# The Complexity of Finding Periods

Robert Sedgewick*
*Brown University*
*Providence, Rhode Island*

Thomas G. Szymanski
*Bell Laboratories*
*Murray Hill, New Jersey*

**Abstract**

Given a function $f$ over a finite domain $D$ and an arbitrary starting point $x$, the sequence $x, f(x), f(f(x)), \ldots$ is ultimately periodic. Such sequences typically are used for constructiong random number generators. The *cycle problem* is to determine the first repeated element $f^n(x)$ in the sequence. Previous algorithms for this problem have required $3n$ operations. In this paper we present an algorithm which only requires $n(1 + O(1/\sqrt{M}))$ steps, if $M$ memory cells are available to store values of the function. By increasing $M$, this running time can be made arbitrarily close to the information-theoretic lower bound on the running time of any algorithm for the cycle problem. Our treatment is novel in that we explicitly consider the performance of the algorithm as a function of the amount of memory available as well as the relative cost of evaluating $f$ and comparing sequence elements for equality.

## 1. Introduction

Suppose that we are given an arbitrary function $f'$which maps some finite domain $D$ into $D$. If we take an arbitrary element $x$ from $D$ and generate the infinite sequence $f^0(x), f^1(x), f^2(x), \ldots$, then we are guaranteed by the "pigeonhole principle" and the finiteness of $D$ that the sequence becomes cyclic. That is, for some $l$ and $c$ we have $l + c$ distinct values $f^0(x), f^1(x), \ldots, f^{l+c-1}(x)$ but $f^{l+c}(x) = f^l(x)$. This implies, in turn, that $f^{i+c}(x) = f^i(x)$ for all $i \geq l$. The problem of finding this unique pair $(l, c)$ will be termed the *cycle problem* for $f$ and $x$. The integer $c$ is the *cycle length* of the sequence, and $l$ is termed the *leader length*. Similarly, the elements $f^l(x), f^{l+1}(x), \ldots, f^{l+c-1}(x)$ are said to form the *cycle* of $f$ on $x$ and $f^0(x), f^1(x), \ldots, f^{l-1}(x)$ are said to form the *leader* of $f$ on $x$. For notational convenience, the number $l + c$ of distinct values in the sequence will be denoted by $n$.

The cycle problem arises when analyzing the effectiveness of certain random number generators that produce successive "random" values by applying some function to the previous value in the sequence [1, Section 3.1]. Solving the cycle problem gives the number of distinct random numbers which are produced from a given seed. It is often possible to design functions that have null leaders and maximum cycles for all starting values, but such functions may be difficult to implement correctly. An algorithm for the cycle problem can be of use in checking the characteristics of an unknown random number generator.

One method for cycle detection has been given by Floyd [1, Exercise 3.1-7]. The idea is to have two variables taking on the values in sequence, one being advanced twice as fast as the other, as shown in the program of Figure 1.

* This work was supported in part by the National Science Foundation, grant number MCS75-23738.

```
$y \leftarrow z \leftarrow x$;
repeat
  $y \leftarrow f(y)$;
  $z \leftarrow f(f(z))$;
until $y = z$;
```

Figure 1. Floyd's algorithm.

This algorithm stops with $y = f^i(x) = f^{2i}(x) = z$, where $i$ is the smallest multiple of $c$ which is greater than or equal to $l$. If $l = c + 1$ then a total of $6c = 3(n-1)$ function evaluations are performed. This is particularly objectionable when the cost of evaluating $f$ is high relative to the cost of comparisons.

Another method, due to Gosper, *et. al.*, was designed to circumvent the overhead of advancing two independently operating "copies" of the generating function as required in Floyd's method. Their method is to save certain values of the sequence in a small table and to lookup each new value to see if it has previously been generated. The table update rule is to save the $i$th value generated in TABLE[$j$], where $j$ is the number of trailing zeroes in the binary representation of $i$. This method can require as many as $l + 2c$ function evaluations, which, in the worst case, can be $\frac{3}{2}n$. Moreover, it requires an equal number of table lookups, which is undesirable when the cost of comparisons is high relative to the cost of evaluating $f$.

