leq n \leq 29$ we may check directly that $\text{kMax} < 2n^\beta$ (for $2 \leq n \leq 50$ we have $\text{kMax} = 1$ except for $n = 24$ and $n = 48$ for which $\text{kMax} = 2$). $\qquad \square$

We have the following main result about kMax.

Proposition 5. *For $n \geq 2$ we have*

$$S := \min_{1 \leq k \leq n/2} \|k\| + \|n - k\| = \min_{1 \leq k \leq \text{kMax}} \|k\| + \|n - k\|.$$

Proof. If not there would be a k_0 with $1 \leq k_0 \leq n/2$ and

$$(6) \qquad \|k_0\| + \|n - k_0\| < \min_{1 \leq k \leq \text{kMax}} \|k\| + \|n - k\|.$$

In particular $\|k_0\| + \|n - k_0\| < 1 + \|n - 1\|$, so that $\|k_0\| + \|n - k_0\| \leq \|n - 1\|$. One of the two numbers k_0 and $n - k_0$, let us call it u, would satisfy $\|u\| \leq \|n - 1\|/2$. Therefore, $\|u\| \leq t_0$, the initial value of t in our procedure to get kMax, and

$$\|u\| + \|n - u\| \leq \|n - 1\|.$$

We will also have $\|u\| \leq t_e$ the final value of t so that, by definition $\text{kMax} = E(t_e)$. In the other case we would have $t_e < \|u\| \leq t_0$ so that $t' := \|u\|$ would be one of the values of t in the procedure (not the last) and we would therefore have

$$E(t') + E(\|n - 1\| - t') < n.$$

From this we get a contradiction

$$n = u + (n - u) \leq E(\|u\|) + E(\|n - u\|) \leq E(t') + E(\|n - 1\| - t') < n.$$

Therefore, $u \leq E(\|u\|) \leq E(t_e) = \mathrm{kMax}$, so that by (6) we get the contradiction

$$\|u\| + \|n - u\| = \|k_0\| + \|n - k_0\| < \min_{1 \leq k \leq \mathrm{kMax}} \|k\| + \|n - k\| \leq \|u\| + \|n - u\|.$$

$\square$

2.8. Testing the sums. By Proposition 5 we only have to check $\|k\| + \|n - k\| <$ Compl $[n]$ for $1 \leq k \leq \mathrm{kMax}$. In *2.6 Usual best value* we have checked $k = 1$.

We need not check $k = 2$, $k = 3$, ..., $k = 5$. The first value we will check is $k = 6$. We explain: In [1] it is shown that the least value b such that

$$S := \inf_{1 \leq k \leq n} \|k\| + \|n - k\| = \|b\| + \|n - b\|$$

is a *solid number*, where b is a solid number if $b = u + v$ implies $\|b\| < \|u\| + \|v\|$.

It follows that we only need to test $\|k\| + \|n - k\| <$ Compl $[n]$ when k is a solid number. But the sequence of solid numbers starts with 1, 6, 8, ... Hence we arrive at the following algorithm to check the sums:

```
<2.8 Testing the sums>  ≡
for m in range(6, kMax+1):
    sumvalue = Compl[m]+Compl[n-m]
    if sumvalue < Compl[n]:
        Compl[n] = sumvalue
```

2.9. Testing the products. Here we consider all the multiples $k \cdot n$ of n and substitute the value of Compl $[k \cdot n]$ if necessary. Therefore, when we arrive at the case n all its divisors will have been checked, so that we will have the correct value of Compl $[n]$.

```
<2.9 Testing the products>  ≡
for k in range(2, min(n, nMax/n)+1):
    prodvalue = Compl[k]+Compl[n]
    if prodvalue < Compl[k*n]:
        Compl[k*n] = prodvalue
```

3. Fuller algorithm.

In practice the main limitation of the algorithm presented in Section 1 is the space requirement $\mathcal{O}(N \log \log N)$. We present here an idea of Fuller, partially expressed in [7] to overcome this difficulty.

The output of the new program is different from that of Section 1. Now we will compute successively the complexities of all numbers $n \leq N$ but we do not store all these values. So the output of the program will consist of several statistics about the complexities of the numbers $1 \leq n \leq N$. These statistics, which may change from run to run, will be computed simultaneously with the complexities.

3.1. **The basic idea.** The algorithm will store the minimal data needed to calculate $\|n\|$

- $\|a\|$ for $a \mid n$.
- $\|k\|$ for $k \leq \mathrm{kMax}$.
- $\|n - k\|$ for $k \leq \mathrm{kMax}$.

Small divisors and small summands of n will be stored in a *fixed block*, containing $\|n\|$ up to some fixed upper limit H. Recent calculations will be stored in a *running block* around n which is used to retrieve $\|n - k\|$. Large divisors $\|n/2\|$, $\|n/3\|$, etc. will be calculated and stored in the same way as n, using *additional running blocks* and sharing the same fixed block.

The program will have several parameters: N (or nMax in the program) will be the limit to which we compute the complexities. We will have a fixed block where we will store the complexities for $1 \leq n \leq H$. Thus H is the length of the fixed block. We will have a number of running blocks B_j all of them of length $L = 2\ell$. B_1 will be the main running block, where we will store the last computed complexities. The other running blocks B_2, B_3, ... will contain the complexities of the large divisors $n/2$, $n/3$, ... of the numbers immediately following B_1.

