3-1992

3x + 1 Search Programs

Gary T. Leavens
Iowa State University

Mike Vermeulen
Iowa State University

$3x + 1$ Search Programs

Gary T. Leavens and Mike Vermeulen

TR #92-01
March, 1992

Keywords: $3x+1$ problem, Collatz problem, optimization, algorithmic improvement, efficiency, performance.
1991 CR Categories: C.4 [*Performance of Systems*] Design studies, performance attributes; F.2.1 [*Analysis of Algorithms and Problem Complexity*] Numerical Algorithms and Problems — computations on polynomials, number-theoretic computations; I.1.2 [*Algebraic Manipulation*] Algorithms — algebraic algorithms; J.2 [*Physical Sciences and Engineering*] Mathematics and statistics.

To appear in *Computers and Mathematics with Applications.*

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Computers Math. Applic. Vol. ??, No. ??, pp. 1–??, 1992
Printed in Great Britain.
0097-4934/92 $3.00 + 0.00

$3X + 1$ SEARCH PROGRAMS

Gary T. Leavens*
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu
Mike Vermeulen
Hewlett Packard, 3404 East Harmony Road
Fort Collins, CO 80525 USA
mev@hpfcrt.fc.hp.com

Abstract — Algorithms for computing peaks of certain statistics related to the $3x+1$ problem are described, along with data on such peaks up to 56 trillion (5.6×10^{13}). The data result from several years of computation. The design of the algorithms used illustrates several techniques for program optimization.

1. Introduction

The $3x + 1$ problem concerns iterates of the following function:

$$T(n) = \begin{cases} (3n+1)/2, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{1}$$

which takes odd integers n to $(3n + 1)/2$ and even integers n to $n/2$ [5]. The $3x + 1$ Conjecture asserts that, starting from any *positive* integer n, repeated iteration of this function eventually produces the value 1. This conjecture is apparently intractable.

The iterates of T are simply defined. Let $T^{(0)}(n) = n$, and for all integers $k > 0$, let $T^{(k)}(n) = T(T^{(k-1)}(n))$. The sequence of iterates $(T^{(0)}(n), T^{(1)}(n), T^{(2)}(n), \ldots)$ is called the *T-trajectory* of n. For example, the T-trajectory of 7 is:

$$7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1, 2, 1, 2, 1, \ldots$$

An alternative formulation of the $3x + 1$ problem considers iterates of the function H that does not map odd integers n to $(3n + 1)/2$, but rather to $3n + 1$:

$$H(n) = \begin{cases} 3n+1, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{2}$$

The function H is modeled after the so-called hailstone algorithm, see Hayes [2]. One defines the iterates of H in the same way as T. For example, if n is 7, then the sequence of successive iterates of H is:

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$$

*Leavens's work was supported in part by the National Science Foundation under grant CCR-9108654 by the ISU Achievement Foundation and by a GenRad/AEA Faculty Development Fellowship; while graduate students at MIT both authors' work was supported in part by the National Science Foundation under Grants DCR-8510014 and CCR-8716884, and the Defense Advanced Research Projects Agency under Contract N00014-83-K-0125. Computer support was provided in part by the National Science Foundation under Grant DCR 8503662.

We will call such a sequence a *H-trajectory*.

Notice how the H-trajectory of 7 differs from its T-trajectory. The difference is stated precisely in the following lemma. The first part states that the T-trajectory of a number is a subsequence of the corresponding H-trajectory, with the property that every even number in the sequence of iterates of H can be paired with a number in the iterate sequence of T. The second part states that every number k in the H-trajectory of n either occurs in the T-trajectory of n or else $k/2$ does.

LEMMA 1 *(a) For all $i > 0$, for all $n > 0$, and for all $k > 0$, if $T^{(i)}(n) = k$, then there is some $j \geq 0$ such that, $H^{(j)}(n) = 2k$, and $H^{(j+1)} = k$.*

(b) For all $j > 0$, for all $n > 0$, and for all $k > 0$, if $H^{(j)}(n) = k$, then there is some $i \geq 0$ such that either $T^{(i)}(n) = k$ or $T^{(i)}(n) = k/2$.

PROOF: by induction on i, respectively j, for (a) and (b). ∎

There are graphs in the article by Hayes that show the wildly erratic and unpredicatable behavior of the iterates of H [2]. The behavior of T is, of course, similarly wild and unpredictable.

We define certain statistics that measure various attributes of the iterates of the functions T and H. These statistics are related to the $3x + 1$ conjecture, since one counts the number of iterations needed to reach 1, or the maximum value reached in a trajectory.

A necessary condition for $T^{(k)}(n) = 1$ is that there is some m such that $T^{(m)}(n) < n$. The *stopping time* $\sigma(n)$ is the least whole number k such that $T^{(k)}(n)$ is less than n, with the convention that $\sigma(1) = 0$. If there is no such k, then let $\sigma(n)$ be ∞. For example, $\sigma(7) = 7$.

The *total stopping time* $\sigma_\infty(n)$ is the least whole number k such that $T^{(k)}(n)$ is one, with the convention that $\sigma_\infty(1) = 0$. If there is no such k, then let $\sigma_\infty(n)$ be ∞. For example, $\sigma_\infty(7) = 11$.

The value of $steps(n)$ is the least whole number k such that $H^{(k)}(n)$ is one. If there is no such k, then let $steps(n)$ be ∞. It is the analog of total stopping time for H. For example, $steps(7) = 16$.

The maximum value, max_value(n), is the least upper bound of all the integers reached by iterating H until the value of the iterates reach one. That is,

$$max\_value(n) = \text{lub}\{H^{(k)}(n) \mid 0 \leq k \leq steps(n)\}. \tag{3}$$

For example, $max\_value(7) = 52$.

Using T instead of H gives an alternative definition of maximum value.

$$alt\_max\_value(n) = \text{lub}\{T^{(k)}(n) \mid 0 \leq k \leq \sigma_\infty(n)\}. \tag{4}$$

For example, $alt\_max\_value(7) = 26$.

Except in certain special cases, the *max_value* of a number is always twice its *alt_max_value*. The proof of this simple fact depends on part (a) of the following lemma, which states that if a value occurs in the T-trajectory of a number greater than two, then it must occur before the iterates reach one.

LEMMA 2 *(a) For all $n > 2$, and for all k, if there is some $i \geq 0$ such that $H^{(i)}(n) = k$, then there is some $0 \leq j \leq steps(n)$ such that $H^{(j)}(n) = k$; furthermore, if there is some $i \geq 0$ such that $T^{(i)}(n) = k$, then there is some $0 \leq j \leq \sigma_\infty(n)$ such that $T^{(j)}(n) = k$.*

(b) For all $n > 0$, if $alt\_max\_value(n) \neq n$, then $max\_value(n) = 2 \cdot alt\_max\_value(n)$. ∎

We are interested in the behavior of these statistics as n varies. Many facts are known about them, see Lagarias [5]. A number of researchers have observed that if the input n is drawn randomly, say with the uniform distribution on an interval $[1, N]$, then these statistics appear to have nice limiting distributions as N approaches infinity (see, for example, Rawsthorne [10] and Wagon [11]). Lagarias and Weiss [6] describe various random walk models intended to simulate $3x + 1$ function iterates.

Particularly interesting statistics concern the behavior of extreme values of these statistics as n varies, which we call peaks. An integer $n > 0$ is a *peak* in a statistic f, if and only if for all

	σ	σ_∞	*steps*	*max_value*	*alt_max_value*
σ	same				
σ_∞	distinct	same			
steps	distinct	*unknown*	same		
max_value	distinct	distinct	distinct	same	
alt_max_value	distinct	distinct	distinct	*same*	same

Table 1: Relationships between peaks in various statistics.

$0 < m < n$, $f(m) < f(n)$. These numbers are called peaks because if one graphs the function, then each peak will be a point higher than has been reached for smaller n. For example, 3 is a peak in *max_value*, because $max\_value(3) = 16$, $max\_value(2) = 2$, and $max\_value(1) = 1$.