These algorithms are suitable for detecting the existence of a cycle, and the value of $c$ can be found by proceeding around the cycle one additional time. Of course, this may be undesirable if $c$ is very large. Furthermore, neither algorithm has provision for directly finding $l$ except by starting back at the initial value.

In this paper, we develop an algorithm that solves the cycle problem using $n(1 + O(1/\sqrt{M}))$ function evaluations in the worst case, where $M$ is the amount of memory available for storing generated function values. The algorithm does $O(n/\sqrt{M} + M \log \frac{n}{M})$ memory operations. In Section 2, it is shown that any algorithm for the cycle problem requires at least $n$ function evaluations, so the new algorithm can be made as close to optimal as desired. Our algorithm is developed in Section 3 in two parts: one algorithm detects the cycle and a companion algorithm recovers the values of $l$ and $c$. The worst case analysis is given in Section 4. A generalization of the problem and some concluding remarks are offered in Section 5.

## 2. A LOWER BOUND

In this section we establish a lower bound on the complexity of the cycle problem by showing that any algorithm for the problem must generate each of the values on the leader and the cycle at least once. It should be emphasized here that we regard the function $f$ as a "black box". The only method for obtaining information about its behavior is by evaluating $f$ on points of $D$.

**Theorem 1.** Let $A$ be an algorithm for the cycle problem, and let $(f, x)$ be an instance of the cycle problem with solution $(l, c)$. Then $A$ evaluates $f$ at least $l + c$ times when run with input $(f, x)$.

*Proof:* Let $y_1, y_2, \ldots, y_i$ be the sequence of elements of the domain of $f$ on which $f$ is evaluated during $A$'s execution on $(f, x)$. (Clearly it must generate at least one value, so we may assume that $i > 0$ and $l + c > 1$ for the rest of the proof.) Since the only information that $A$ can obtain about a function is obtained by sampling the value of that function for various arguments, it must be that any function agreeing with $f$ on the $y$'s has the same period and leader as $f$.

However, if $i < l + c$ it is possible to find a contradiction by constructing a function $\overline{f}$ which agrees with $f$ on the $y$'s but which has a different leader. To construct such an $\overline{f}$, consider the sequence of elements $f^0(x), f^1(x), \ldots, f^{l+c-1}(x)$ which by definition of $l$ and $c$, contains $l + c$ distinct values. If $i < l + c$ then, by the "pigeonhole principle" there exists some $i_0$, with $0 < i_0 \leq l + c - 1$, for which $f^{i_0}(x)$ does not occur among the $y$'s. If $l = 0$, define $\overline{f}$ by

$\bar{f}(f^{i_0}(x)) = f(x)$ and $\bar{f}(z) = f(z)$ for $z \neq f^{i_0}(x)$. This function has a leader of 1, not zero. If $l > 0$, define $\bar{f}$ by $\bar{f}(f^{i_0}(x)) = x$ and $\bar{f}(z) = f(z)$ for $z \neq f^{i_0}(x)$. This function has a leader of 0, not greater.

The assumption $i < l + c$ implies that $A$ could not have solved the cycle problem, so we must have $i \geq l + c$. ∎

## 3. The algorithm

Theorem 1 says that any algorithm for the cycle problem must have a running time exceeding $nt_f$ where $t_f$ is the (assumed constant) time to perform one evaluation of $f$. Clearly, an algorithm could be designed which achieves a running time of $nt_f + O(n \log n)$ by employing, for example, a balanced tree scheme to save all the generated elements. Such an algorithm is unsatisfactory for at least two reasons. First, it is unrealistic to assume an unlimited supply of memory. Second, the analysis does not take into consideration the relative cost of evaluating $f$ and comparing two domain elements for equality. Let us therefore construct a framework in which these latter considerations can be addressed. We shall be particularly interested in the tradeoff between memory size and execution time.

Let TABLE be an associative store capable of storing up to $M$ pairs $(y, i)$ of domain elements and integers. Both elements of the pair are *keys* in the sense that at most one pair can occur in TABLE with a given first (or second) component. Let $t_u$ be the time needed to insert or delete a pair from the TABLE, i.e., to update the TABLE, and let $t_s$ be the time needed to search the TABLE for a given key. Depending on the implementation of TABLE, $t_u$ and $t_s$ might be constants, logarithmic functions of $M$ or even linear functions of $M$. (See Section 5.) As mentioned above, $t_f$ is assumed to be a constant, and all other operations of the algorithm are assumed to be free.

