

# Cycle detection

From Wikipedia, the free encyclopedia

In computer science, **cycle detection** or **cycle finding** is the algorithmic problem of finding a cycle in a sequence of iterated function values.

For any function  $f$  that maps a finite set  $S$  to itself, and any initial value  $x_0$  in  $S$ , the sequence of iterated function values

$$x_0, x_1 = f(x_0), x_2 = f(x_1), \dots, x_i = f(x_{i-1}), \dots$$

must eventually use the same value twice: there must be some pair of distinct indices  $i$  and  $j$  such that  $x_i = x_j$ . Once this happens, the sequence must continue periodically, by repeating the same sequence of values from  $x_i$  to  $x_j - 1$ . Cycle detection is the problem of finding  $i$  and  $j$ , given  $f$  and  $x_0$ .

Several algorithms for finding cycles quickly and with little memory are known. Floyd's tortoise and the hare algorithm moves two pointers at different speeds through the sequence of values until they both point to equal values. Alternatively, Brent's algorithm is based on the idea of exponential search. Both Floyd's and Brent's algorithms use only a constant number of memory cells, and take a number of function evaluations that is proportional to the distance from the start of the sequence to the first repetition. Several other algorithms trade off larger amounts of memory for fewer function evaluations.

The applications of cycle detection include testing the quality of pseudorandom number generators and cryptographic hash functions, computational number theory algorithms, detection of infinite loops in computer programs and periodic configurations in cellular automata, and the automated shape analysis of linked list data structures.

## Contents

- 1 Example
- 2 Definitions
- 3 Computer representation
- 4 Algorithms
  - 4.1 Floyd's Tortoise and Hare
  - 4.2 Brent's algorithm
  - 4.3 Gosper's algorithm
  - 4.4 Time – space tradeoffs
- 5 Applications
- 6 References
- 7 External links

## Example

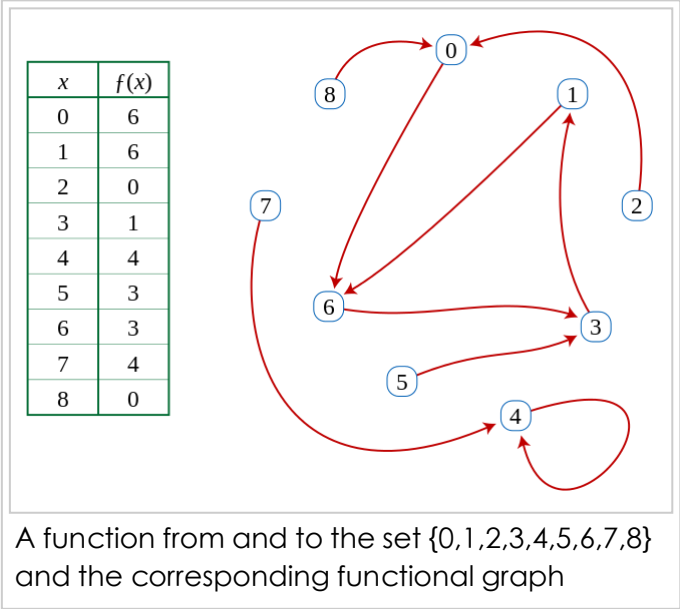
The figure shows a function  $f$  that maps the set  $S = \{0,1,2,3,4,5,6,7,8\}$  to itself. If one starts from  $x_0 = 2$  and repeatedly applies  $f$ , one sees the sequence of values

2, 0, 6, 3, 1, 6, 3, 1, 6, 3, 1, ....

The cycle in this value sequence is 6, 3, 1.

