THE TORTOISE AND THE HARE

A Cycle Detection Algorithm

Link to repo: https://github.com/manaar95/cycle-detection

Manaar Hyder
1323089
Comp Sci 3EA3

Table of Contents

Abstract

One of Aesop's infamous fables is the one about the tortoise and the hare. In this age-old tale, a tortoise and a hare set out to compete against each other in a race. Although undoubtedly faster than the tortoise, the hare loses the race. The tortoise is predictably slower than the hare, thus giving the hare enough confidence to stop for a break. The tortoise, meanwhile, carries on, and soon passes the hare, and the finish line.

Unlike the story, the coding version of the tortoise and the hare algorithm doesn't involve the hare taking a break – at least not in most instances. The hare and the tortoise serve as two iterators over any given list, whether standard or linked. The tortoise traditionally moves 1 element forward at each iteration, while the hare moves 2. The program ends when the hare either reaches the end of the list or crosses the tortoise, depending on the specific algorithm.

This report will examine the traditional hare and tortoise algorithm, as well as similar algorithms that also relate to cycle detection. The applications covered in this report will include one instance of the selection algorithm and 3 instances of the cycle detection algorithm; one basic detector, the Floyd algorithm and the Brent algorithm.

Introduction

Algorithms Chosen

My interest in the hare and the tortoise algorithm first took shape when I began studying for job interviews. I found that many practice interview recourses referred to this algorithm and asked questions about it. When I sat down to research the concept, I realized that it wasn't just one algorithm, there were a couple of algorithms that served the same purpose with varying levels of efficiency.

This assignment served as a good reason delve back into this concept. I chose to do the selection algorithm because it's a very simple way to use two iterators at different speeds. It will help me understand how the hare and tortoise work together to determine anything in a list. From there, I plan to do a basic cycle detection algorithm. This will get me started on the traditional way the tortoise and hare algorithm is used. The base version of cycle detection simply detects whether a cycle exists. It returns a Boolean value.

Furthermore, I plan to explore two other algorithms using the hare and tortoise concept. Floyd's cycle detection algorithm and Brent's cycle detection algorithm both provide more functionality than a basic cycle detection algorithm. These two algorithms are able to determine the starting point and the length of the loop. The difference between the two is the way the tortoise and the hare behave in each iteration. In Floyd's algorithm, the hare moves two steps for every one step the tortoise takes while in Brent's algorithm, the hare consistently takes one step forward while the tortoise stays still except the occasional teleportation to the hare's location. This will be explained in the Algorithm exploration section of this report.

Contributors to the Algorithms

Robert W. Floyd invented his namesake algorithm back in 1960. It works in linear time to detect a cycle in a singly linked list. (Isaacson, 2010). The algorithm is also known as "The Tortoise and the Hare" algorithm. Twenty years later, Richard P. Brent invented his own version of a cycle detection algorithm. While it also works in linear time, it is more efficient than Floyd's algorithm since it takes fewer steps to find the cycle. Brent's algorithm is aptly named "The Teleporting Turtle", and the reasons for this will be explained in the code walkthrough for the algorithm (Brent's Cycle Detection Algorithm (The Teleporting Turtle), 2010).

Applications of The Algorithms

This algorithm is useful for - but not limited to - testing the quality of pseudorandom number generators and cryptographic hash functions, detection of infinite loops in computer programs and periodic configurations in cellular automata, computational number theory algorithms, and the automated shape analysis of linked list data structures (Cycle detection, 2018).

Tools Chosen

The tools chosen for this project are Dafny and Alloy, both languages were new to me. I am now looking forward to adding them both to my resume.

I chose Dafny because the language already supported program verification and would not require any extensions to be downloaded. Another huge selling point for me was that the Dafny site had online tutorials, an online verifier, and lots of sample code with heavy explanations to help new users understand and recreate similar code from scratch. Despite all of this, however, I had troubles checking for null elements in a list and thought I might have better luck in a different language or with a different compiler.