Within this model, we are ready to consider an efficient algorithm for the cycle problem. The idea is to limit the number of operations performed on TABLE by avoiding the store and lookup operations for most generated function values. Rather, these operations will be performed only sufficiently often to guarantee detection of the cycle. To this end, we shall introduce two parameters, $b$ and $g$. Figure 2 exhibits an algorithm which only stores every $b$th function value in TABLE and which only does lookups after every $g$ stores (at which point it does lookups on $b$ consecutive values).

```
$i \leftarrow 0$;
$y \leftarrow x$;
**repeat**
    **if** $i \equiv 0 \pmod{b}$ **then** *insert* $(y, i)$;
    $y \leftarrow f(y)$;
    $i \leftarrow i + 1$;
    **if** $i < b \pmod{gb}$ **then** *lookup* $(y, j)$;
**until** *found*;
```

Figure 2. Preliminary version of the algorithm.

Here the procedure *lookup* sets *found* to false if $y$ is not in TABLE, otherwise it sets *found* to true and $j$ to the minimum value of $j$ for which $(y, j)$ is in TABLE. The procedure *insert* puts $(y, i)$ into TABLE without checking to see if there is another entry with $y$ as the first component. The modulus computations in this program are used for clarity: an actual implementation would use simple counters instead.

The program must halt because once the cycle is reached, at least one value out of every block of $b$ consecutive values looked up must be in TABLE. It is possible for the algorithm to overshoot the point at which the cycle first returns to itself, but the algorithm will always detect the cycle before the $(n + gb)$th evaluation of $f$. The running time is thus bounded by

$$(n + gb)\left(t_f + \frac{t_s}{g} + \frac{t_u}{b}\right).$$

It is interesting to note that a dual algorithm can be developed by interchanging the roles of *lookup* and *insert* in Figure 2. Many of the results of this paper can be carried through for the dual algorithm as well (with the roles of $b$ and $g$ interchanged). In fact, for many search strategies, *insert* might be implemented by doing a *lookup* first, so it is tempting to contemplate an algorithm which involves only one operation on TABLE. However, the memory management and the analysis become quite complicated in this case, and it is convenient to keep the *lookup* and *insert* functions separated.

We could arrange to have the algorithm spend virtually all its time doing the (unavoidable) task of stepping $f$ by choosing $b$ and $g$ suitably, were it not for the fact that TABLE will soon fill up. Accordingly, we introduce the following memory management mechanism: whenever TABLE gets filled, remove every other entry from TABLE, double $b$ and continue. This has the same effect as restarting the program from the beginning with the larger value of $b$, with no additional function evaluations being required. The algorithm thus adapts its behavior to the problem at hand. The final version of the algorithm is shown in Figure 3. Note that $b$ is now a variable of the algorithm, while $g$ is still a parameter. We shall see later how to best choose the value of $g$.

```
$i \leftarrow 0$;
$y \leftarrow x$;
$b \leftarrow 1$;
$m \leftarrow 0$;
repeat
    if $i \equiv 0 \pmod{b}$ and $m = M$ then
      begin
        purge ($b$);
        $b \leftarrow 2b$;
        $m \leftarrow m/2$;
      end;
    if $i \equiv 0 \pmod{b}$ then
      begin
        insert ($y, i$);
        $m \leftarrow m + 1$;
      end;
    $y \leftarrow f(y)$;
    $i \leftarrow i + 1$;
    if $i < b \pmod{gb}$ then lookup ($y, j$);
until found;
```

Figure 3. The cycle detecting algorithm.

Memory management is controlled with a variable $m$, which counts the number of items currently in TABLE, and a procedure *purge* ($b$), which removes all entries $(z, j)$ with $j/b$ odd from TABLE.

This algorithm does not perform exactly as if the simple algorithm of Figure 2 had been run from the beginning with the final value of $b$, because of subtle interactions between the *lookup*, *insert*, and *purge* procedures. To make the algorithm perform properly, we need to introduce suitable restrictions on the choice of $g$, as shown in the following lemmas.

**Lemma 1.** If $M$ is a multiple of $2g$, then the value of $i$ when *purge* is called satisfies $i \equiv 0 \pmod{2gb}$ and $i = 2^k M$ for some integer $k \geq 0$.