Definitions

Let  $S$  be any finite set,  $f$  be any function from  $S$  to itself, and  $x_0$  be any element of  $S$ . For any  $i > 0$ , let  $x_i = f(x_{i-1})$ . Let  $\mu$  be the smallest index such that the value  $x_\mu$  reappears infinitely often within the sequence of values  $x_i$ , and let  $\lambda$  (the loop length) be the smallest positive integer such that  $x_\mu = x_{\lambda + \mu}$ . The cycle detection problem is the task of finding  $\lambda$  and  $\mu$ .<sup>[1]</sup>



One can view the same problem graph-theoretically, by constructing a functional graph (that is, a directed graph in which each vertex has a single outgoing edge) the vertices of which are the elements of  $S$  and the edges of which map an element to the corresponding function value, as shown in the figure. The set of vertices reachable from starting vertex  $x_0$  form a subgraph with a shape resembling the Greek letter rho ( $\rho$ ): a path of length  $\mu$  from  $x_0$  to a cycle of  $\lambda$  vertices.<sup>[2]</sup>

Computer representation

Generally,  $f$  will not be specified as a table of values, the way it is shown in the figure above. Rather, a cycle detection algorithm may be given access either to the sequence of values  $x_i$ , or to a subroutine for calculating  $f$ . The task is to find  $\lambda$  and  $\mu$  while examining as few values from the sequence or performing as few subroutine calls as possible. Typically, also, the space complexity of an algorithm for the cycle detection problem is of importance: we wish to solve the problem while using an amount of memory significantly smaller than it would take to store the entire sequence.

In some applications, and in particular in Pollard's rho algorithm for integer factorization, the algorithm has much more limited access to  $S$  and to  $f$ . In Pollard's rho algorithm, for instance,  $S$  is the set of integers modulo an unknown prime factor of the number to be factorized, so even the size of  $S$  is unknown to the algorithm. To allow cycle detection algorithms to be used with such limited knowledge, they may be designed based on the following capabilities. Initially, the algorithm is assumed to have in its memory an object representing a pointer to the starting value  $x_0$ . At any step, it may perform one of three actions: it may copy any pointer it has to another object in memory, it may apply  $f$  and replace any of its pointers by a pointer to the next object in the sequence, or it may apply a subroutine for determining whether two of its pointers represent equal values in the sequence. The equality test action may involve some nontrivial computation: for instance, in Pollard's rho algorithm, it is implemented by testing whether the difference between two stored values has a nontrivial greatest common divisor with the number to be factored.<sup>[2]</sup> In this context, by analogy to the pointer machine model of computation, an algorithm that only uses pointer copying, advancement within the sequence, and equality tests may be called a pointer algorithm.

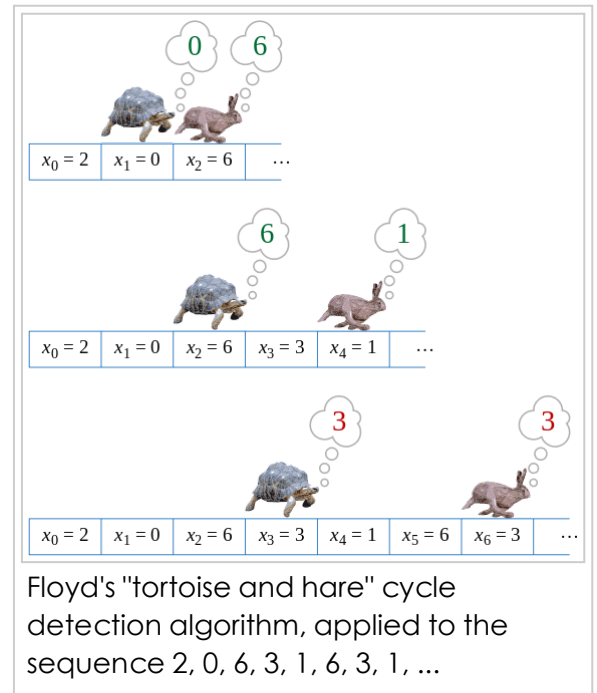
Algorithms

If the input is given as a subroutine for calculating  $f$ , the cycle detection problem may be trivially solved using only  $\lambda + \mu$  function applications, simply by computing the sequence of values  $x_i$  and using a data structure such as a hash table to store these values and test whether each subsequent value has already been stored. However, the space complexity of this algorithm is proportional to  $\lambda + \mu$ , unnecessarily large. Additionally, to implement this method as a pointer algorithm would require applying the equality test to each pair of values, resulting in quadratic time overall. Thus, research in this area has concentrated on two goals: using less space than this naive algorithm, and finding pointer algorithms that use fewer equality tests.