All running blocks B_j (for $1 \leq j \leq N/H$), each of length L will contain $\|n\|$ for the range $H_j - L < n \leq H_j$, where H_j will start at H and then move in steps of size ℓ to finish at or just above N/j.

For the computation to run smoothly we need that these parameters satisfy

$$(7) \qquad \mathrm{kMax} < \ell, \quad N \leq \ell H, \quad \ell \mid H, \quad \ell \mid N, \quad H \geq \sqrt{LN}.$$

The algorithm was used by Fuller in 2009 to compute n up to $N = 10^{11}$ using just over 10^9 bytes of memory. The running time was 106 hours using 1 processor of a 2.8 GHz dual core PC. The parameters used for this run were $N = 10^{11}$, $H = 10^9$, $L = 10^6$. Sieving was used to find potential factors $2 \leq a \leq b$, $H_j - \ell < ab \leq H_j$. Thus there was a performance gain by using a large L. The program runs roughly $1/3$ of the speed when using $L = 10^5$ instead of $L = 10^6$.

3.2. **General description.** Roughly the program executed the following steps:

1. Calculate $\|n\|$ for the fixed block B_0, using the earlier algorithm of Section 2.

2. Copy from B_0 into B_1 for the range $H - L < n \leq H$. Set $H_1 = H$.

3. Iterate the following steps while $H_1 < N$:

   a. Set $h = H_1 + \ell$.

   b. In descending order $\left\lceil \frac{h}{H} \right\rceil > j \geq 1$:

      i. If B_j has not been initialized, copy it from B_0 and set $H_j = H$.

      ii. If $H_j < \ell\lceil h/j\ell\rceil$, increase it by ℓ. Fill the bottom half of the new range with the values of the top half of the old range. Calculate new values as shown in Section 3.8.

Here we have the scheme of Fuller's program

```
<3.2 Fuller Program>  ≡
<3.9 Some definitions>
<3.3 setting parameters>
<3.4 initialize B[0]>
<3.5 creating the first running block>
<3.6 definition of the function Shift>
<3.8 definition of the function CalculateRunningBlock>
<3.7 Main Loop>
```

3.3. **Setting parameters.** As we have said we fix the three parameters N or nMax, H the length of the fixed block, and ℓ or step being the size of the unit shift for the running blocks. This is also half of the length L of the running blocks. We use a function to check that the parameters satisfy the conditions (7). The program prints the result of this check. Bad parameters may cause an index error.

```
<3.3 Setting parameters>  ≡
<3.3.1 def of checkparameters>
nMax = 1000000
H= 200000
step = 10000
L = 2*step
test = checkparameters(nMax,H,step)
if test:
    print 'GOOD PARAMETERS'
else:
    print 'NOT GOOD PARAMETERS'
```

3.3.1. *Checking parameters.* The function 'checkparameters( )' is defined. It checks whether our parameters satisfy the inequalities (7). We apply Corollary 4 to check whether kMax $\leq \ell$.

```
<3.3.1 def of checkparameters>  ≡
def checkparameters(nMax,H,step):
    test = True
    beta = 0.584962501
    if step < 2*nMax**beta:
        test = False
    if nMax > step*H:
        test = False
    if step*(H/step) != H:
        test = False
    if step*(nMax/step) != nMax:
        test = False
    if H < sqrt(L*nMax):
        test = False
    return test
```

3.4. **Initialize B_0.** The fixed block B_0 and the running blocks B_j are members of a list B. Here we simply compute the complexities contained in the fixed block (using the function defined in Section 1) and put them as the first term in the list. The successive terms of this list will be the running blocks. Hence we will have $B[0][n] = \|n\|$ for $1 \le n \le H$.

We also initialize Heads. This will be a list with $(H_j)_{j=0}^r$ as elements, where the fixed block B_0 will contain the complexities of n for $1 \le n \le H = H_0$, and B_j, when initialized, will be the list formed by the complexities of n for $H_j - L < n \le H_j$. Hence we will have $B[j][n] = \|n + H_j - L + 1\|$ for $0 \le n \le L - 1$.

```
<3.4 Initialize B₀>  ≡
B = [complexity(H)]
Heads=[H]
```

3.5. **Creating the first running block.** All running blocks B_j with $j \ge 1$ are created with $H_j = H$. Later they will be shifted by steps of length ℓ to finish at or just above N/j.

```
<3.5 Creating the first running block>  ≡
<3.5.1 definition of CreateRunningBlock>
CreateRunningBlock(B,Heads)
```

3.5.1. *Definition of CreateRunningBlock.* The definition of this function is very simple. We simply copy the last half of the fixed block and put it as running block. We also have to update the Heads list.

```
<3.5.1 definition of CreateRunningBlock>  ≡
def CreateRunningBlock(B,Heads):
    RB = B[0][H-L+1:H+1]
    B.append(RB)
    Heads.append(H)
```

3.6. **Definition of the function Shift.** The running block B_j contains the complexities of n for $H_j - 2\ell < n \le H_j$. While running the program this block is shifted ℓ units to the right. This is done in two steps, first the function Shift simply puts $H_j = H_j + \ell$, and copies the second half of the block B_j into the first half of this block and initializes the new values to any upper bound for $\|n\|$. Subsequently the function 'ComputeRunningBlock' will compute the new complexities $\|n\|$ for $H_j < n \le H_j + \ell$.

```
<3.6 definition of the function Shift>  ≡
def Shift(B,j):
    Heads[j] = Heads[j] + step
    B[j] = B[j] + [255 for n in range(0,step)]
    B[j] = B[j][step:3*step]
```

The most delicate point of the program will be to show that the function ComputeRunningBlock can complete its task. But we may assume that in some way this has been done and analyze the Main Loop. This allows us to have a picture of which running blocks have been initialized at a given point in the program and which values the H_j will have.