*Proof:* By induction on the number of memory purges, it is clear that the $(k+1)$st call to *purge* happens with $i = 2^k M$, and changes $b$ from $2^k$ to $2^{k+1}$. That is, purges occur when $i = bM$. If $M$ is a multiple of $2g$, then $bM$ is certainly a multiple of $2gb$. ▮

**Lemma 2.** The value of $b$ when *lookup* is called is $i/M$ rounded up to the next power of 2.

*Proof:* As in the proof of Lemma 1, we have $b = 2^{k+1}$ for $2^k M < i \leq 2^{k+1} M$. ▮

**Lemma 3.** Whenever *lookup*, *insert*, and *purge* are called during the execution of the cycle detecting algorithm, we have $(f^j(x), j) \in$ TABLE if and only if $0 \leq j < i$ and $j \equiv 0 \pmod{b}$.

*Proof:* Obvious by the definition of *purge*. ▮

**Lemma 4.** If $M$ is a multiple of $2g$, then the cycle detecting algorithm terminates with

$$n \leq i < n + (g+2)b_n,$$

where $b_n$ is $n/M$ rounded up to the next power of 2 (i.e., the value of $b$ when $i$ is $n$).

*Proof:* Since we must have $i > j$ and $f^i(x) = f^j(x)$ at termination, it is clear from the definition of $n$ that $i$ cannot be less than $n$. To complete the proof, we shall show that $i < n+(g+2)b_n$ by contradiction.

Suppose, therefore, that the algorithm is still running when $i$ becomes $n + (g+2)b_n$. Let $i_0$ be the unique integer such that $n \leq i_0 < n + gb_n$ and $i_0 \equiv 0 \pmod{gb_n}$. Then $i_0$ is the first value of $i$ past $i = n$ for which the algorithm might attempt to *lookup* an entire block of $b_n$ consecutive values. By Lemma 1, $i_0$ is also the first value of $i$ past $i = n$ at which a memory purge could take place. Two cases now arise.

If $i_0 \neq 2^k M$ then Lemma 1 guarantees that a *purge* will not occur during the block of $b_n$ *lookup*s which are performed starting at $i_0$. Since $i_0 \geq n$, the values looked up are the same as the $b_n$ values starting at $i_0 - c$ and, by Lemma 3, one of them must be in TABLE, so the algorithm must halt with $i \leq i_0 + b_n < n + (g + 1)b_n$. This contradicts the assumption that the algorithm is still running at $i = n + (g + 2)b_n$.

If $i_0 = 2^k M$ then a memory purge will occur after the *lookup* for $f^{i_0}(x)$. By Lemma 1, we also know that $i_0 \equiv 0 \pmod{2gb_n}$. Thus, immediately after the *purge*, $b = 2b_n$ and $i_0 \equiv 0 \pmod{gb}$. The algorithm will therefore proceed to perform *lookup*s on $2b_n$ values starting at $i_0$. As before, one of these values must be in TABLE, causing the algorithm to halt with $i \leq i_0 + 2b_n < n + (g + 2)b_n$, again in contradiction to our assumption. ∎

The condition that $M$ must be a multiple of $2g$ provides a convenient bound on the overshoot and is essential for the proper performance of the algorithm. If this condition does not hold, it is possible to find $l$ and $c$ for which the algorithm gets caught in a long loop where each block of *lookup*s misses finding the cycle because of the previous *purge*.

These lemmas describe the performance of the algorithm well enough for us to calculate its worst case running time. Intuitively, we expect that the time required should decrease as $g$ and $M$ increase. A small lookup interval implies frequent *lookup*s but a short overshoot past the beginning of the cycle; a large interval limits the number of *lookup*s but at the risk of allowing a long overshoot.

Before proceeding to the detailed analysis we need to finish the solution to the problem: determine $l$ and $c$ once the cycle has been detected.