This paper describes algorithms for computing such peaks; tables of peaks appear in the Appendix. The algorithmic optimizations described below were developed in a spirit of friendly competition between the two authors. Each author developed a program, and so many of our results have been validated by more than one program. The first program (by Leavens) designed to experiment with the Argus distributed programming language and system [9] [8]. A second program (by Vermeulen) was written in C, to understand the inherent costs of Argus as opposed to C. Various distributed programs have been running since 1986, although only Vermeulen's program is currently running. Vermeulen's current system, which was started in August 1990, runs on about 15 workstations (the exact number varies), searches an interval of about 100 billion per night, and has accumulated between 5 and 15 *years* of CPU time. (That is, if the search were run sequentially it would take about 5 CPU-years on the fastest machine, or about 15 CPU-years on the slowest.)

Peaks appear more and more rarely as one tests larger numbers. While it is easy to verify the value of $steps(n)$ or $max\_value(n)$ for any particular n, it is very expensive to verify that n is a peak in either statistic, because this involves showing that all numbers less than n have a smaller value for *steps* or *max_value*.

One of the mathematical questions we have investigated is whether peaks in one statistic must be peaks in some other statistic. Since the trajectories under H and T are related (as in Lemma 1), one might guess that numbers that are peaks in a measure based on iterates of T would necessarily be peaks in a measure based on H. This is true for maximum values, but the question remains open for stopping times and steps. The relationships are summarized in Table 1.

The fact that peaks in *max_value* and *alt_max_value* are the same is a corollary of Lemma 2 (b).

Corollary 3 *An integer $k > 0$ is a peak in* max_value *if and only if k is also a peak in* alt_max_value. ∎

Counter-examples that distinguish between peaks in most other measures can be found by examining the tables in Appendix A.

The relationship between peaks in *steps* and peaks in σ_∞ is one of the difficult open questions that often appear in the study of the $3x + 1$ problem. However, it is easy to see that the total stopping time cannot be greater than the number of steps, nor as small as half the number of steps.

Lemma 4 *For all $n > 0$, if $steps(n) \neq \infty$, then $steps(n)/2 < \sigma_\infty(n) \leq steps(n)$.*

Proof: In any H-trajectory, every step of the form $3x + 1$ is followed by a division by 2. Thus the T-trajectory omits at most half of these steps. However, if $steps(n)$ is defined, then there must be more division by 2 steps in the iteration of H than there are steps that multiply by 3 and add one. ∎

A different relationship holds between total stopping time and *steps*. Let $odd(n)$ be the number of odd integers in the iterate sequence of H (excluding 1) and $even(n)$ be the number of even integers that occur until 1 is reached. That is:

$$odd(n) \stackrel{\text{def}}{=} \#\{k \mid k \bmod 2 = 1, H^{(i)}(n) = k, 0 \le i < steps(n)\} \quad (5)$$

$$even(n) \stackrel{\text{def}}{=} \#\{k \mid k \bmod 2 = 0, H^{(i)}(n) = k, 0 \le i < steps(n)\}. \quad (6)$$

Since every step produces an odd or an even number, the sum of odd and even is the number of steps; that is, For all $n > 0$, $steps(n) = odd(n) + even(n)$.

It is interesting that the number of even steps is the same as the total stopping time.

LEMMA 5 *For all $n > 0$, $\sigma_\infty(n) = even(n)$.*

PROOF: If $\sigma_\infty(n) = \infty$, then $even(n) = \infty$. Furthermore, $\sigma_\infty(1) = 0 = even(1)$ and $\sigma_\infty(2) = 1 = even(2)$.

So suppose $n > 2$ and $\sigma_\infty(n) = m < \infty$. By Lemma 1, for each $0 \le i \le \sigma_\infty(n)$, there is some $j \ge 0$ such that $2T^{(i)}(n) = H^{(j)}(n)$. By Lemma 2 j can be chosen so that $j \le steps(n)$. Thus $\sigma_\infty(n) \ge even(n)$. However, if $\sigma_\infty(n) > even(n)$, then there would have to be two iterates of T with the same value (that is, $0 \le i < l \le \sigma_\infty(n)$, such that $T^{(i)}(n) = T^{(l)}(n)$), but if this happened there would be infinitely many such cases, and so $\sigma_\infty(n)$ would be infinite. ∎

The above analysis does not seem to lead to a proof that peaks in *steps* are also peaks in total stopping time. However, the peaks do coincide at least to 12.3 billion (12.3×10^9), as the first author's program verified. So we offer the following conjecture.

CONJECTURE 6 *An integer $k > 0$ is a peak in* steps *if and only if k is a peak in total stopping time (σ_∞).*

This conjecture seems difficult to prove.

The rest of the paper is organized as follows. Section 2. discusses smaller-scale efficiency issues, and describes how we turn mathematical insight into better ways to prune the search space. Section 3. draws some conclusions from this experience. Appendix A gives tables of results from the search.

From the tables we make the following observations.

- 1, 2, 3, 7, 27, and 703, are the only known peaks in *steps*, stopping time (σ), and *max_value*; no larger number is known to be a peak in all three of these statistics.
- 12,235,060,455 is the largest known number that is a peak in both *steps* and σ; no larger number is known to be a peak in more than one of: *steps*, stopping time (σ), and *max_value*.
- Despite the previous remark, many of the peaks in *steps* have the same *max_value*, and hence their trajectories are identical after a certain number of iterations.

  Unlike the peaks in *steps*, the maximum values reached by peaks in stopping time, σ, rarely repeat.

See also Lagarias and Weiss for more detailed comparisons in a similar vein [6].

2. Small Scale Design Issues

This section describes algorithms for iterating H and T. Another issue of practical importance, how to efficiently distribute the search among several computers, will not be discussed in this paper (see [7]).

A fundamental observation is that peaks are extremely rare. For example, in the first 50 billion positive integers, there are only 49 peaks in *max_value* and only 78 peaks in *steps*. The peaks become more and more rare as the search progresses; between 1 billion and 50 billion there are only 5 peaks in *max_value* and 12 peaks in *steps*. So a typical number is not a peak, and the main task of the search is to find this out as quickly as possible. For doing this there are three basic strategies.

Optimization	Speedup	Cumulative Speedup
None	1.00	1.0
Use Composite polynomials	7.00	7.0
Search only odd numbers	2.00	14.0
Values: *a posteriori* cutoffs	1.84	25.7
Ignore $k \bmod 6 = 5$	1.41	36.4
Values: *a priori* cutoff with 8 bit polynomial	1.71	62.1
Steps: *a priori* cutoff using 8 bit polynomial	1.15	71.3
Use 16 bit polynomials for *a priori* cutoffs	1.36	97.1
Steps: *a posteriori* cutoffs	1.88	182.0

Table 2: Effectiveness of optimizations

- *Cutting off the search* by discovering that the input number is not a peak before taking it through all the iterates of H or T down to 1 (or until the values of the iterates fall below the starting value if one is searching for peaks in stopping time).
- Running the steps of the iteration algorithm faster.
- Multiplying, dividing, adding, and comparing numbers faster. Algorithms and data structures for large precision integers were important practical considerations, because the searches went well past the usual 32 bit integers, and because the iterates exceed these limits quickly. For example, $max\_value(159{,}487) = 17{,}202{,}377{,}752$. Such algorithms and data structures are well known [4].

The benefits of the optimizations discussed in this section are summarized in Table 2. The speedup due to a particular optimization depends on the order in which optimizations are applied. For example, the *a priori* cutoff of all numbers $k \bmod 6 = 5$ reduces the set of numbers to be searched by a sixth if one is searching all numbers, but by a third if the set is already restricted to even numbers. The table lists the optimizations in the order that they were added to Mike Vermeulen's program, and then lists the speedup achieved by that optimization assuming the optimizations listed above it were already applied.

The speedup of seven for using polynomials results from a comparison between an assembly language program and a polynomial based program. All other speedups were measured while selectively disabling optimizations on the polynomial search.

2.1. A Priori Cutoffs

The best way to cut off the search on a given input number is to prove that the input cannot be a peak and to ignore it without spending time on it; this is called an *a priori* cutoff. A less effective way to cut off the search on a given input is to prove that the number cannot be a peak after learning something about its trajectory; this is called an *a posteriori* cutoff.