We have initialized the new values of B_j to 255. We need that this value is an upper bound of $\|n\|$ for all $n \leq$ nMax. Since $\|n\| \leq \frac{3}{\log 2} \log n$ we may safely take 255 for nMax $\leq 3.86 \times 10^{25}$. The value 255 is useful because in this way we may take B_j as a byte array.

3.7. Main Loop.

The main computation is that of the first running block B_1, while H_1 will indicate where we are in the task of computing all the complexities up to nMax. Each run of the main loop consists essentially of a shift of this running block from H_1 to $H_1 + \ell$. Hence we have to compute $\|n\|$ for $H_1 < n \leq H_1 + \ell$. For $n = ab$ we must read $\|a\|$ and $\|b\|$. This is the purpose of the other running blocks and the fixed block. For small a, the running block B_a must contain the values of $\|b\|$. The value of $\|a\|$ will be read from the fixed block. For $n = a + b$ with $a <$ kMax we read $\|a\|$ from the fixed block and $\|b\|$ from the running block B_1.

It follows that we have to shift B_a, if needed, before computing B_1. To update B_a we follow the same procedure as to update B_1. Hence we proceed to update the B_a in descending order of a. We also have to initialize the needed running blocks.

Here is the algorithm for the main loop. We will show that the necessary readings can easily be performed.

```
<3.7 Main Loop>  ≡
while Heads[1] < nMax:
    h = min(Heads[1]+step,nMax)
    for j in range(ceiling(h,H)-1,0, -1):
        newH = step*ceiling(h,(j*step))
        if notInitialized(j)and(newH > H):
            CreateRunningBlock(B,Heads)
        if Heads[j] < newH:
            Shift(B,j)
            CalculateRunningBlock(j,newH)
```

Proposition 6. *At the start of each run of the main loop, except the first, we will have initialized the running blocks B_j for $1 \leq j < \lceil H_1/H \rceil$ and $H_j = \ell \lceil H_1/j\ell \rceil$.*

Proof. At the start we have initialized only B_1 and $H_1 = H$. Then we put $h = H_1 + \ell = H + \ell$. Since $1 < \ell < H$[1] we would have $\lceil h/H \rceil = 2$. The index j in this first run will take only the value $j = 1$. Since B_1 is initialized we do not initialize any running block in this run. Since $H < \ell \lceil (H + \ell)/\ell \rceil = H + \ell$, the block B_1 will be shifted and H_1 will be put equal to $H + \ell$.

Therefore, at the end of the first run of the loop we have $H_1 = H + \ell$, so that $\lceil H_1/H \rceil = 2$, and the only running block initialized is B_1, and $H_1 = \ell \lceil H_1/\ell \rceil = H + \ell$. Therefore, the Proposition in true in this case.

Now by induction we may show that if our claim is true when a run starts, then it will be true at the end of this run.

---

[1] Since $\ell \mid H$, we have $\ell \leq H$. If $\ell = H$, we have $\ell = H \geq \sqrt{LN} = \sqrt{2\ell N}$, so that $\ell \geq 2N$, contradicting $\ell \mid N$.

At the start of a run we will have initialized the running block B_j for $1 \leq j < \lceil H_1/H \rceil$, and $H_j = \ell \lceil H_1/j\ell \rceil$.

We will initialize in the next loop those B_j not yet initialized, i. e. such that $\lceil H_1/H \rceil \leq j$, and such that newH $> H$, i. e. such that $H < \ell \lceil (H_1 + \ell)/j\ell \rceil$. Given natural numbers a, b and c, the relation $a < \lceil b/c \rceil$ is equivalent to $ac < b$. Therefore, since $\ell \mid H$, the condition $H < \ell \lceil (H_1 + \ell)/j\ell \rceil$ is equivalent to $Hj < H_1 + \ell$ and this is equivalent to $j < \lceil (H_1 + \ell)/H \rceil$. Thus at the end of the loop all blocks B_j with $1 \leq j < \lceil (H_1 + \ell)/H \rceil$ will be initialized. Since the new H_1 will then be equal to $H_1 + \ell$ we get half of our assertion.

Now we have to show that at the end of the loop H_j will be equal to $H_j = \ell \lceil (H_1 + \ell)/j\ell \rceil$.

For one of those blocks B_j that were initialized at the start of the loop we have $H_j = \ell \lceil H_1/j\ell \rceil$. If we have $H_j < \ell \lceil (H_1 + \ell)/j\ell \rceil$ then this block is shifted in this loop. If not, then we will have $H_j \geq \ell \lceil (H_1 + \ell)/j\ell \rceil$.

In the first case $H_j/\ell = \lceil H_1/j\ell \rceil < \lceil (H_1 + \ell)/j\ell \rceil$. This is an instance of $\lceil x \rceil < \lceil x+y \rceil$ with $0 < y \leq 1$, so that we will have $\lceil x + y \rceil = \lceil x \rceil + 1$. Therefore, $\lceil (H_1 + \ell)/j\ell \rceil = H_j/\ell + 1$. It follows that $H_j + \ell = \ell \lceil (H_1 + \ell)/j\ell \rceil$ and after the shift we will have what we want.