The cycle detection algorithm halts as soon as it discovers a pair $i > j$ of integers for which $f^i(x) = f^j(x)$. This implies that $j > l$ and $i \equiv j \pmod{c}$, but we need to use our stored TABLE values to find the exact values of $l$ and $c$. The variety of possible situations makes it necessary to design this part of the algorithm carefully. Figure 4 shows a companion algorithm to the algorithm of Figure 3 which recovers the solution $(l, c)$ once the cycle detection algorithm has terminated.

```
$i' \leftarrow gb\lfloor i/gb \rfloor - gb$;
$j' \leftarrow j - (i - i')$;
if $i' > j$
  then $c \leftarrow i - j$;
  else $c \leftarrow$smallest $c$ with $f^j(x) = f^{j+c}(x)$;
$l \leftarrow j' +$smallest $l'$ with $f^{j'+l'}(x) = f^{i'+l'}(x)$;
```

**Figure 4.** The recovery algorithm.

In this program, $i'$ points to the beginning of the first full block of *lookup*s previous to the block containing $i$ and $j'$ is the same distance behind $j$ as $i'$ is behind $i$. Since $i' \equiv 0 \pmod{gb}$, we know from Lemma 3 that $f^{i'}(x)$ is in TABLE. Although $f^{j'}(x)$ is not necessarily stored in TABLE, it can be found by doing a *lookup* for $f^{b\lfloor j'/b \rfloor}(x)$ and then applying $f$ exactly $j' \pmod{b}$ times. This takes at most $t_s + 2b_n t_f$ time because the final value of $b$ is either $b_n$ or $2b_n$.

**Lemma 5.** The recovery algorithm correctly finds $c$.

*Proof:* Since $f^i(x) = f^j(x)$, we have $i \equiv j \pmod{c}$, that is, the cycle size $c$ divides $i - j$. If $i' \leq j$, then $i - j < gb + b \leq (g + 1)2b_n$ and the algorithm proceeds by brute force. If $i' > j$, then an entire block of unsuccessful *lookup*s for $f^{i'}(x), \ldots, f^{i'+b_n-1}(x)$ was performed between $j$ and $i$, and the cycle must have been traversed only once. Thus $c$ is exactly $i - j$. ∎

**Lemma 6.** The recovery algorithm correctly finds $l$.

*Proof:* Clearly $j' < l \leq j$ and $i' < l+c \leq i$ or else the algorithm would have terminated earlier. Since

$i \equiv j \pmod{c}$ and $i - i' = j - j'$, we have $j' \equiv i' \pmod{c}$. Proceeding forward in synchronization from $j'$ and $i'$ will lead to the first duplicate value. ▮

It is possible to design faster recovery procedures for many situations. For example, $c$ could be found by applying "divide and conquer" to the prime decomposition of $i - j$, and $l$ could be found by a binary search procedure. However, the recovery time is heavily dominated by the cycle detection time, so such sophisticated implementations might not be worth the effort.

## 4. WORST CASE ANALYSIS

The algorithms of the previous section can provide an efficient solution to the cycle problem if the parameter $g$ is chosen intelligently. In this section, we shall analyze the running time of the algorithms to find the best choice of $g$. We shall concentrate on choosing a value of $g$ which minimizes the worst case running time.

To begin, we need to add up all the costs involved when the algorithms are run to produce the solution $(l, c)$ to a particular instance $(f, x)$ of the cycle problem.

**Theorem 2.** In the worst case, the running time of the cycle detection algorithm is

$$n\left(1 + \frac{2g+4}{M}\right)\left(t_f + \frac{t_s}{g}\right) + t_u M \log_2 \frac{4\sqrt{2}n}{M}.$$

The additional time required to find the values of $l$ and $c$ is at most

$$n\frac{4(3g+1)}{M}t_f + t_s.$$

*Proof:* For the cycle detection algorithm, the first term follows immediately from Lemma 4, since $b_n$ could be as large as $2n/M$. For the second term, we need to count $t_u$ for each element remaining in TABLE (for its insertion) and $2t_u$ for elements entered in TABLE and subsequently removed in *purge*. The algorithm either performs $\log_2 b_n + 1$ *purges* and finishes with a half full TABLE for a total cost of $t_u(\frac{1}{2}M + M(\log_2 b_n + 1))$, or else it performs $\log_2 b_n$ *purges* and finishes with a TABLE which is more than half full, for a total worst case cost of $t_u(M + M \log_2 b_n)$. The given bound follows from Lemma 4 as above. Note that the worst case is achieved for infinitely many values of $l$ and $c$.