A basic result is that it is possible, *a priori*, to limit the search to odd numbers.

For *max_value*, it suffices to note that the first step of H for an even number is to divide it by two.

LEMMA 7 *For all* $k > 0$, $max\_value(2k) = \max\{2k, max\_value(k)\}$. ∎

The technique used in the proof of the following corollary is an example of reasoning about the convergence of different trajectories.

COROLLARY 8 *The number 2 is the only even peak in* max_value.

PROOF: Let $k > 1$ be given. By the above lemma, $max\_value(2k)$, is the maximum of $2k$ or $max\_value(k)$. If $max\_value(2k) = max\_value(k)$, then $2k$ is not a peak. So suppose

$max\_value(2k) = 2k$. But then $2k$ is not a peak in *max_value* either, since $max\_value(2k-1) \geq 3(2k-1) > 2k$. The inequality $max\_value(2k-1) \geq 3(2k-1)$ holds because $2k-1$ is odd, hence $H(2k-1) = 3(2k-1)$. That $3(2k-1) > 2k$ holds for $k > 1$ holds is shown by the following:

$$\begin{aligned} k > 1 &\Rightarrow 4k > 4 && (7)\\ &\Rightarrow 4k - 3 > 1 && (8)\\ &\Rightarrow (6k-3) - 2k > 1 && (9)\\ &\Rightarrow (6k-3) > 2k + 1 && (10)\\ &\Rightarrow 3(2k-1) > 2k. && (11) \end{aligned}$$

∎

Results similar to the above apply to *alt_max_value* as well.

For *steps*, the same observation about the first step of H means that an even number k will take only one more step than $k/2$ to return to 1.

LEMMA 9 *For all $k > 0$, $steps(2k) = 1 + steps(k)$.* ∎

COROLLARY 10 *If k is a peak in* steps, *then the least even number greater than k that can be a peak in* steps *is $2k$.*

PROOF: Let k be a peak in *steps*. By definition, for all $0 < j < k$, $steps(j) < steps(k)$. Thus by the preceding lemma, $steps(2j)$ is constrained as follows:

$$steps(2j) = 1 + steps(j) \leq steps(k) = steps(2k) - 1 < steps(2k). \quad (12)$$

∎

A similar result applies to total stopping time.

By these corollaries, it is easy to predict all the even peaks in *steps* and *max_value*. Thus the search for these peaks ignores all the even numbers, giving a factor of 2 speedup.

For stopping time, the first division by two means that an even number always has a stopping time of 1.

LEMMA 11 *For all $k > 0$, $\sigma(2k) = 1$.* ∎

COROLLARY 12 *If $k > 2$ is a peak in stopping time, then k is odd.* ∎

So the search for peaks in stopping time also ignores all the even numbers.

The following results allow the search for peaks in stopping time to effectively ignore half of the odd numbers as well, that is, those that are equal to 1 modulo 4. The lemma predicts the first iteration of T from the fact that the number is equal to 1 modulo 4, and the corollary carries this analysis one iteration further to predict the stopping time.

LEMMA 13 *For all $k > 0$, if $k \bmod 4 = 1$, then $T(k)$ is even.*

PROOF: Suppose $k > 0$ and $k \bmod 4 = 1$. Then $k \bmod 2 = 1$, and hence $T(k) = (3k+1)/2$. But $(3k+1)$ is evenly divisible by 4:

$$\begin{aligned} k \bmod 4 = 1 &\Rightarrow 3k \bmod 4 = 3 && (13)\\ &\Rightarrow (3k+1) \bmod 4 = 0, && (14) \end{aligned}$$

and therefore $(3k+1)/2$ must be evenly divisible by 2. ∎

COROLLARY 14 *For all $k > 1$, if $k \bmod 4 = 1$, then $\sigma(k) = 2$.*

PROOF: Suppose $k > 1$. Then $3k + k > 3k + 1$, so $k > (3k + 1)/4$. By the above lemma, $T(k) = (3k + 1)/2$. But $(3k + 1)/2 > k$ and $(3k + 1)/2$ is even, so $T^{(2)}(k) = (3k + 1)/4$. So by definition, the stopping time of k is 2. ∎

Sad to say, the first author's search for peaks in stopping time never used the above idea, which would have resulted in a factor of 1.25 speedup (over and above ignoring the even numbers).

A fruitful idea for finding *a priori* cutoffs is to see how a number can result from (smaller) numbers in the course of iterating H or T. This should be contrasted with the techniques used above to find cutoffs in stopping time, which see what happens to the number itself when it is used as input to iterations of T. The idea of looking at convergence between the trajectories of smaller numbers and the number in question is related to the *Collatz graph* discussed in [5].

LEMMA 15 *Let j and k be given so that $0 < j < k$. If there is some $m > 0$ such that $H^{(m)}(j) = k$, then k cannot be a peak in* steps *or* max_value.

PROOF: Since the steps taken by k are the same as those taken by j after m initial steps, $steps(j) = m + steps(k)$ and $max\_value(j) \geq max\_value(k)$. ∎

LEMMA 16 *Let j and k be given so that $0 < j < k$. If there is some $m > 0$ such that $T^{(m)}(j) = k$, then k cannot be a peak in stopping time, total stopping time, or* alt_max_value. ∎

The most important practical example of this kind of *a priori* cutoff is that if $k \bmod 6 = 5$, then k cannot be a peak in any of the statistics mentioned above. This is because if $k \bmod 6 = 5$, then k lies on the trajectory of $(2k-1)/3$, which is smaller than k. Indeed the iterates of H first multiply $(2k-1)/3$ by 3 and add 1, obtaining $2k$, and then divide $2k$ by 2 obtaining k. This result is proved in the following lemma.

LEMMA 17 *Let $k > 0$. If $k \bmod 6 = 5$, then $T((2k-1)/3) = k$ and $H^{(2)}((2k-1)/3) = k$.*

PROOF: Suppose $k > 0$ and $k \bmod 6 = 5$. First we note that $2k - 1$ is divisible by 3:

$$\begin{aligned} k \bmod 6 = 5 &\Rightarrow 2k \bmod 6 = 4 && (15)\\ &\Rightarrow 2k \bmod 3 = 1 && (16)\\ &\Rightarrow (2k-1) \bmod 3 = 0. && (17)\end{aligned}$$

To see where T or H maps $(2k-1)/3$ we must know if $(2k-1)/3$ is even or odd:

$$\begin{aligned} k \bmod 6 = 5 &\Rightarrow 2k \bmod 6 = 4 && (18)\\ &\Rightarrow (2k-1) \bmod 6 = 3 && (19)\\ &\Rightarrow (2k-1)/3 \bmod 2 = 1. && (20)\end{aligned}$$

The last implication above follows because there is some integer q such that:

$$\begin{aligned} (2k-1) = 6q+3 &\Rightarrow (2k-1) = 3(2q)+3 && (21)\\ &\Rightarrow (2k-1)/3 = 2q+1 && (22)\end{aligned}$$

Since $(2k-1)/3$ is odd, according to the definitions of T and H,

$$\begin{aligned} T((2k-1)/3) &= k && (23)\\ H((2k-1)/3) &= 2k && (24)\\ H^{(2)}((2k-1)/3) &= k. && (25)\end{aligned}$$

∎

COROLLARY 18 *Let $k > 0$. If $k \bmod 6 = 5$, then k cannot be a peak in* steps, max_value, *stopping time, total stopping time, or* alt_max_value. ∎

Thus the search programs ignore odd numbers that are equal to 5 modulo 6, for an factor of 1.33 speedup (over a program that ignores even numbers).

The reader might see what happens when $k \bmod 18 = 13$. This idea can be carried as far as one desires. For example, one could keep a table of which numbers modulo 216 cannot be peaks in *max_value* or *steps*, and a counter that gives the value of the current iterate modulo 216. One could then use the table to avoid testing numbers that have no hope of being peaks. In the extreme, one can organize the entire search by constructing the Collatz graph, but the space requirements become prohibitive.

Other *a priori* cutoffs are discussed below, after the introduction of composite polynomials.