## Floyd's Tortoise and Hare

Floyd's cycle-finding algorithm is a pointer algorithm that uses only two pointers, which move through the sequence at different speeds. It is also called the "tortoise and the hare algorithm", alluding to Aesop's fable of The Tortoise and the Hare.

The algorithm is named after Robert W. Floyd, who was credited with its invention by Donald Knuth.<sup>[3][4]</sup> However, the algorithm does not appear in Floyd's published work, and this may be a misattribution: Floyd describes algorithms for listing all simple cycles in a directed graph in a 1967 paper,<sup>[5]</sup> but this paper does not describe the cycle-finding problem in functional graphs that is the subject of this article. In fact, Knuth's statement (in 1969), attributing it to Floyd, without citation, is the first known appearance in print, and it thus may be a folk theorem, not attributable to a single individual.<sup>[6]</sup>



The key insight in the algorithm is that, for any integers  $i \geq \mu$  and  $k \geq 0$ ,  $x_i = x_{i+k\lambda}$ , where  $\lambda$  is the length of the loop to be found and  $\mu$  is the index of the first element of the cycle. In particular,  $i = k\lambda \geq \mu$ , if and only if  $x_i = x_{2i}$ . Thus, the algorithm only needs to check for repeated values of this special form, one twice as far from the start of the sequence as the other, to find a period  $\nu$  of a repetition that is a multiple of  $\lambda$ . Once  $\nu$  is found, the algorithm retraces the sequence from its start to find the first repeated value  $x_\mu$  in the sequence, using the fact that  $\lambda$  divides  $\nu$  and therefore that  $x_\mu = x_{\mu+\nu}$ . Finally, once the value of  $\mu$  is known it is trivial to find the length  $\lambda$  of the shortest repeating cycle, by searching for the first position  $\mu + \lambda$  for which  $x_{\mu+\lambda} = x_\mu$ .

The algorithm thus maintains two pointers into the given sequence, one (the tortoise) at  $x_i$ , and the other (the hare) at  $x_{2i}$ . At each step of the algorithm, it increases  $i$  by one, moving the tortoise one step forward and the hare two steps forward in the sequence, and then compares the sequence values at these two pointers. The smallest value of  $i > 0$  for which the tortoise and hare point to equal values is the desired value  $\nu$ .

The following Python code shows how this idea may be implemented as an algorithm.

```
def floyd(f, x0):
    # Main phase of algorithm: finding a repetition  $x_i = x_{2i}$ .
    # The hare moves twice as quickly as the tortoise and
    # the distance between them increases by 1 at each step.
    # Eventually they will both be inside the cycle and then,
```

```

# at some point, the distance between them will be
# divisible by the period  $\lambda$ .
tortoise = f(x0) # f(x0) is the element/node next to x0.
hare = f(f(x0))
while tortoise != hare:
    tortoise = f(tortoise)
    hare = f(f(hare))

# At this point the tortoise position,  $v$ , which is also equal
# to the distance between hare and tortoise, is divisible by
# the period  $\lambda$ . So hare moving in circle one step at a time,
# and tortoise (reset to x0) moving towards the circle, will
# intersect at the beginning of the circle. Because the
# distance between them is constant at  $2v$ , a multiple of  $\lambda$ ,
# they will agree as soon as the tortoise reaches index  $\mu$ .

# Find the position  $\mu$  of first repetition.
mu = 0
tortoise = x0
while tortoise != hare:
    tortoise = f(tortoise)
    hare = f(hare) # Hare and tortoise move at same speed
    mu += 1

# Find the length of the shortest cycle starting from  $x_\mu$ 
# The hare moves one step at a time while tortoise is still.
# lam is incremented until  $\lambda$  is found.
lam = 1
hare = f(tortoise)
while tortoise != hare:
    hare = f(hare)
    lam += 1

return lam, mu