In the second case, the block has not been shifted, and $\ell \lceil (H_1 + \ell)/j\ell \rceil \leq H_j = \ell \lceil H_1/j\ell \rceil$. It follows that $H_j = \ell \lceil (H_1 + \ell)/j\ell \rceil$. Again what we want.

Now let B_j be one of the running blocks that have just been initialized. So $j \geq \lceil H_1/H \rceil$ and $H < \ell \lceil (H_1 + \ell)/j\ell \rceil$. After the initialization we have $H_j = H$, so that this block will be shifted. After the shift H_j is changed to $H + \ell$. Therefore, we must show that $H + \ell = \ell \lceil (H_1 + \ell)/j\ell \rceil$. First notice that $j \geq \lceil H_1/H \rceil$ is equivalent to $H \geq \ell \lceil H_1/j\ell \rceil$. Therefore

$$\Big\lceil \frac{H_1}{j\ell} \Big\rceil \leq \frac{H}{\ell} < \Big\lceil \frac{H_1}{j\ell} + \frac{1}{j} \Big\rceil.$$

But since $0 < 1/j \leq 1$ this implies

$$\Big\lceil \frac{H_1}{j\ell} \Big\rceil = \frac{H}{\ell} < \Big\lceil \frac{H_1}{j\ell} + \frac{1}{j} \Big\rceil = \frac{H}{\ell} + 1$$

from which we get $H + \ell = \ell \lceil (H_1 + \ell)/j\ell \rceil$. $\square$

3.8. **Definition of the function CalculateRunningBlock.** After each application of the function Shift(B, j), we must apply CalculateRunningBlock(j, newH). This computes the improved values of the complexity for $H_j - \ell < n \leq H_j$. Recall that the values of B_j were initialized to the upper bound 255.

Calculating new values for B_j. The steps are:

    1. For each $2 \leq a \leq b$, $H_j - \ell < ab \leq H_j$:

        a. Read $\|a\|$ from B_0. Read $\|b\|$ from B_0 if $b \leq H$, otherwise from B_{aj}.

        b. Set $\|a\| + \|b\|$ as the array value of $\|ab\|$ if it is lower than the current value.

    2. For each $H_j - \ell < n \leq H_j$ in ascending order:

a. Read $\|n-1\|+1$ from B_j and set this as the array value of $\|n\|$ if it is lower than the current value.

b. Calculate kMax for n using the current array value for $\|n\|$.

c. For each $6 \leq k \leq kMax$:

   * Read $\|k\|$ from B_0 and $\|n-k\|$ from B_j. Set $\|n-k\|+\|k\|$ as the array value for $\|n\|$ if it is lower than the current value.

```
<3.8 definition of the function CalculateRunningBlock>  ≡
<3.8.1 definition of Products>
<3.8.2 definition of Sums>
def CalculateRunningBlock(j,newH):
    Products(j,newH)
    Sums(j,newH)
```

3.8.1. *Definition of Products.* We consider all the products ab where $2 \leq a \leq b$, $H_j - \ell < ab \leq H_j$. We denote by Block the current running block, by BlockA and BlockB the blocks where $\|a\|$ and $\|b\|$, respectively, are to be found. At the same time we define shift, shiftA and shiftB, in such a way that $\|ab\|$ must be situated in Block $[ab - \text{shift}]$, $\|a\|$ in BlockA $[a - \text{shiftA}]$ and $\|b\|$ in BlockA $[b - \text{shiftB}]$.

```
<3.8.1 definition of Products>  ≡
def Products(j,newH):
    Block = B[j]
    shift = newH-L+1
    BlockA = B[0]
    shiftA = 0
    a = 2
    b = max(1+(newH-step)/a, a)
    while (a*a <= newH) and (b >= a):
        if newH/a <= H:
            BlockB = BlockA
            shiftB = shiftA
        else:
            BlockB = B[a*j]
            shiftB = Heads[a*j]-L+1
        bmax = newH/a
        ab = a*b
        while b <= bmax:
            if Block[ab-shift] > BlockA[a-shiftA] + BlockB[b-shiftB]:
                Block[ab-shift] = BlockA[a-shiftA] + BlockB[b-shiftB]
            b = b+1
            ab = ab+a
        a = a+1
        b = max(1+(newH-step)/a, a)
```

Proposition 7. *The procedure* Products(j, newH) *is correct, i. e. the needed complexities* $\|a\|$ *and* $\|b\|$ *are contained in the indicated blocks.*

Proof. 1. Proof that $a \leq H$ so that $\|a\|$ can always be read from the fixed block.

Since $a^2 \leq ab \leq \text{newH}$ we have $a \leq \sqrt{\text{newH}}$. On the other hand we have $\text{newH} \leq N + \ell$, because we only compute the complexities up to N. Then by (7)

$$a \leq \sqrt{\text{newH}} \leq \sqrt{N + \ell} < \sqrt{NL} \leq H.$$

2. Proof that $\frac{\mathrm{new}H}{a} \leq H$ or that B_{aj} has been initialized.

Assume that B_{aj} is not initialized. Since we are shifting the block B_j we are in step j of the Main Loop 3.8. The first not initialized running block will be B_u with $u = \lceil h/H \rceil$, so that $j < u \leq aj$, since B_j is initialized and B_{aj} is not. From $\lceil h/H \rceil \leq aj$ we get $h/H \leq aj$, so that $h/j \leq aH$.