2.2. A Posteriori Cutoffs

When iterating H to search for peaks in *max_value*, one has to check periodically to see if the values produced are greater than the value of the previous iterate (or greater than the value of the previous peak). However, these comparisons are fairly expensive for large precision numbers. It should be obvious that one does not have to make a comparison after dividing by 2, since the next iterate is smaller than the last. Neither does one have to make a comparison after every $3n+1$ step of iterating H, but only until the iterates have fallen below the initial value (stopped); this is due to the following result.

LEMMA 19 *Let $k > 0$ be a peak in* max_value. *If for some $m > 0$, $H^{(m)}(k) < k$, then $max\_value(k) = \max\{H^{(i)}(k) \mid 0 \leq i \leq m\}$.*

PROOF: Let $m > 0$, be such that $H^{(m)}(k) = j < k$. Since $j < k$, $max\_value(j) < max\_value(k)$, because k is a peak. Since after this point the H-trajectory of k is the same as that taken by j, it cannot be the case that more iterations will reach or exceed the maximum value obtained up to this point. ∎

The way this lemma is used in an *a posteriori* cutoff is to stop making comparisons for purposes of finding a peak in *max_value* after the value of the iterates falls below the initial input number, k. Note that the lemma depends on k being a peak in *max_value*. If this is not the case, the maximum value may be obtained after the value of an iterate falls below its starting value. An example is the number 55, which reaches a value of 376 before it first falls below 55 (to 47). It then goes on to reach a maximum value of 9,232.

After the value of the iterates has fallen below the input value, one can cut off the search for peaks in *steps* (or total stopping time) *a posteriori* using the following lemma to estimate the maximum number of further iterations that will be needed.

LEMMA 20 *For all $k > 0$, if $H^{(p)} = n \leq j$, where j is a peak in* steps, *then $steps(k) \leq p + steps(j)$*

PROOF: If $n = j$, then $steps(k) = p + steps(j)$. If $n < j$, then, since j is a peak in *steps*, $steps(n) < steps(j)$, by the definition of a peak. ∎

A similar results holds for T and σ_∞.

In practice, the above lemma is used as follows. Let the number to be tested be k. After a step where H divides the current iterate value by two, one finds (if possible) the largest peak, j, in *steps* such that the current iterate's value is no greater than j, and uses the lemma above to bound $steps(k)$. If this bound on $steps(k)$ indicates that k is not a peak in steps, then k can be dismissed as far as *steps* is concerned.

The importance of this *a posteriori* cutoff is the empirical observation that the cutoff allows the search for peaks in *steps* to be cut off, on the average, after a constant number of steps. This seems to be true in any sufficiently large interval, provided that all but one or two peaks in *steps* less than the interval are known. That is, if one knows all the peaks in steps up to j, and if j is sufficiently large, then over the interval from $j+1$ to $2j$ one should always be able to cut off the search after an average of about 15 steps, independent of the value of j.[1] Considering

[1] We recorded data for the 100,000 odd numbers in the interval from 17,828,259,369 to 17,828,459,369. Note that 17,828,259,369 is a peak in *steps*. In this interval the average number of steps is 276.

```
% input:  an integer n > 0
% output:  number of steps and max_value reached
steps:  int := 0
max_value:  int := n
while n ~= 1 do
      if (n // 2) = 0
      then n := n / 2
      else n := 3*n + 1
           max_value := int$max(max_value, n)
      end
      steps := steps + 1
end
```

Figure 1: The hailstone algorithm, which computes iterates of H.

that the number of steps taken by n goes up logarithmically with the n, this cutoff makes sense as soon as the average (odd) number starts taking more than 15 steps worth of time plus the time required to see if the search can be cut off. Furthermore, the value of this cutoff grows as the search proceeds towards infinity. (The first author used this cutoff instead of some of cutoffs based on the polynomials discussed below.)

2.3. Speeding Up the Iterations

Even with the results above, there are still infinitely many trajectories that must be computed, at least in part, for a search. Thus these trajectories must be computed efficiently.

The hailstone algorithm in Figure 1, which finds the values of $steps(n)$ and $max\_value(n)$ is coded in Argus [8]. This algorithm is does not save the entire H-trajectory of n, but just records the values of the statistics. (In the code `%` starts a comment, `//` is a modulo operator and `~=` means "not equal". In the real programs, `n` would not be an `int`, it would be an object of some type of large-precision natural numbers.)

The problem considered in this sub-section is finding an equivalent algorithm that can be executed in less time. The focus in this section is on the hailstone algorithm, and the search for peaks in *steps* and *max_value*.

2.3.1. Make_odd

Division by 2 is best implemented by shifting in a binary representation. Also, shifting a number by several bit positions is roughly as fast as shifting a number by one bit position. Thus, one idea for making a faster algorithm is to replace the division by 2 step in the hailstone algorithm by a step that shifts the input as many bits as necessary in order to make it odd. How effective will this be? If the value of the hailstone algorithm's variable n were uniformly distributed among the even integers by the $3n+1$ step, then half of the time n would not be divisible by 4, and one fourth of the time n would not be divisible by 8, and so on. Thus the expected number of bit positions that an even number would be shifted is:

$$1/2 + 2(1/4) + 3(1/8) + 4(1/16) + \cdots \quad = \quad \sum_{i=1}^{\infty} \frac{i}{2^i} = 2. \tag{26}$$

Thus on the average, shifting n by as many bits as necessary to make it odd does the work of two divisions by 2. It is therefore cost-effective if it takes no more than the time taken by doing two divisions.

An advantage of shifting n so that it is odd is that one no longer has to check to see whether n is odd or even, because one can write the hailstone algorithm (for odd inputs) as in Figure 2.

```
% input:  an odd integer n > 0
while n ~= 1 do
      % n is odd
      n := 3*n + 1
      % n is even
      % check for max_value here
      n, p := make_odd(n)
      % the number of steps taken this time around the loop is p+1
      % check for a posteriori cutoffs here
end
```

Figure 2: Hailstone algorithm using *make_odd*.

```
% input:  integers n > 0, m > 0
while n ~= 1 do
      p, s := mBitPoly(n // 2**m)
      n := polyEval(p, n)
      % the number of steps taken this time around the loop is s
      % check for a posteriori cutoffs
end
```

Figure 3: Hailstone algorithm using composite polynomials.

In the figure, the procedure *make_odd* returns the new value of n and the number of bit positions that the old value of n had to be shifted to make it odd. Another pleasing property of this hailstone algorithm is that one can check for cutoffs when n is as low as it can be before going up again, this means that one spends less time checking for cutoffs, on the average.

2.3.2. Composite Polynomials

A more efficient hailstone algorithm than the above takes bigger steps, doing several iterates at a time. This idea gives a speedup of 7 over the hailstone algorithm of Figure 1. It also leads to several strategies for cutoffs.

The standard hailstone algorithm looks at the last bit of the value of the variable n to decide what step to take. By looking at the last m bits of the binary representation of n, one can decide what the next several steps that will be taken are, combine all these steps into a polynomial, and then do the work of all those steps by evaluating the polynomial at the value of n. An algorithm that uses this idea for computing the iterates of H is shown in simplified form in Figure 3. In the figure, *mBitPoly* returns both a polynomial and the number of steps that the polynomial represents. Checking for *max_value* is described below.

There are two strategies for expressing the polynomial.

One strategy is to obtain a polynomial of the form:

$$\frac{3^k x + z}{2^m}$$

which is equivalent to the sequence of $k + m$ steps taken. For example, for $n = 7$, the first step is to multiply by 3 and add 1 (obtaining 22), the second divides by 2 (obtaining 11), the third multiples by 3 and adds 1 (obtaining 34), and the forth divides by 2 (obtaining 17). These steps are represented by the following polynomial.

$$\frac{1}{2}\left(3\left(\frac{3x+1}{2}\right)+1\right) = \frac{1}{2}\left(\frac{9x+3}{2}+1\right) = \frac{9x+5}{4} \tag{27}$$

Such a polynomial is called the *standard polynomial*, it represents k steps of the form $3n+1$ and m divisions by 2. (It will be shown below why the number of divisions by 2 is equal to the number of bits considered.) For each n, the standard m bit polynomial for n will be written $Spoly_m(n)$. This polynomial can be evaluated by multiplying by the appropriate power of 3, adding in the appropriate integer z and then shifting by the appropriate power of 2. As in the previous section one ends up with an odd number.