```

This code only accesses the sequence by storing and copying pointers, function evaluations, and equality tests; therefore, it qualifies as a pointer algorithm. The algorithm uses  $O(\lambda + \mu)$  operations of these types, and  $O(1)$  storage space.<sup>[7]</sup>

## Brent's algorithm

Richard P. Brent described an alternative cycle detection algorithm that, like the tortoise and hare algorithm, requires only two pointers into the sequence.<sup>[8]</sup> However, it is based on a different principle: searching for the smallest power of two  $2^i$  that is larger than both  $\lambda$  and  $\mu$ . For  $i = 0, 1, 2, \dots$ , the algorithm compares  $x_{2^i-1}$  with each subsequent sequence value up to the next power of two, stopping when it finds a match. It has two advantages compared to the tortoise and hare algorithm: it finds the correct length  $\lambda$  of the cycle directly, rather than needing to search for it in a subsequent stage, and its steps involve only one evaluation of  $f$  rather than three.<sup>[9]</sup>

The following Python code shows how this technique works in more detail.

```

def brent(f, x0):
    # main phase: search successive powers of two
    power = lam = 1
    tortoise = x0
    hare = f(x0) # f(x0) is the element/node next to x0.
    while tortoise != hare:
        if power == lam: # time to start a new power of two?
            tortoise = hare
            power *= 2
            lam = 0
        hare = f(hare)
        lam += 1

```

```

# Find the position of the first repetition of length  $\lambda$ 
mu = 0
tortoise = hare = x0
for i in range(lam):
    # range(lam) produces a list with the values 0, 1, ..., lam-1
    hare = f(hare)
    # The distance between the hare and tortoise is now  $\lambda$ .

# Next, the hare and tortoise move at same speed until they agree
while tortoise != hare:
    tortoise = f(tortoise)
    hare = f(hare)
    mu += 1

return lam, mu

```

Like the tortoise and hare algorithm, this is a pointer algorithm that uses  $O(\lambda + \mu)$  tests and function evaluations and  $O(1)$  storage space. It is not difficult to show that the number of function evaluations can never be higher than for Floyd's algorithm. Brent claims that, on average, his cycle finding algorithm runs around 36% more quickly than Floyd's and that it speeds up the Pollard rho algorithm by around 24%. He also performs an average case analysis for a randomized version of the algorithm in which the sequence of indices traced by the slower of the two pointers is not the powers of two themselves, but rather a randomized multiple of the powers of two. Although his main intended application was in integer factorization algorithms, Brent also discusses applications in testing pseudorandom number generators.<sup>[8]</sup>

## Gosper's algorithm

R. W. Gosper's algorithm<sup>[10][11]</sup> finds the period but not the starting point of the first cycle. Its main feature is that it never backs up to reevaluate the generator function, and is economical in both space and time. For example, if it is known a priori that the generator function has a period of  $2^{32}$  then 33 words is sufficient.

## Time – space tradeoffs

A number of authors have studied techniques for cycle detection that use more memory than Floyd's and Brent's methods, but detect cycles more quickly. In general these methods store several previously-computed sequence values, and test whether each new value equals one of the previously-computed values. In order to do so quickly, they typically use a hash table or similar data structure for storing the previously-computed values, and therefore are not pointer algorithms: in particular, they usually cannot be applied to Pollard's rho algorithm. Where these methods differ is in how they determine which values to store. Following Nivasch,<sup>[12]</sup> we survey these techniques briefly.