Since Dafny was a new language for me, I thought I might have better luck if I worked in a language I'm already familiar with and used some sort of extension. I looked into Java verifiers and settled on an extension called Kopitiam. Once it was downloaded from the eclipse marketplace, I couldn't figure out how to have Eclipse use it.

As a third attempt, I began looking for languages similar to Java that would have built in verification properties. After one of my classmates did a presentation on Alloy, I was sold. I saw the similarities between Alloy and Java, although there were also quite a few distinctions. I found that Alloy worked well with predicates. Interesting fact: Alloy does not support Booleans, I had to define how true and false worked on my own.

My tool selection came full circle however, I ended up working with Dafny again by the end of the project. Now, 3 of my algorithms are in Dafny and one is in Alloy.

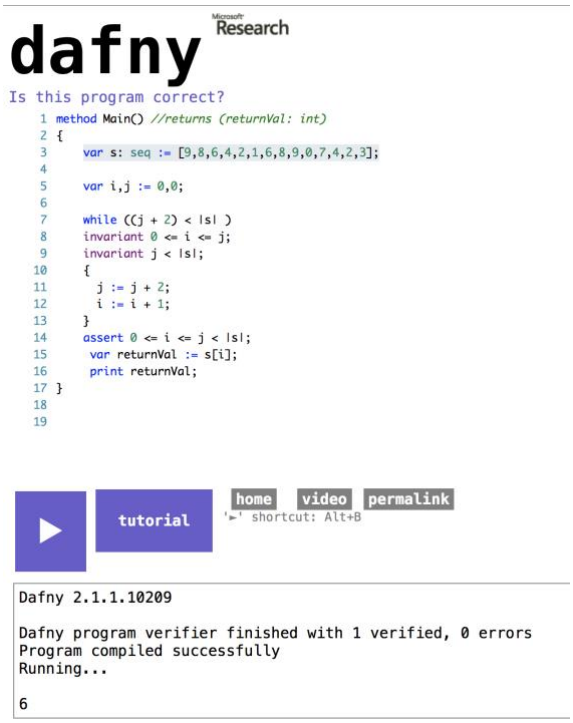
Below are more details about these two languages and how to work with them for these algorithms.

Dafny

Dafny is an object-oriented programming language with a program verifier. As the program is typed, the verifier is constantly checking and flagging any errors (Leino, 2018).

Running the code online

To run the code online, simply go to: https://rise4fun.com/dafny and replace the existing code with the code you would like to run. Click the play button towards the bottom when ready to execute the code. The output will display directly under that.



Running the code online is probably the most efficient way to check the code, as it does not require any installations or any sort of set up. The way to utilize the online IDE is uniform across all platforms

Running the Code Locally (Windows)

Download Dafny.
Follow the instructions at: https://github.com/Microsoft/dafny/wiki/INSTALL

Running the Code Locally (Mac/Linux)

Start off by downloading mono.
Follow the instruction at: http://www.mono-project.com/docs/getting-started/install/mac/
Set up mono. This will allow the code to compile, but won't execute it.
Follow the instructions at: https://www.cs.cmu.edu/~mfredrik/15414/hw/guide.pdf.

To be able to execute code, use wine:
Follow the instructions at: https://www.davidbaumgold.com/tutorials/wine-mac/#part-3:-install-wine-using-homebrew.

Alloy

Alloy is a language for describing structures and a tool for exploring them. An Alloy model is a collection of constraints that describe a set of structures. The Alloy Analyzer takes the constraints of a model and finds structures that satisfy them. It can be used to check properties of the model (Jackson, 2012).