A disadvantage of the standard polynomial is that the intermediate result after multiplying by the appropriate power of 3 is rather large, possibly leading to inefficient use of storage or inadvertent overflow. The second strategy overcomes this problem by doing the shifting first, and this is the strategy used by the second author's programs. The idea is to represent the steps by a "polynomial" of the following form.

$$\left\lfloor \frac{x}{2^m} \right\rfloor 3^k + y$$

This *Vermeulen polynomial* also represents k steps of the form $3n+1$ and m divisions by 2. For each n, the m bit Vermeulen polynomial for n will be written $Vpoly_m(n)$. Vermeulen polynomials will yield an answer equivalent to the normal sequence of steps when evaluated in the following manner:

1. divide by 2^m and truncate, that is, shift the binary representation right by m bits,
2. multiply the result (the most significant part of n) by the appropriate power of 3, and
3. add y.

The idea is that a Vermeulen polynomial represents the effect of the steps on the most significant part of n only. The number y represents the overflow from the least significant m bits.

To explain how these polynomials are generated, consider the following examples. Items to the right of the dot (.) in last bits do not affect the other bits, and items to the left of the dot do. The following is the generation of an 8 bit Vermeulen polynomial, $Vpoly_8(3)$.

Max	Last bits	N	Partial poly	Next step
	.00000011	3	x	*3
	.00001010	10	3x+1	/2
	.0000101	5	3(x/2)+2	*3
*	.0010000	16	9(x/2)+7	/16
	.001	1	9(x/32)+1	*3
	.100	4	27(x/32)+4	/4
	.1	1	27(x/128)+1	*3
	10.0	4	81(x/128)+2	/2
	10.	2	81(x/256)+2	

Notice that division by 2 removes one bit from the right, this corresponds to moving in bits from the left that are unknown. The process of computing partial polynomials stops when all the known bits are shifted out, that is, when m divisions by 2 have been performed. The row with the asterisk (*) marks the partial polynomial that produces the largest intermediate result. If one is checking for peaks in *max_value*, then the evaluation must be broken into three steps: evaluating the partial polynomial at this point, checking for a peak in *max_value*, and then evaluating the rest of the polynomial. For comparison, $Spoly_8(3)$ is $(81x+269)/256$.

As another example, $Vpoly_8(27)$ is $\lfloor x/256 \rfloor 2187 + 242$; for comparison $Spoly_8(27)$ is $(2187x + 2903)/256$.

The composite polynomial idea leads to a hailstone algorithm that is about an order of magnitude faster than the algorithm in Figure 2. It is an open question whether this approach is optimal.

The practical drawback to such an algorithm is its complexity of implementation. One needs automated tools for generating the polynominals. Furthermore, it is difficult to write the code to check for peaks in *max_value* or stopping time. This implementation difficulty makes the results of the algorithm less reliable (as the implementation is more difficult to verify).

Polynomial Width (bits)	Values	Steps
4	81.25%	43.75%
8	92.57%	59.76%
12	94.48%	67.67%
16	96.78%	73.98%

Table 3: Effectiveness of A Priori Cutoffs based on Polynomials, percentage of numbers eliminated

2.4. A Priori Cutoffs based on Composite Polynomials

Even if composite polynomials are not used in the iteration algorithm, they can be used to generate *a priori* cutoffs.

The second author's program uses the following lemma for *a priori* cutoffs in the search for peaks in *steps*. It allows one to ignore numbers k such that $k \bmod 2^m$ has a m-bit Vermeulen polynomial less that is the same as some number smaller than $k \bmod 2^m$. The potential benefits of this optimization are listed in Table 3, showing that 60% of all numbers may be eliminated using an 8 bit wide table. Increasing the width of the cutoff polynomial can further speed up the search.

LEMMA 21 *Let $m > 0$ be given. Let $0 < r_1 < r_2 < 2^m$, $Vpoly_m(r_1) = Vpoly_m(r_2)$. Then there is no $k \geq 2^m + r_1$ such that $k \bmod 2^m = r_2$ and k is a peak in* steps.

PROOF: Let $k \geq 2^m + r_1$ be such that $k \bmod 2^m = r_2$. Let b be the largest number such that $b < k$ and $b \bmod 2^m = r_1$. Since the polynomials $Vpoly_m(r_1)$ and $Vpoly_m(r_2)$ are identical they represent the same number of steps. After these steps, b and k take the same steps since $(Vpoly_m(r_1))(b) = (Vpoly_m(r_2))(k)$. So k cannot be a peak, because $b < k$ and $steps(b) = steps(k)$. ∎

The second author's program cuts off the search for peaks in *max_value a priori* if the m bit Vermeulen polynomial has as the coefficient of x a term that is less than unity. For example, using 8 bit polynomials, one need not look for peaks in values when the last 8 bits are ".00000011," because $Vpoly_8(3) = \lfloor x/256 \rfloor 81 + 2$ and $81/256 < 1$ (where the coefficient is $81/256$). Another example: one *does* look for peaks in values when the last 8 bits are ".00011011" because $Vpoly_8(27) = \lfloor x/256 \rfloor 2187 + 242$.

The potential benefits of this optimization are listed in Table 3, showing that 92% of all numbers may be eliminated using an 8 bit wide table. The actual benefits are smaller because the eliminated numbers are also detected by an A Posteriori cutoff based on lemma 19.

This kind of cutoff is formalized using the notion of the "largest" partial polynomial. A *partial polynomial* is a polynomial incorporating the first $s \geq 0$ steps of the hailstone algorithm, as predicted (see above) from the least significant m bits. We say that $p \geq q$ for m bit partial polynomials if there is some $N > 0$ such that, for all $n > N$, $p(n) \geq q(n)$.

Let $LVpoly_m(r)$ be the largest m bit partial Vermeulen polynomial for r. Similarly, let $LSpoly_m(r)$ be the largest m bit partial standard polynomial for r.

LEMMA 22 *Let $m > 0$ be given. Let $poly_m(r)$ and $Lpoly_m(r)$ denote either the standard or Vermeulen m bit polynomial and largest m bit partial polynomial for r.*

There is some $N > 0$ such that for all $k > N$, if $k \bmod 2^m = r$ and

1. *$poly_m(r)(k) < k$, and*
2. *there is some* max_value *peak $j < k$ such that $Lpoly_m(r)(k) \leq max\_value(j)$,*

then k is not a peak in max_value.

PROOF: Choose N such that for all $k > N$ and for all predicted m bit partial polynomials, p and q, $p \geq q$ implies $p(k) \geq q(k)$. Now, given $k > N$, the highest value an iterate reaches before falling

below its initial value, k, is $Lpoly(r)(k)$, where $r = k \bmod 2^m$. This is because $(poly(r))(k) < k$, and so n falls below k (after the number of steps represented by the polynomial) and because by construction $(Lpoly(r))(k)$ is the highest value that n attains (in these first steps). Thus by Lemma 19, if k is a peak in *max_value*, then

$$(Lpoly(r))(k) = max\_value(k) < max\_value(j)$$

which is a contradiction. So k cannot be a peak in *max_value*. ∎

Note that the above lemma is independent of the kind of polynomial used. In practice, the choice of N is not a problem, because the input numbers soon dwarf the coefficients of the composite polynomials.

The above lemma allows one to set up a table that tells which numbers modulo 2^m need to be checked for peaks in *max_value*. For each $0 < r < 2^m$, one must check the two conditions of the above lemma. For sufficiently large numbers, one can check the first condition of the lemma above *a priori*.

One can also satisfy the second condition of the lemma *a priori* over a certain interval. Suppose N is chosen to be the least number to satisfy the lemma and such that the first condition of the lemma can be checked *a priori*. Suppose the largest known peak in *max_value*, call it j, is such that $j > N$. Finally, suppose that there is a rational number $R > 0$ such that, for all $k > N$ and for all $0 \le r \le 2^m$ such that $(poly_m(r))(k) < k$:

$$(LVpoly_m(r))(k) < R \cdot k.$$

Then in the interval from N to $max\_value(j)/R$ the second condition of the lemma can be checked *a priori*. In fact, in such an interval, the second condition of the lemma is satisfied whenever the first condition of the lemma is satisfied.