- Brent<sup>[8]</sup> already describes variations of his technique in which the indices of saved sequence values are powers of a number  $R$  other than two. By choosing  $R$  to be a number close to one, and storing the sequence values at indices that are near a sequence of consecutive powers of  $R$ , a cycle detection algorithm can use a number of function evaluations that is within an arbitrarily small factor of the optimum  $\lambda + \mu$ .<sup>[13][14]</sup>
- Sedgewick, Szymanski, and Yao<sup>[15]</sup> provide a method that uses  $M$  memory cells and requires in the worst case only  $(\lambda + \mu)(1 + cM^{-1/2})$  function evaluations, for some constant  $c$ , which they show to be optimal. The technique involves maintaining a

numerical parameter  $d$ , storing in a table only those positions in the sequence that are multiples of  $d$ , and clearing the table and doubling  $d$  whenever too many values have been stored.

- Several authors have described *distinguished point* methods that store function values in a table based on a criterion involving the values, rather than (as in the method of Sedgewick et al.) based on their positions. For instance, values equal to zero modulo some value  $d$  might be stored.<sup>[16][17]</sup> More simply, Nivasch<sup>[12]</sup> credits D. P. Woodruff with the suggestion of storing a random sample of previously seen values, making an appropriate random choice at each step so that the sample remains random.
- Nivasch<sup>[12]</sup> describes an algorithm that does not use a fixed amount of memory, but for which the expected amount of memory used (under the assumption that the input function is random) is logarithmic in the sequence length. An item is stored in the memory table, with this technique, when no later item has a smaller value. As Nivasch shows, the items with this technique can be maintained using a stack data structure, and each successive sequence value need be compared only to the top of the stack. The algorithm terminates when the repeated sequence element with smallest value is found. Running the same algorithm with multiple stacks, using random permutations of the values to reorder the values within each stack, allows a time – space tradeoff similar to the previous algorithms. However, even the version of this algorithm with a single stack is not a pointer algorithm, due to the comparisons needed to determine which of two values is smaller.

Any cycle detection algorithm that stores at most  $M$  values from the input sequence must perform at least  $(\lambda + \mu)(1 + \frac{1}{M-1})$  function evaluations.<sup>[18][19]</sup>

## Applications

Cycle detection has been used in many applications.

- Determining the cycle length of a pseudorandom number generator is one measure of its strength. This is the application cited by Knuth in describing Floyd's method.<sup>[3]</sup> Brent<sup>[8]</sup> describes the results of testing a linear congruential generator in this fashion; its period turned out to be significantly smaller than advertised. For more complex generators, the sequence of values in which the cycle is to be found may not represent the output of the generator, but rather its internal state.
- Several number-theoretic algorithms are based on cycle detection, including Pollard's rho algorithm for integer factorization<sup>[20]</sup> and his related kangaroo algorithm for the discrete logarithm problem.<sup>[21]</sup>
- In cryptographic applications, the ability to find two distinct values  $x_{\mu-1}$  and  $x_{\lambda+\mu-1}$  mapped by some cryptographic function  $f$  to the same value  $x_\mu$  may indicate a weakness in  $f$ . For instance, Quisquater and Delescaille<sup>[17]</sup> apply cycle detection algorithms in the search for a message and a pair of Data Encryption Standard keys that map that message to the same encrypted value; Kaliski, Rivest, and Sherman<sup>[22]</sup> also use cycle detection algorithms to attack DES. The technique may also be used to find a collision in a cryptographic hash function.<sup>[23]</sup>