Since B_j has already been shifted if needed, we have by Proposition 6, that $H_j = \mathrm{new}H = \ell \lceil h/j\ell \rceil$. Because $\ell \mid H$ it follows that

$$\mathrm{new}H = \ell \left\lceil \frac{h}{j\ell} \right\rceil \leq \ell \left\lceil aH\frac{1}{\ell} \right\rceil = aH.$$

Therefore, $b \leq \frac{\mathrm{new}H}{a} \leq H$. Hence if B_{aj} is not initialized, then $\frac{\mathrm{new}H}{a} \leq H$ and we may read $\|b\|$ from the fixed block B_0. But the program chooses $\mathrm{BlockB} = B_0$ when $\frac{\mathrm{new}H}{a} \leq H$.

3. If B_{aj} is initialized. By Proposition 6, we have in this case $aj < \lceil h/H \rceil$ and $\mathrm{new}H = \ell \lceil h/j\ell \rceil$. This implies respectively that $aj < h/H$, and (since $\ell \mid \mathrm{new}H$) $h/j\ell \leq \mathrm{new}H /\ell$. It follows that

$$aH < \frac{h}{j} \leq \mathrm{new}H \quad \text{so that} \quad H < \frac{\mathrm{new}H}{a}.$$

Hence in this case the program puts $\mathrm{BlockB} = B_{aj}$. In fact we can read $\|b\|$ from B_{aj} because

$$H_{aj} - \ell \leq \frac{\mathrm{new}H - \ell}{a} < b \leq \frac{\mathrm{new}H}{a} \leq H_{aj}.$$

The two intermediate inequalities are true because we have by hypothesis $\mathrm{new}H - \ell < ab \leq \mathrm{new}H$.

To prove the first inequality put $h = j\ell p - r$ with $0 \leq r < j\ell$ and $p = aq - s$ with $0 \leq s < a$. Then $h = aj\ell q - (sj\ell + r)$ and $0 \leq sj\ell + r < (s+1)j\ell \leq aj\ell$, so that

$$H_{aj} - \ell = \ell \left(\left\lceil \frac{h}{aj\ell} \right\rceil - 1 \right) = \ell(q-1) = \ell\frac{aq - s - (a-s)}{a} \leq$$
$$\leq \frac{\ell}{a}(p-1) = \frac{\ell}{a}\left(\left\lceil \frac{h}{j\ell} \right\rceil - 1 \right) = \frac{\mathrm{new}H - \ell}{a}.$$

With the same notation

$$\frac{\mathrm{new}H}{a} = \frac{\ell}{a} \left\lceil \frac{h}{j\ell} \right\rceil = \frac{\ell}{a}p = \frac{\ell}{a}(aq - s) \leq \ell q = \ell \left\lceil \frac{h}{aj\ell} \right\rceil = H_{aj}.$$

Hence if B_{aj} is initialized then $\|b\|$ can be read from BlockB which in this case will be B_{aj}, since in this case, $H < \frac{\mathrm{new}H}{a}$. $\square$

3.8.2. *Definition of Sums.* This is simpler than the case of the products. The definition of the function kMaxfor(n, s) should be clear after our explanation in Section 2.7. The definition of the function $E(\cdot)$ is contained in Section 2.3.

As explained in Section 2.8 we only have to test the values of $k \leq$ kMax which are solid numbers. We precompute the first few of these numbers and put them in a file in the form solid $= [\, 1, 6, 8, 9, 12, \dots \,]$. If this file is not large enough an index-error will appear. In practice this will not be a problem because kMax is relatively small and the sequence of solid numbers is easy to compute.

```
<3.8.2 definition of Sums>  ≡
from solid import *
def kMaxfor(n,s):
    target = s
    t = target/2
    while E(t)+E(target-t) < n:
        t = t-1
    return E(t)
def Sums(j,newH):
    for n in range(newH-step+1,newH+1):
        Block = B[j]
        shift = newH-L+1
        s = Block[n-1-shift]+1
        m = n
        kMax = kMaxfor(m,s)
        Block0 = B[0]
        r = 0
        b = solid[r]
        while b <= kMax:
            if Block[n-shift] > Block[n-b-shift]+Block0[b]:
                Block[n-shift] = Block[n-b-shift]+Block0[b]
            r = r+1
            b = solid[r]
```

Since we have taken $\ell >$ kMax for all values of n in $1 \leq n \leq N$ it is clear that $\|n - k\|$ can always be read from the current running block. Indeed, since the complexities $\|n\|$ for newH $-\ell < n \leq$ newH are computed in increasing order and the values in the first half of the running block (which has length ℓ) are correct from the start. Also the value of $\|k\|$ can always be read from the fixed block since $k \leq$ kMax $\leq \ell \leq H$.

3.9. Some definitions.
There are two simple functions in the main loop that we have not yet defined. The ceiling function is used for certain ranges in the Main Loop. To check whether a running block is initialized we use the length of the Heads list. Each time a running block is initialized we put a new element in this list.

```
<3.8.2 Some definitions>  ≡
def ceiling(n,m):
        return n/m+(n%m!=0)
def notInitialized(j):
    return len(Heads) < j+1
```

4. Upper bounds for $\|n\|$.

To estimate the running time of our algorithms we need upper bounds for $\|n\|$. We get some useful upper bounds by expressing n in a base b and using Horner's algorithm. For

example $n = r_0 r_1 r_2 \ldots r_k$ where $0 \le r_j < b$ and $r_0 \ge 1$. This is equivalent to

(8) $\quad n = r_k + r_{k-1}b + r_{k-2}b^2 + \cdots r_0 b^k = r_k + b(r_{k-1} + b(r_{k-2} + \cdots + b(r_1 + br_0)\cdots))$.