In the recovery algorithm, $gb$ function evaluations could be required to find $c$, and $2gb$ function evaluations may be needed to find $l$, with an additional $b$ function evaluations and 1 table search needed to find $f^{j'}(x)$. The final value of $b$ is either $b_n$ or $2b_n$, and is therefore bounded by $4n/M$. ▮

To find the value of $g$ which minimizes the total running time, we would like to set the derivative with respect to $g$ of the expression for the total from Theorem 2 to zero and solve for $g$. However, there is a subtle difficulty involved because we also have the constraint that $g$ must divide $M/2$. (As an extreme example, suppose that $M/2$ is prime.) To make it possible to find a reasonable value of $g$, we shall introduce the further restriction that $M$ should be a power of two.

**Theorem 3.** If $M$ is a power of two, then the total running time of the algorithms to solve the cycle problem will be less than

$$nt_f\left(1 + O\left(\sqrt{\frac{t_s}{M}}\right)\right),$$

if $g$ is chosen to be the closest power of two to $\sqrt{(M+4)t_s/14t_f}$.

*Proof:* It follows immediately from minimizing the total of the running times given in Theorem 2 that the given choice of g minimizes the total running time to

$$n\left(t_f + 2\sqrt{\frac{14t_st_f}{M}} + O\left(\frac{t_s}{M}\right)\right),$$

which implies the stated result, since $t_f$ is assumed constant. The asymptotic result is not affected by restricting $g$ to be a power of two, which insures that it divides $M/2$. ▮

Note, in particular, that a balanced tree implementation will have $t_s = O(\log M)$, and an address calculation (hashing) method could have $t_s = O(1)$. In both cases, the worst case running time will approach $nt_f$ as $M$ gets large. For the typical case where $l + c >> M$, the algorithms can be made to run in very nearly optimal time by increasing $M$.

## 5. Concluding remarks

We have dealt exclusively with algorithms with good worst case performance for solving a particular instance of the cycle problem on an unknown function. The problem is also interesting under other variations of the model.

If we take the function to be *random* in some sense, then we can talk about an average case measure of complexity. R. W. Floyd has pointed out that studying the probabalistic structure of random functions over $D$ could lead to savings on the average. For example, $l$ is relatively large in this case, so it may not be worthwhile to save or lookup values at the beginning.

Another variation is to consider the case where the cost of computing $f^j(x)$ is $t_f$, independent of $j$ and $x$. The algorithms of this paper will work in this case, with a worst case running time (corresponding to Theorem 3) of $O(nt_f/M)$. There may exist algorithms customized for this variant which could achieve better performance.

In general, the domain $D$ can be partitioned by $f$ into disjoint sets with the property that all points in each set lead to the same cycle. Properties of the *cycle structure* (i.e. the number of sets, their sizes, or the sizes of the cycles) can be found by solving the cycle problem on all points of $D$. The algorithms of this paper can be adapted to avoid retraversing long cycles by maintaining versions of TABLE for each cycle.

It is possible to generalize the problem of finding cycles in the following way. As before, we are given a unary function $f$, only now we allow the domain of $f$ to be infinite. We suppose that we are also given a binary predicate $P$ on $D \times D$. The problem is to find the smallest $n$ for which there exists an $l < n$ such that $P(f^n(x), f^l(x))$. In the absence of any further information, it is easy to show that this problem requires $\binom{n}{2}$ evaluations of $P$. However, if $P$ is *preserved* by $f$, that is, $P(a, b)$ implies $P(f(a), f(b))$, then the algorithms of this paper can be made to run in time $n(t_f + O(t_p))$, where $t_p$ is the time needed to evaluate $P$. If we extend the domain D to be infinite, then the cycle will still be detected if one exists. It is interesting to note that the algorithms of [1] and [2] simply do not work for this problem.

It remains an open question whether an algorithm exists for the cycle problem which uses a finite amount of memory and an optimal number of function evaluations.

## 6. References

[1] Knuth, D. E. *Seminumerical Algorithms*, (Vol. 2 of *The Art of Computer Programming*), Addison-Wesley, 1969.

[2] Gosper, R., *et.al.*,"HAKMEM," M.I.T. Artificial Intelligence Lab Report No. 239, Feb. 1972, item 132.