- Cycle detection may be helpful as a way of discovering infinite loops in certain types of computer programs.<sup>[24]</sup>
- Periodic configurations in cellular automaton simulations may be found by applying cycle detection algorithms to the sequence of automaton states.<sup>[12]</sup>
- Shape analysis of linked list data structures is a technique for verifying the correctness of an algorithm using those structures. If a node in the list incorrectly points to an earlier node in the same list, the structure will form a cycle that can be detected by these algorithms.<sup>[25]</sup> In Common Lisp, the S-expression printer, under control of the `*print-circle*` variable, detects circular list structure and prints it compactly.
- Teske<sup>[14]</sup> describes applications in computational group theory: determining the structure of an Abelian group from a set of its generators. The cryptographic algorithms of Kaliski et al.<sup>[22]</sup> may also be viewed as attempting to infer the structure of an unknown group.
- Fich (1981) briefly mentions an application to computer simulation of celestial mechanics, which she attributes to William Kahan. In this application, cycle detection in the phase space of an orbital system may be used to determine whether the system is periodic to within the accuracy of the simulation.<sup>[18]</sup>

## References

1. Joux, Antoine (2009), *Algorithmic Cryptanalysis* ([https://books.google.com/books?id=buQajqt-\\_iUC&pg=PA223](https://books.google.com/books?id=buQajqt-_iUC&pg=PA223)), CRC Press, p. 223, ISBN 9781420070033.
2. Joux (2009), p. 224.
3. Knuth, Donald E. (1969), *The Art of Computer Programming, vol. II: Seminumerical Algorithms*, Addison-Wesley, p. 7, exercises 6 and 7
4. *Handbook of Applied Cryptography*, by Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, p. 125 (<https://books.google.com/books?id=nSzoG72E93MC&pg=PA125>), describes this algorithm and others
5. Floyd, R.W. (1967), "Non-deterministic Algorithms" (<http://doi.acm.org/10.1145/321420.321422>), *J. ACM*, 14 (4): 636 – 644, doi:10.1145/321420.321422 (<https://doi.org/10.1145%2F321420.321422>)
6. *The Hash Function BLAKE*, by Jean-Philippe Aumasson, Willi Meier, Raphael C.-W. Phan, Luca Henzen (2015), p. 21 (<https://books.google.com/books?id=nhPmBQAAQBAJ&pg=PA21>), footnote 8
7. Joux (2009), Section 7.1.1, Floyd's cycle-finding algorithm, pp. 225 – 226.
8. Brent, R. P. (1980), "An improved Monte Carlo factorization algorithm" (<http://www.maths.anu.edu.au/~brent/pd/rpb051i.pdf>) (PDF), *BIT Numerical Mathematics*, 20 (2): 176 – 184, doi:10.1007/BF01933190 (<https://doi.org/10.1007%2FBF01933190>).
9. Joux (2009), Section 7.1.2, Brent's cycle-finding algorithm, pp. 226 – 227.
10. <http://www.hackersdelight.org/hdcodetxt/loopdet.c.txt>
11. <http://www.inwap.com/pdp10/hbaker/hakmem/flows.html>
12. Nivasch, Gabriel (2004), "Cycle detection using a stack", *Information Processing Letters*, 90 (3): 135 – 140, doi:10.1016/j.ipl.2004.01.016 (<https://doi.org/10.1016%2Fj.ipl.2004.01.016>).
13. Schnorr, Claus P.; Lenstra, Hendrik W. (1984), "A Monte Carlo factoring algorithm with linear storage", *Mathematics of Computation*, 43 (167): 289 – 311, JSTOR 2007414 (<https://www.jstor.org/stable/2007414>), doi:10.2307/2007414 (<https://doi.org/10.2307%2F2007414>).
14. Teske, Edlyn (1998), "A space-efficient algorithm for group structure computation", *Mathematics of Computation*, 67 (224): 1637 – 1663, doi:10.1090/S0025-5718-98-00968-5 (<https://doi.org/10.1090%2FS0025-5718-98-00968-5>).