For each digit $0 \le r < b$ we write $D(b, r)$ for the complexity of multiplying by b and adding r. E. g. $D(6, 4) \le 7$ because $\|6n + 4\| = \|3(2n + 1) + 1\| \le \|n\| + 3 + 2 + 1 + 1 = \|n\| + 7$. In general we define $D(b, r)$ as the least number satisfying

(9) $\qquad\qquad\qquad\qquad \|r + bn\| \le \|n\| + D(b, r)$

for all $n \ge 1$.

We will always have $D(b, r) \le \|b\| + \|r\|$. This inequality may be strict. For example we have just seen that $D(6, 4) \le 7$ and $\|6\| + \|4\| = 9$.

Proposition 8. *If the expansion of n in base b is given by $n = r_0 r_1 r_2 \ldots r_k$, then*

(10) $\qquad\qquad\qquad\qquad \|n\| \le \|r_0\| + \sum_{j=1}^{k} D(b, r_j).$

Proof. For numbers with one digit this is trivially true. In the other case we have $n = r_k + mb$ with $m = r_0 r_1 r_2 \ldots r_{k-1}$ so that by induction we get

$$\|n\| \le \|m\| + D(b, r_k) = \|r_0\| + \sum_{j=1}^{k-1} D(b, r_j) + D(b, r_k).$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \square$

By means of the following proposition we may easily obtain upper bounds for $D(b, r)$.

Proposition 9. *Let $d \mid b$ with $1 < d < b$, where $b = da$. For $1 \le r \le b$ put $r = qd + s$. Then*

(11) $\qquad\qquad D(b, r) = D(da, qd + s) \le D(d, s) + D(a, q).$

Proof. We have

$$\|nb + r\| = \|(na + q)d + s\| = D(d, s) + \|na + q\| \le D(d, s) + D(a, q) + \|n\|.$$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \square$

We define a function $D_0(b, r)$ for $b \ge 2$ and $0 \le r < b$ by induction on $b \ge 2$.

First when $b = p$ is prime

(12) $\qquad\qquad\qquad\qquad D_0(p, r) := \|p\| + \|r\|$

where we take $\|0\| := 0$, and in general

(13) $\qquad D_0(b, r) := \min\{\|b\| + \|r\|, \min_{1 < d < b, d \mid b} D_0(d, \mathrm{mod}(r, d)) + D_0(b/d, \lfloor r/d \rfloor)\}.$

By induction we will then find that

$$(14) \qquad D(b, r) \le D_0(b, r).$$

Remark 10. The best bounds are usually obtained with bases of the form $b = 2^n 3^m$. Notice that to compute $D(b, r)$ we need to precompute only the values of $D(b', r)$ for all proper divisors $b' \mid b$.

4.1. **Average bound of the complexities.** We will give here an application to the general theory of the complexity of natural numbers. We may define several constants associated with the bounds on the complexity.

Definition 11. Let $C_{\max}$ denote the $\limsup_{n \to \infty} \frac{\|n\|}{\log n}$. Let C_{avg} be the infimum of all C such that

$$\|n\| \le C \log n$$

for a set of natural numbers of density 1.

We have

$$\frac{3}{\log 3} \le C_{\text{avg}} \le C_{\max} \le \frac{3}{\log 2}.$$

In [4] it is said that Isbell has shown using the expression of n in basis 24 that $C_{\text{avg}} \le 3.475$. In [12] Steinerberger considered also a related problem obtaining for a slightly different constant the bound $C'_{\text{avg}} \le 3.332$.

Proposition 12. *For any basis $b \ge 2$ we have*

$$(15) \qquad C_{avg} \le \frac{1}{b \log b} \sum_{r=0}^{b-1} D(b, r).$$

In particular we get

$$C_{avg} = \frac{41\,747\,875}{2^7 3^8 \log(2^9 3^8)} = 3.3080772123153688960\ldots\cdots \le 3.309.$$

Proof. Given a number in base b as $n = r_0 r_1 \ldots r_k$ with $r_0 \ne 0$, then $n \ge b^k$ so that by (10) we have

$$\frac{\|n\|}{\log n} \le \frac{\|r_0\|}{k \log b} + \frac{1}{k \log b} \sum_{j=1}^{k} D(b, r_j).$$

The first term tends to 0 as n tends to ∞. By Chernoff's Theorem, for almost all numbers the difference

$$\frac{1}{k} \sum_{j=1}^{k} D(b, r_j) - \frac{1}{b} \sum_{r=0}^{b-1} D(b, r)$$

is small. This proves (15).