Thus the practical justification for setting up a table of cutoffs as described above is that, for sufficiently large input numbers, the maximum value of the largest peak is many orders of magnitude greater than the inputs numbers and the coefficients of the Vermeulen polynomial cannot be very large. In fact the following result holds.

LEMMA 23 *Let $m > 1$ be given. For all $0 < r < 2^m$, if there is some M_r such that for all $k > M_r$, $k \bmod 2^m = r$ implies that $(Vpoly_m(r))(k) < k$, then the coefficient of x in $LVpoly_m(r)$ is at most $3^j/2^{(j-1)}$, where j is the largest integer such that $3^j < 2^m$.*

PROOF: Every step in the hailstone algorithm of the form $3n + 1$ is followed by a step that divides by 2. Thus the coefficient of x in the partial Vermeulen polynomial is made as large as possible when it is of the form $3^j/2^{(j-1)}$. However, because the final polynomial is of the form $\lfloor x/2^m \rfloor 3^{j+p} + y$, for some $p \ge 0$, 3^j must be less than 2^m if the Vermeulen polynomial is to be such that $Vpoly_m(r)(k) < k$, for all sufficiently large $k \bmod 2^m = r$. ∎

As an example, for 8 bit Vermeulen polynomials, the coefficient can be at most

$$3^5/2^4 = 243/16 = 15.1875.$$

Thus, as long as the is the largest peak j in *max_value* is at least 16 times the input numbers being checked, then for all sufficiently large k:

$$(Vpoly_8(r))(k) < k \Rightarrow (LVpoly_8(r))(k) \le max\_value(j).$$

This condition seems to be met for all numbers past 7, although we have no proof of this conjecture.

3. Conclusions

The optimizations described in Section 2. illustrate an old story: mathematical insight can greatly improve the efficiency of a program. Both authors expected that the C program would be faster

because of low-level coding issues; the Argus compiler did almost no optimization, and C allows one to write fast programs. There were some speedups from using faster integer arithmetic in C, but these were soon dwarfed by the algorithmic speedups. Even though Argus code is perhaps 5 to 10 times slower than C, the algorithmic improvements give a cumulative speedup by a factor of 182. (Both programs used assembly language for time-critical parts, such as arbitrary precision integer arithmetic. However, our comments about the relative speeds of Argus and C are still accurate, because in Argus integers must be decoded before they can be processed by the CPU's native instruction set, and then encoded on return.)

Our optimizations can be analyzed in terms of Bentley's taxonomy of program efficiency improvement strategies [1]. This comparison shows that we used some standard strategies in interesting ways.

Bentley's logic rule 1 "exploit algebraic identities" states that: "If the evaluation of a logical expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate" [1, page 148]. (This strategy is often called "strength reduction" when one replaces the expression with one entirely equivalent.) We used this strategy several times. In Section 2.1. we noted that it was not necessary to test even numbers for peaks; this is similar to Bentley's example of eliminating the square root in a distance calculation, since all that mattered for the program was the relative distances. Another use of this strategy is in the *a priori* cutoffs based on convergence of trajectories, also described in Section 2.1..

Our optimization of not making a comparison for peaks in *max_value* after a step of division by 2, described in Section 2.2., is similar to Bentley's logic rule 5 "boolean variable elimination." Bentley remarks that his rule can be generalized to the idea of storing arbitrary conditions in the program counter [1, page 73]. We like to think that instead of storing conditions in the program counter, one pushes desired assertions about the program state back through earlier statements. That is, at the end of the loop, we want to assert that we have recorded the largest peak in *max_value*; in the case where the number was divided by 2, this is automatically satisfied. This strategy was also applied at lower levels of coding. For example, in the C program when the iterates take on a value that will fit in a 32 bit integer, faster code is used that avoids the overhead needed for a complete arbitrary precision integer computation.

We described a similar optimization in Lemma 19. This lemma allows us to not check for a peak in *max_value* after the iterates have fallen below the initial value (stopped). This can also be thought of as an application of Bentely's logic rule 5. However, another way to think of it is an application of Bentley's loop rule 1 "code motion out of loops," because the tests for a peak in *max_value* cannot succeed after the iterates have stopped.

In Section 2.2. we described an *a posteriori* cutoff in the search for peaks in *steps* by estimating the maximum number of steps that an iterate can take to reach 1. We do this by using precomputed results, the value of *steps* for all known peaks. This is an application of Bentley's space-for-time rule 2 "store precomputed results". (This strategy is classically called "dynamic programming" [3, Chapter 5].) To see this as an instance of Bentley's strategy, one has to imagine that our search algorithms was originally recursive, checking the precomputed results on each call. This recursive process is then transformed to an iterative one (Bentley's procedure rule 4). Then as above, one notes that the check only has to be made after division by 2.

The main idea behind our first faster iteration algorithm, discussed in Section 2.3.1., can be thought of as Bentley's expression rule 2 "exploit algebraic identities". The algebraic identity that a division by a power of two can be implemented by a shift. Since after the shift it is known that the result is odd, Bentley's logic rule 5 "boolean variable elimination" is used remove the test to see whether the number is odd. This also allows the remaining tests to be done less often, as described above.

The composite polynomials, discussed in Section 2.3.2., are an application of Bentley's space for time rule 2 "store precomputed results". However, Bentley does not describe this kind of use of precomputed results — taking one large step instead of several small steps. As Bentley notes, applying one optimization can often pave the way for others; this was certainly true of using the polynomials. For example, we also used the polynomials to invent *a priori* cutoffs. As Table 3 shows, increasing the bits considered in the polynomials used in the *a priori* cutoffs trades an increase in space for an increase the effectiveness of the cutoff (hence decreased time).

We also used Bentley's expression rule 4 "pairing computation" to search for peaks in *steps* and *max_value* at the same time. Finally, we also used his procedure rule 5 "parallelism" extensively, although we did not discuss that aspect of our programs in Section 2., see [7].

A final conclusion is that even with the ability to distribute the search programs on many fairly fast computers, and even with the ability to run such programs for years (literally), algorithmic improvements are necessary to achieve interesting results. Finding a peak becomes more and more rare as the search progresses towards infinity. To maintain interest in such a program, one must make continual improvements in its speed, so that the results appear at a more or less constant rate. Without such algorithmic improvements as described above, the programs would never have been able to search into the trillions within our lifetimes.

That such a simple problem could exhibit such interesting mathematics was wholly surprising and a source of great pleasure.

Acknowledgements

The work described in this report was done while the authors were graduate students at MIT, and hence was supported indirectly by MIT, the Laboratory for Computer Science, and in particular by professors Barbara Liskov and Bill Weihl.

While the Argus group has benefited from having a running application, Paul Johnson has been helpful beyond the call of duty in fixing problems with the Argus system (and our programs!) in a timely manner. Paul has also helped with assembly language programming. We also thank all the members of the Argus group for putting up with the hailstone system with such good grace.

Thanks also to Kelvin Nilsen who helped correct an earlier draft.

A Tables of Peaks

This appendix contains tables of results from the various search programs.

A1. Peaks in more than one Statistic

A few numbers have been found to be peaks in several statistics. In a sense, these are the most interesting numbers we found.

Tables 8, 9, and 10 list the peaks in both *steps* and total stopping time; these are known to be identical up to 12.3 billion (12.3×10^9).

Table 4 lists numbers that are peaks in other combinations of statistics; each entry is filled in if the number is a peak in the corresponding function and left empty otherwise.

Table 4: Peaks in more than one statistic.

n	steps(n)	$\sigma(n)$	max_value(n)
1	0	0	1
2	1	1	2
3	7	4	16
7	16	7	52
27	111	59	9,232
703	170	81	250,504
26,623	307		106,358,020
270,271		164	24,648,077,896
626,331	508	176	
63,728,127	949	376	
12,235,060,455	1,184	547	

A2. Maximum Values

The peaks in *max_value* are listed in Tables 5 and 6. These tables are complete up to 56 trillion (5.6×10^{13}). The peaks up to 100 billion (100×10^9) have been verified by two programs. Mike Vermeulen's program found all the peaks above 100 billion.