15. Sedgewick, Robert; Szymanski, Thomas G.; Yao, Andrew C.-C. (1982), "The complexity of finding cycles in periodic functions", *SIAM Journal on Computing*, 11 (2): 376 – 390, doi:10.1137/0211030 (<https://doi.org/10.1137%2F0211030>).
16. van Oorschot, Paul C.; Wiener, Michael J. (1999), "Parallel collision search with cryptanalytic applications", *Journal of Cryptology*, 12 (1): 1 – 28, doi:10.1007/PL00003816 (<https://doi.org/10.1007%2FPL00003816>).
17. Quisquater, J.-J.; Delescaille, J.-P., "How easy is collision search? Application to DES", *Advances in Cryptology – EUROCRYPT '89, Workshop on the Theory and Application of Cryptographic Techniques* (<http://www.springerlink.com/content/0wnq56dava5nnm17/>), Lecture Notes in Computer Science, 434, Springer-Verlag, pp. 429 – 434.
18. Fich, Faith Ellen (1981), "Lower bounds for the cycle detection problem", *Proc. 13th ACM Symposium on Theory of Computing*, pp. 96 – 105, doi:10.1145/800076.802462 (<https://doi.org/10.1145%2F800076.802462>).
19. Allender, Eric W.; Klawe, Maria M. (1985), "Improved lower bounds for the cycle detection problem", *Theoretical Computer Science*, 36 (2 – 3): 231 – 237, doi:10.1016/0304-3975(85)90044-1 (<https://doi.org/10.1016%2F0304-3975%2885%2990044-1>).
20. Pollard, J. M. (1975), "A Monte Carlo method for factorization", *BIT*, 15 (3): 331 – 334, doi:10.1007/BF01933667 (<https://doi.org/10.1007%2FBF01933667>).
21. Pollard, J. M. (1978), "Monte Carlo methods for index computation (mod  $p$ )", *Mathematics of Computation*, American Mathematical Society, 32 (143): 918 – 924, JSTOR 2006496 (<https://www.jstor.org/stable/2006496>), doi:10.2307/2006496 (<https://doi.org/10.2307%2F2006496>).
22. Kaliski, Burton S., Jr.; Rivest, Ronald L.; Sherman, Alan T. (1988), "Is the Data Encryption Standard a group? (Results of cycling experiments on DES)", *Journal of Cryptology*, 1 (1): 3 – 36, doi:10.1007/BF00206323 (<https://doi.org/10.1007%2FBF00206323>).
23. Joux (2009), Section 7.5, Collisions in hash functions, pp. 242 – 245.
24. Van Gelder, Allen (1987), "Efficient loop detection in Prolog using the tortoise-and-hare technique", *Journal of Logic Programming*, 4 (1): 23 – 31, doi:10.1016/0743-1066(87)90020-3 (<http://doi.org/10.1016%2F0743-1066%2887%2990020-3>).
25. Auguston, Mikhail; Hon, Miu Har (1997), "Assertions for Dynamic Shape Analysis of List Data Structures", *AADEBUG '97, Proceedings of the Third International Workshop on Automatic Debugging* (<http://www.ep.liu.se/ea/cis/1997/009/04/>), Linköping Electronic Articles in Computer and Information Science, Linköping University, pp. 37 – 42.

## External links

- Gabriel Nivasch, The Cycle Detection Problem and the Stack Algorithm (<http://www.gabrielnivasch.org/fun/cycle-detection>)
- Tortoise and Hare (<http://c2.com/cgi/wiki/Curry?TortoiseAndHare>), Portland Pattern Repository
- Floyd's Cycle Detection Algorithm (The Tortoise and the Hare) (<http://www.siafoo.net/algorithm/10>)
- Brent's Cycle Detection Algorithm (The Teleporting Turtle) (<http://www.siafoo.net/algorithm/11>)

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Cycle\\_detection&oldid=783561725](https://en.wikipedia.org/w/index.php?title=Cycle_detection&oldid=783561725)"

Categories: Fixed points (mathematics) | Combinatorial algorithms



- This page was last edited on 2017-06-03, at 11:42:03.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.