We may easily compute the numbers $\frac{1}{b}\sum_{r=0}^{b-1} D(b,r)$, for increasing bases. The smaller values are obtained for basis of the form $b = 2^n 3^m$. Having computed all numbers $D(b,r)$ for all bases $2^n 3^m < 3359232 = 2^9 3^8$, we get the best value for $b = 2^9 3^8$ for which

$$\frac{1}{b \log b}\sum_{r=0}^{b-1} D(b,r) = 3.3080772123153688960\ldots.$$

$\square$

5. PERFORMANCE OF THE TIME-IMPROVED ALGORITHM.

Theorem 13. *The time-improved algorithm presented in Section 2 computes all $\|n\|$ for $n \leq N$ in time $\mathcal{O}(N^{1.230175})$ and space $\mathcal{O}(N \log\log N)$.*

Proof. Clearly we need space for the array Compl. At the start of the algorithm we initialize this array by a common value cMax of the order $\log N$ (observe that to simplify the notation we have put $N = \mathrm{nMax}$), and we need $\log\log N$ bits to store this value. During the calculation each element of the array is decreased, so that we need $\mathcal{O}(N \log\log N)$ bits of space. The space needed for the rest of the computation is only $\mathcal{O}(\log N)$, needed to store some numbers $\leq N$ or to compute $E(t)$ for t of the order of $\log N$.

The test for the products takes a number of operations of the order

$$\sum_{n=1}^{N} \min(n, N/n) = \sum_{n=1}^{\sqrt{N}} n + \sum_{n=\sqrt{N}}^{N} \frac{N}{n} = \mathcal{O}(N) + \mathcal{O}(N \log N) = \mathcal{O}(N \log N).$$

The cost of computing kMax for a value of n is $\mathcal{O}(\log^2 n)$ operations. In fact the starting value of k is of the order $\|n-1\|$, so of order $\log n$. In the procedure k will be changed at most k times and for each of these we have to compute two values of E for numbers of size $\log n$ at a cost of $\mathcal{O}(\log n)$ operations and $\mathcal{O}(\log n)$ space.

In the main loop for each $n \leq \mathrm{nMax}$ we have precomputed $\|n-1\|$. We compute kMax for such an n and then in the check of sums we run the variable m from 6 to kMax requiring a fixed number of operations for each value of m. The cost of all this is of the order of

$$C = \sum_{n=1}^{N} (\log^2 n + \mathrm{kMax}(n)) = \mathcal{O}(N \log^2 N) + \sum_{n=1}^{N} n\Big(1 - \sqrt{1 - \frac{4}{n^2} 3^{\|n-1\|/3}}\Big).$$

Since

$$4\frac{3^{\|n-1\|/3}}{n^2} \leq 4\frac{\exp\Big(\frac{\log 3}{3}\frac{3}{\log 2}\log n\Big)}{n^2} \leq 4\, n^{-0.415037}$$

tends to 0 when $n \to \infty$ we may bound the cost by

$$C \leq \mathcal{O}(N \log^2 N) + \mathcal{O}\Big(\sum_{n=1}^{N} \frac{3^{\|n-1\|/3}}{n}\Big).$$

Now choose a base b and apply the bound (10). Taking a number a such that $b^{a-1} \leq N < b^a$ we will have

$$\mathcal{O}\Big(\sum_{n=1}^{N} \frac{3^{\|n-1\|/3}}{n}\Big) \leq \mathcal{O}\Big(\sum_{\ell=1}^{a} \sum_{b^{\ell-1} \leq n < b^\ell} \frac{3^{\|n\|/3}}{n}\Big) = \mathcal{O}_b\Big(\sum_{\ell=1}^{a} b^{-\ell} \sum_{b^{\ell-1} \leq n < b^\ell} 3^{\|n\|/3}\Big).$$

Now in the inner sum n runs through all the numbers that in base b have ℓ digits. By (10) we will have

$$\sum_{b^{\ell-1} \leq n < b^\ell} 3^{\|n\|/3} \leq C_b \sum_{b^{\ell-1} \leq n < b^\ell} 3^{\frac{1}{3}\sum_{j=1}^{\ell} D(b,b_j)} \leq C_b\Big(\sum_{d=0}^{b-1} 3^{\frac{1}{3}D(b,d)}\Big)^\ell = C_b A_b^\ell.$$

Hence

$$C \leq \mathcal{O}(N \log^2 N) + \mathcal{O}_b\Big(\sum_{\ell=1}^{a} (A_b/b)^\ell\Big) = \mathcal{O}(N \log^2 N) + \mathcal{O}_b((A_b/b)^a).$$

Since $N \sim b^a$ with a constant only depending on b

$$(A_b/b)^a = \exp\big(a \log(A_b/b)\big) = b^{a\frac{\log(A_b/b)}{\log b}} = \mathcal{O}(N^\alpha),$$

where

$$\alpha = \frac{\log(A_b/b)}{\log b} = -1 + \frac{1}{\log b} \log\Big(\sum_{d=0}^{b-1} 3^{\frac{1}{3}D(b,d)}\Big).$$

We have computed α for all bases $b = 2^n 3^m \leq 3\,188\,246$. For $b = 2\,239\,488 = 2^{10}3^7$ we found the smallest value

$$\alpha = \frac{\log(3^6 2^{-10}(30\,357\,189 + 21\,079\,056 \cdot 3^{1/3} + 14\,571\,397 \cdot 3^{2/3})}{\log(2^{10}3^7)} =$$

$$1.230\,174\,997\,215\,298\,061\,586 \cdots < 1.230175.$$

$\square$

6. Performance of Fuller's algorithm.

Proposition 14. *The space-improved algorithm presented in Section 3 computes $\|n\|$ for all $n \leq N$ in time $\mathcal{O}(N^\alpha)$ using $\mathcal{O}(N^{(1+\beta)/2} \log \log N)$ bits of storage. ($\alpha = 1.230175$ and $(1 + \beta)/2 \approx 0.792481$.)*

Proof. In the last run of the Main Loop we have $H_1 \leq N + \ell$ so that the total number of initialized running blocks will be $\leq \lceil (N + \ell)/H \rceil \leq N/H + 2$.