Of particular interest here are the peaks at 27, 6,631,675, 319,804,831, and 3,716,509,988,199. Also listed in the tables is the ratio of the peak's maximum value reached to the previous maximum value reached (labeled "ratio") and the approximate expansion factor (labeled $s(n)$), where $s(n)$ is $max\_value(n)/(2n)$.

Table 5: Peaks in *max_value* up to n = 5,000,000.

n	$max\_value(n)$	ratio	$s(n)$
1	1		0.5
2	2	2.0	0.5
3	16	8.0	2.7
7	52	3.3	3.7
15	160	3.1	5.3
27	9,232	57.7	1.7×10^2
255	13,120	1.4	2.5×10^1
447	39,364	3.0	4.4×10^1
639	41,524	1.1	3.2×10^1
703	250,504	6.0	1.8×10^2
1,819	1,276,936	5.1	3.5×10^2
4,255	6,810,136	5.3	8.0×10^2
4,591	8,153,620	1.2	8.9×10^2
9,663	27,114,424	3.3	1.4×10^3
20,895	50,143,264	1.8	1.2×10^3
26,623	106,358,020	2.1	2.0×10^3
31,911	121,012,864	1.1	1.9×10^3
60,975	593,279,152	4.9	4.9×10^3
77,671	1,570,824,736	2.6	1.0×10^4
113,383	2,482,111,348	1.6	1.1×10^4
138,367	2,798,323,360	1.1	1.0×10^4
159,487	17,202,377,752	6.1	5.4×10^4
270,271	24,648,077,896	1.4	4.6×10^4
665,215	52,483,285,312	2.1	3.9×10^4
704,511	56,991,483,520	1.1	4.0×10^4
1,042,431	90,239,155,648	1.6	4.3×10^4
1,212,415	139,646,736,808	1.5	5.8×10^4
1,441,407	151,629,574,372	1.1	5.3×10^4
1,875,711	155,904,349,696	1.0	4.2×10^4
1,988,859	156,914,378,224	1.0	3.9×10^4
2,643,183	190,459,818,484	1.2	3.6×10^4
2,684,647	352,617,812,944	1.9	6.6×10^4
3,041,127	622,717,901,620	1.8	1.0×10^5
3,873,535	858,555,169,576	1.4	1.1×10^5
4,637,979	1,318,802,294,932	1.5	1.4×10^5

Table 6: Peaks in max_value from n = 5,000,000 upwards.

n	$max\_value(n)$	ratio	$s(n)$
5,656,191	2,412,493,616,608	1.8	2.1×10^5
6,416,623	4,799,996,945,368	2.0	3.7×10^5
6,631,675	60,342,610,919,632	12.6	4.5×10^6
19,638,399	306,296,925,203,752	5.1	7.7×10^6
38,595,583	474,637,698,851,092	1.5	6.1×10^6
80,049,391	2,185,143,829,170,100	4.6	1.4×10^7
120,080,895	3,277,901,576,118,580	1.5	1.4×10^7
210,964,383	6,404,797,161,121,264	2.0	1.5×10^7
319,804,831	1,414,236,446,719,942,480	220.8	2.2×10^9
1,410,123,943	7,125,885,122,794,452,160	5.0	2.5×10^9
8,528,817,511	18,144,594,937,356,598,024	2.5	1.1×10^9
12,327,829,503	20,722,398,914,405,051,728	1.1	8.4×10^8
23,035,537,407	68,838,156,641,548,227,040	3.3	1.5×10^9
45,871,962,271	82,341,648,902,022,834,004	1.2	9.0×10^8
51,739,336,447	114,639,617,141,613,998,440	1.4	1.1×10^9
59,152,641,055	151,499,365,062,390,201,544	1.3	1.3×10^9
59,436,135,663	205,736,389,371,841,852,168	1.4	1.7×10^9
70,141,259,775	420,967,113,788,389,829,704	2.0	3.0×10^9
77,566,362,559	916,613,029,076,867,799,856	2.2	5.9×10^9
110,243,094,271	1,372,453,649,566,268,380,360	1.5	6.2×10^9
204,430,613,247	1,415,260,793,009,654,991,088	1.0	3.4×10^9
231,913,730,799	2,190,343,823,882,874,513,556	1.5	4.7×10^9
272,025,660,543	21,948,483,635,670,417,963,748	10.0	4.0×10^{10}
446,559,217,279	39,533,276,910,778,060,381,072	1.8	4.4×10^{10}
567,839,862,631	100,540,173,225,585,986,235,988	2.5	8.8×10^{10}
871,673,828,443	400,558,740,821,250,122,033,728	4.0	2.3×10^{11}
2,674,309,547,647	770,419,949,849,742,373,052,272	1.9	2.9×10^{11}
3,716,509,988,199	207,936,463,344,549,949,044,875,464	269.9	5.6×10^{13}
9,016,346,070,511	252,229,527,183,443,335,194,424,192	1.21	2.7×10^{13}

A3. Steps and Total Stopping Time

Peaks in *steps* are listed in Tables 8, 9, and 10. These tables are complete up to 56 trillion (5.6×10^{13}). The peaks up to 100 billion (100×10^9) have been verified by two programs. Mike Vermeulen's program found all the peaks above 100 billion. It seems that every peak in *steps* is also a peak in total stopping time, σ_∞, although we have only verified this conjecture up to 12.3×10^9. That is tables 8, 9, and 10 also contain all peaks in total stopping time up to 12.3 billion.

Also listed in these tables are the difference between each peak's number of steps and the previous peak's number of steps, the value of the stopping time $\sigma(n)$, the value of $\sigma_\infty(n)$, and the maximum value reached. Many of the peaks in *steps* have the same *max_value*, and hence their trajectories are identical after a certain number of iterations. For example, $H^{(6)}(27) = 94 = H^{(10)}(73)$.

Of particular interest here are the peaks at n equal to 27, 63,728,127, and 3,743,559,068,799 which have large increments in the number of steps and the peaks that are simply twice the previous peak in steps, found in Table 7.

Table 7: Even Peaks in *steps*.

peak (n)	steps(n)
2	1
6	8
18	20
54	112
31,466,382	705
127,456,254	950
537,099,606	965
1,341,234,558	987
9,780,657,630	1,132
63,389,366,646	1,220
404,970,804,222	1,308
7,487,118,137,598	1,550

Table 8: Peaks in *steps* up to n = 100,000.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
1	0		0	0	1
2	1	1	1	1	2
3	7	6	4	5	16
6	8	1	1	6	16
7	16	8	7	11	52
9	19	3	2	13	52
18	20	1	1	14	52
25	23	3	2	16	88
27	111	88	59	70	9,232
54	112	1	1	71	9,232
73	115	3	2	73	9,232
97	118	3	2	75	9,232
129	121	3	2	77	9,232
171	124	3	5	79	9,232
231	127	3	12	81	9,232
313	130	3	2	83	9,232
327	143	7	21	91	9,232
649	144	1	2	92	9,232
703	170	26	81	108	250,504
871	178	8	35	113	190,996
1,161	181	3	2	115	190,996
2,223	182	1	8	116	250,504
2,463	208	26	21	132	250,504
2,919	216	8	26	137	250,504
3,711	237	21	37	150	481,624
6,171	261	24	58	165	975,400
10,971	267	6	8	169	975,400
13,255	275	8	8	174	497,176
17,647	278	3	73	176	11,003,416
23,529	281	3	2	178	11,003,416
26,623	307	26	65	194	106,358,020
34,239	310	3	92	196	18,976,192
35,655	323	13	135	204	41,163,712
52,527	339	16	18	214	106,358,020
77,031	350	11	89	221	21,933,016

Table 9: Peaks in *steps* from n = 100,000 to n = 5,000,000,000.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
106,239	353	3	97	223	104,674,192
142,587	374	21	24	236	593,279,152
156,159	382	8	37	241	41,163,712
216,367	385	3	83	243	11,843,332
230,631	442	57	73	278	76,778,008
410,011	448	6	75	282	76,778,008
511,935	469	21	16	295	76,778,008
626,331	508	39	176	319	7,222,283,188
837,799	524	16	105	329	2,974,984,576
1,117,065	527	3	2	331	2,974,984,576
1,501,353	530	3	2	333	90,239,155,648
1,723,519	556	26	176	349	46,571,871,940
2,298,025	559	3	2	351	46,571,871,940
3,064,033	562	3	2	353	46,571,871,940
3,542,887	583	21	180	366	294,475,592,320
3,732,423	596	13	8	374	294,475,592,320
5,649,499	612	16	116	384	1,017,886,660
6,649,279	664	52	146	416	15,208,728,208
8,400,511	685	21	214	429	159,424,614,880
11,200,681	688	3	2	431	159,424,614,880
14,934,241	691	3	2	433	159,424,614,880
15,733,191	704	13	8	441	159,424,614,880
31,466,382	705	1	1	442	159,424,614,880
36,791,535	744	39	34	466	159,424,614,880
63,728,127	949	205	376	592	966,616,035,460
127,456,254	950	1	1	593	966,616,035,460
169,941,673	953	3	2	595	966,616,035,460
226,588,897	956	3	2	597	966,616,035,460
268,549,803	964	8	5	602	966,616,035,460
537,099,606	965	1	1	603	966,616,035,460
670,617,279	986	21	18	616	966,616,035,460
1,341,234,558	987	1	1	617	966,616,035,460
1,412,987,847	1,000	13	8	625	966,616,035,460
1,674,652,263	1,008	8	13	630	966,616,035,460
2,610,744,987	1,050	42	46	656	966,616,035,460
4,578,853,915	1,087	37	81	679	966,616,035,460
4,890,328,815	1,131	44	135	706	319,497,287,463,520

Table 10: Peaks in *steps* from n = 5,000,000,000 upwards.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
9,780,657,630	1,132	1	1	707	319,497,287,463,520
12,212,032,815	1,153	21	15	720	319,497,287,463,520
12,235,060,455	1,184	31	547	739	1,037,298,361,093,936
13,371,194,527	1,210	26	62	755	319,497,287,463,520
17,828,259,369	1,213	3	2	757	319,497,287,463,520
31,694,683,323	1,219	6	7	761	319,497,287,463,520
63,389,366,646	1,220	1	1	762	319,497,287,463,520
75,128,138,247	1,228	8	7	767	319,497,287,463,520
133,561,134,663	1,234	6	10	771	319,497,287,463,520
158,294,678,119	1,242	8	15	776	319,497,287,463,520
166,763,117,679	1,255	13	35	784	319,497,287,463,520
202,485,402,111	1,307	52	270	816	2,662,567,439,048,656
404,970,804,222	1,308	1	1	817	2,662,567,439,048,656
426,635,908,975	1,321	13	40	825	2,662,567,439,048,656
568,847,878,633	1,324	3	2	827	2,662,567,439,048,656
674,190,078,379	1,332	8	5	832	2,662,567,439,048,656
881,715,740,415	1,335	3	329	834	5,234,135,688,127,384
989,345,275,647	1,348	13	165	842	1,219,624,271,099,764
1,122,382,791,663	1,356	8	16	847	2,662,567,439,048,656
1,444,338,092,271	1,408	52	202	879	1,219,624,271,099,764
1,899,148,184,679	1,411	3	72	881	1,037,298,361,093,936
2,081,751,768,559	1,437	26	606	897	79,988,992,024,030,705,960
2,775,669,024,745	1,440	3	2	899	79,988,992,024,030,705,960
3,700,892,032,993	1,443	3	2	901	79,988,992,024,030,705,960
3,743,559,068,799	1,549	106	65	966	79,988,992,024,030,705,960
7,487,118,137,598	1,550	1	1	967	79,988,992,024,030,705,960
7,887,663,552,367	1,563	13	70	975	79,988,992,024,030,705,960
10,516,884,736,489	1,566	3	2	977	79,988,992,024,030,705,960
14,022,512,981,985	1,569	3	2	979	79,988,992,024,030,705,960
19,536,224,150,271	1,585	16	327	989	3,813,091,869,769,158,724
26,262,557,464,201	1,588	3	2	991	79,988,992,024,030,705,960
27,667,550,250,351	1,601	13	10	999	79,988,992,024,030,705,960
38,903,934,249,727	1,617	16	211	1,009	1,180,174,841,128,253,392
48,575,069,253,735	1,638	21	13	1,022	1,180,174,841,128,253,392
51,173,735,510,107	1,651	13	21	1,030	1,180,174,841,128,253,392

A4. Stopping Time

Peaks in stopping time, σ, are not necessarily peaks in *steps*, and conversely peaks in *steps* are not necessarily peaks in stopping time. The same remark applies to total stopping time.

Table 11 lists the peaks in stopping time. The list is complete up to 6.8 trillion (6.8×10^{12}); however, only the peaks up to 1.043 trillion (1.043×10^{12}) are confirmed by two programs. The most interesting of these peaks is 12,235,060,455.

Also listed are the difference between each peak's stopping time and the stopping time of the previous peak (labeled "diff."), the value of *steps* for that peak, the total stopping time σ_∞, and the maximum value reached. Unlike the peaks in *steps*, the maximum values reached by these peaks rarely repeat.

Table 11: Peaks in stopping time, σ.

n	$\sigma(n)$	diff.	$steps(n)$	$\sigma_\infty(n)$	$max\_value(n)$
1	0		0	0	1
2	1	1	1	1	2
3	4	3	7	5	16
7	7	4	16	11	52
27	59	52	111	70	9,232
703	81	22	170	108	250,504
10,087	105	24	223	142	2,484,916
35,655	135	30	323	204	41,163,712
270,271	164	29	406	256	24,648,077,896
362,343	165	1	360	228	565,335,124
381,727	173	8	373	236	565,335,124
626,331	176	3	508	319	7,222,283,188
1,027,431	183	7	377	239	17,808,240,724
1,126,015	224	41	527	331	90,239,155,648
8,088,063	246	22	566	356	16,155,154,672
13,421,671	287	41	608	382	1,591,706,254,336
20,638,335	292	5	694	435	89,243,211,616
26,716,671	298	6	658	413	3,696,858,621,088
56,924,955	308	10	742	465	7,209,046,267,252
63,728,127	376	68	949	592	966,616,035,460
217,740,015	395	19	793	395	2,516,021,527,120
1,200,991,791	398	3	873	547	35,681,506,677,556
1,827,397,567	433	25	928	581	118,736,698,851,769,012
2,788,008,987	447	14	944	591	81,887,769,175,732
12,235,060,455	547	100	1,184	739	1,037,298,361,093,936
898,696,369,947	550	3	1,136	712	791,612,079,014,220,715,456
2,081,751,768,559	606	56	1,437	897	79,988,992,024,030,705,960

REFERENCES

1. J. L. Bentley. *Writing Efficient Programs.* Software Series. Prentice-Hall, Englewood Cliffs, N.J., 1982.
2. B. Hayes. Computer recreations: On the ups and downs of hailstone numbers. *Scientific American*, 250(1):10–16, Jan. 1984.
3. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Software Engineering Series. Computer Science Press, Inc., Potomac, Maryland, 1978.
4. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., Reading, Mass., 1969.

5. J. C. Lagarias. The 3x+1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, Jan. 1985.
6. J. C. Lagarias and A. Weiss. The $3x + 1$ problem: Two stochastic models. *Annals of Applied Probability*, 2, 1992. to appear.
7. G. T. Leavens. A distributed search program for the $3x + 1$ problem. Technical Report 89-22, Iowa State University, Department of Computer Science, Ames, Iowa, Nov. 1989.
8. B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. Argus reference manual. Technical Report 400, Massachusetts Institute of Technology, Laboratory for Computer Science, Oct. 1987. An earlier version appeared as Programming Methodology Group Memo 54 in March 1987.
9. B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst.*, 5(3):381–404, July 1983.
10. D. Rawsthorne. Imitation of an iteration. *Mathematics Magazine*, 58(3):172–176, May 1985.
11. S. Wagon. The collatz problem. *Mathematical Intelligencer*, 7(1):72–76, 1985.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 92-01
Submission Date: March 24, 1992