Hence we need a fixed block of length H and $\approx N/H$ running blocks of length L. Each entry in the blocks must contain a value of the complexity, each of $\log \log N$ bytes. So the required space will be

$$\leq \Big(H + \frac{N}{H}L\Big) \log \log N.$$

The value of L is limited by our conditions $\text{kMax} \le \ell$ and $L = 2\ell$. Therefore, an L of the order $\mathcal{O}(N^\beta)$ or larger would be adequate by Corollary 4. Given N and L the best choice of H (the one requiring less space) is $H = \sqrt{LN}$, and this will give a space requirement of $\sqrt{NL} \log \log N$. So the best choice will be to take $L = \mathcal{O}(N^\beta)$, and $H = \mathcal{O}(N^{(1+\beta)/2})$. It is easy to see that increasing H and N a little (if needed) we may also satisfy the conditions $\ell \mid H$ and $\ell \mid N$.

So the space requirement for the algorithm is $\mathcal{O}(N^{(1+\beta)/2} \log \log N)$, and this choice will satisfy all the conditions in (7).

The time and the number of operations needed for the computation is as follows

1. Computing the fixed block takes $\mathcal{O}(H^\alpha)$ operations.

2. Copying from B_0 into B_1 takes $\mathcal{O}(L)$ operations.

The running block B_j starts at H and ends at $\approx N/j$ in steps of size ℓ. Hence B_j must be adapted about $N/j\ell$ times.

There are N/H running blocks each of length L. Initializing all these requires

$$\le C \sum_{j \le N/H} \frac{N}{j} = \mathcal{O}(N \log N/H) \qquad \text{operations.}$$

To compute the new values for the block B_j (for the products) requires a fixed number of operations for each $2 \le a \le b$ with $H - \ell < ab \le N/j$. The total cost of the products is therefore

$$\le C \sum_{j \le N/H} \sum_{a \le \sqrt{N/j}} \frac{N}{aj} \le C \sum_{j \le N/H} \frac{N}{j} \log \sqrt{N/j} \le CN \log N \log(N/H) \le \mathcal{O}(N \log^2 N).$$

To compute the new values for the block B_j (for the sums) we have to perform the same number of operations as in the time-improved algorithm of Section 2. This cost was $\mathcal{O}(N^\alpha)$. Hence the cost for the sums in the block B_j is at most $C(N/j)^\alpha$. It follows that all the sums for the block cost

$$\le C \sum_{j \le N/H} \left(\frac{N}{j}\right)^\alpha \le C\zeta(\alpha)N^\alpha = \mathcal{O}(N^\alpha).$$

So the total cost of the algorithm is

$$\mathcal{O}(N^\alpha) + \mathcal{O}(N \log^2 N) + \mathcal{O}(N \log(N/H)) = \mathcal{O}(N^\alpha).$$

$\square$

7. Acknowledgement

References

[1] H. Altman, J. Zelinsky, *Numbers with integer complexity close to the lower bound*, Integers **12** (2012) 1093-1125.

[2] J. Arias de Reyna, *Complejidad de los números naturales*, Gaceta de la Real Sociedad Matemática Española **3** (2000) 230–250.

[3] J. Arias de Reyna, J. van de Lune, *The question "How many 1's are needed?" revisited*, (2009) arXiv:1404.1850.
http://arxiv.org/abs/1404.1850

[4] R. K. Guy, *What is the least number of ones needed to represent n using only + and × (and parentheses)?*, American Mathematical Monthly **93** (1986) 189–190.

[5] R. K. Guy, *Unsolved Problems in Number Theory*, Third edition, Springer-Verlag, New York, 2004.

[6] J. Iraids, K. Balodis, J. Čerņenoks, M. Opmanis, R. Opmanis, K. Podnieks, *Integer Complexity: Experimental and Analytical results*, Scientific papers University of Latvia, Computer Science and Information Technologies, **787** (2012) 153–179. arXiv:1203.6462, (2012).
http://front.math.ucdavis.edu/1203.6462

[7] M. N. Fuller, *C-program to compute A005245*, February 2008.
http://oeis.org/A005245/a005245.c.txt

[8] D. E. Knuth, *Literate Programming*, The Computer Journal **27** (1984) 97–111.

[9] K. Mahler, J. Popken, *On a maximum problem in arithmetic*, (in Dutch), Nieuw Arch. Wiskunde (3) **1** (1953) 1–15.

[10] D. A. Rawsthorne, *How many 1's are needed?*, Fibonacci Quart. **27** (1989) 14–17.

[11] V. V. Srinivas, B. R. Shankar, *Integer Complexity: Breaking the $\Theta(n^2)$ barrier*, World Academy of Science **48** (2008) 690–691.

[12] S. Steinerberger , *A short note on Integer Complexity*, Contribution to Discrete Mathematics, to appear.

[13] OEIS Foundation Inc, *The On-Line Encyclopedia of Integer Sequences*, (2012).
http://oeis.org.

Facultad de Matemáticas, Universidad de Sevilla,
Apdo. 1160, 41080-Sevilla, Spain

E-mail address: arias@us.es

Langebuorren 49, 9074 CH Hallum, The Neterlands
(Formerly at CWI, Amsterdam)

E-mail address: j.vandelune@hccnet.nl