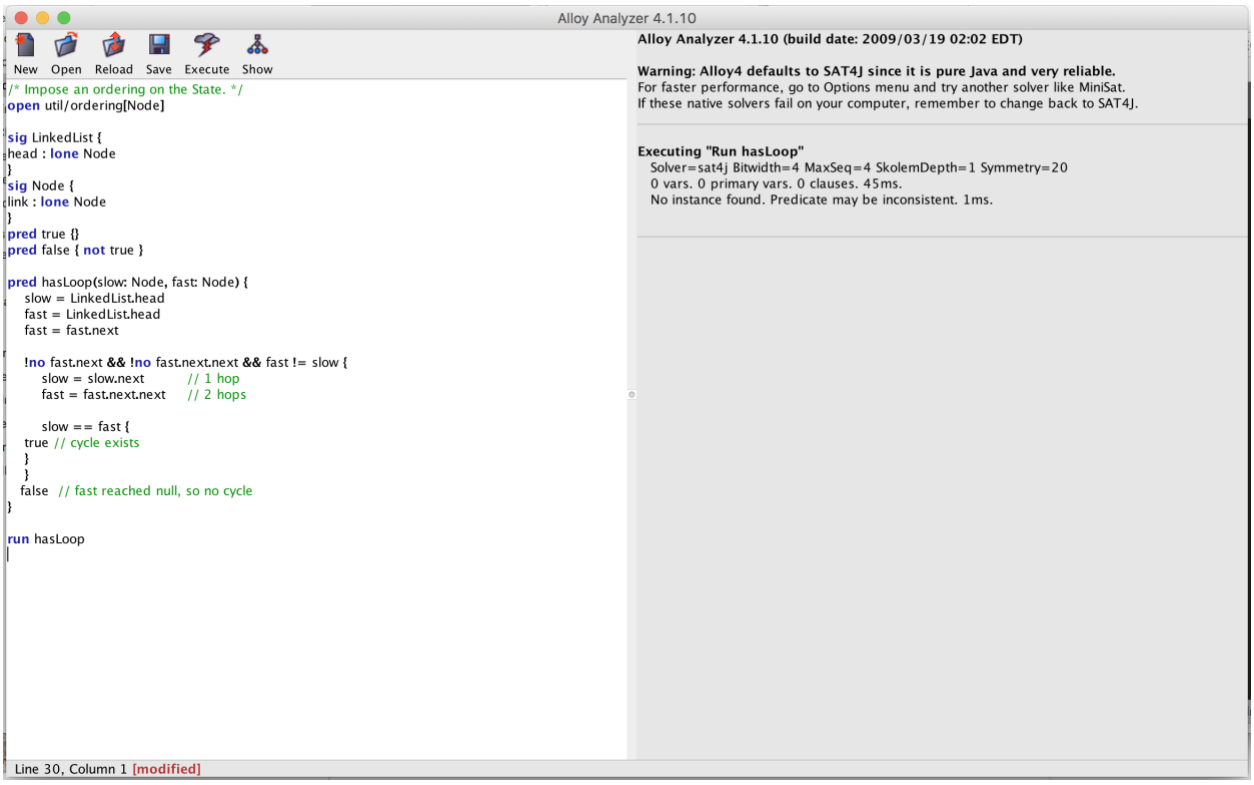
Downloading Alloy

Download alloy from their website
Choose the latest stable release: http://alloytools.org/download.html

***NOTE** Java 9 causes issues with Alloy. I had to revert to Java 8 on my machine to be able to use the compiler.

Running the Code

Double click on the .jar file (mine was called alloy4.jar) to open the compiler. Paste the code into the text area and hit the execute button in the top left:



The Algorithms Explored

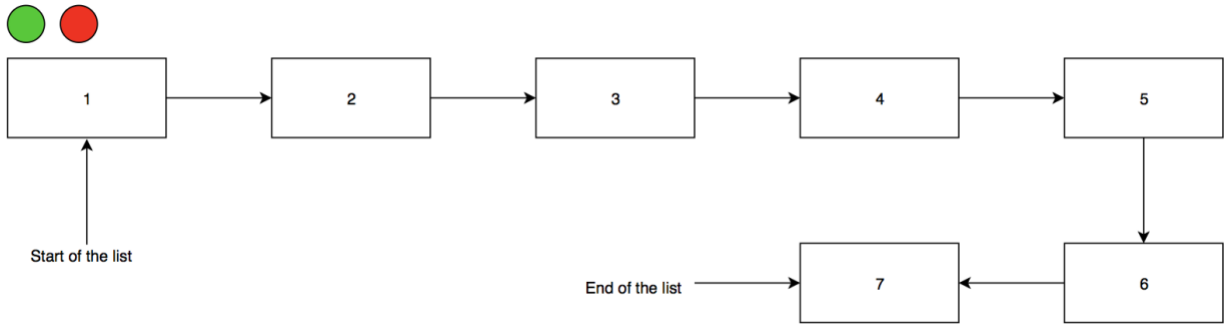
Each walk through will contain a diagram to help illustrate how the algorithm works. The red circle will represent the hare, and the green will represent the tortoise. The code, and its explanations will follow. Since not all of the code was formally verified, there will be some statements about assertions in the comments, which will not be reflected in the existing code.

Selection Algorithm

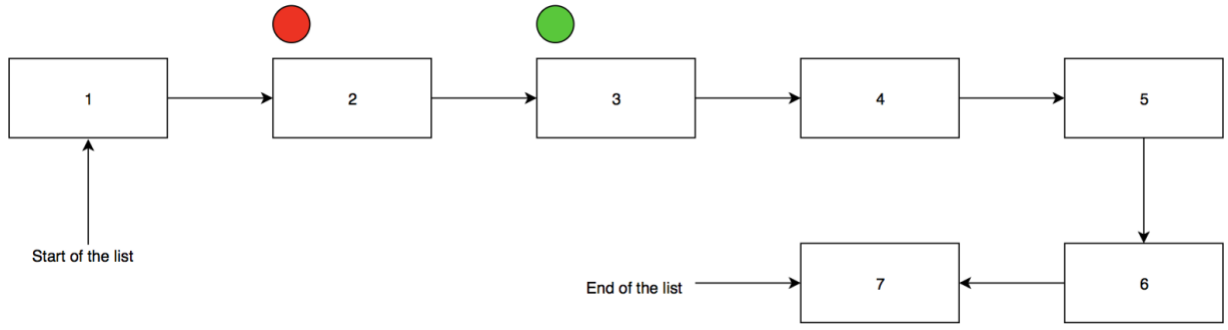
Illustrated Overview

The selection algorithm works by giving the hare a head start in the race. The amount of the head start depends on each case. If the goal is to find the x-last element of the list, then the hare will jump forward x steps before the tortoise and hare both begin moving in sync. When the hare reaches the end of the list, the tortoise will be on the x-last element, thus making it easily retrievable in linear time.

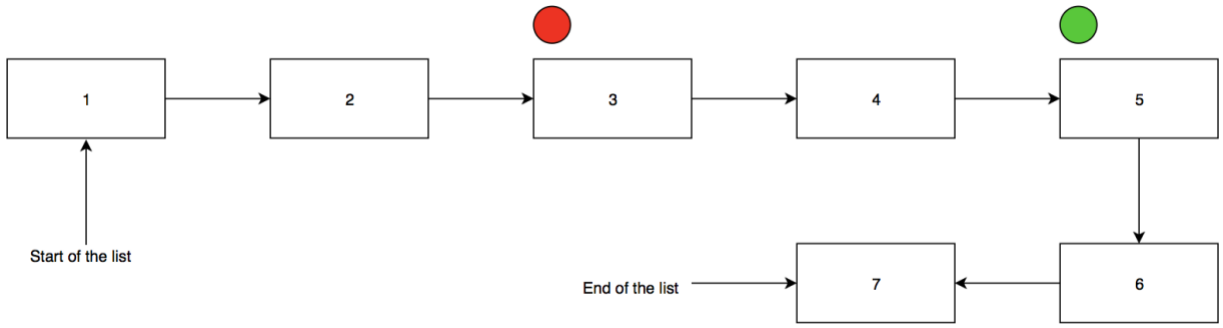
In this case, however, the algorithm will demonstrate how to find the middle of the list using the two iterators. The hare and the tortoise will both start off at the beginning of the list:



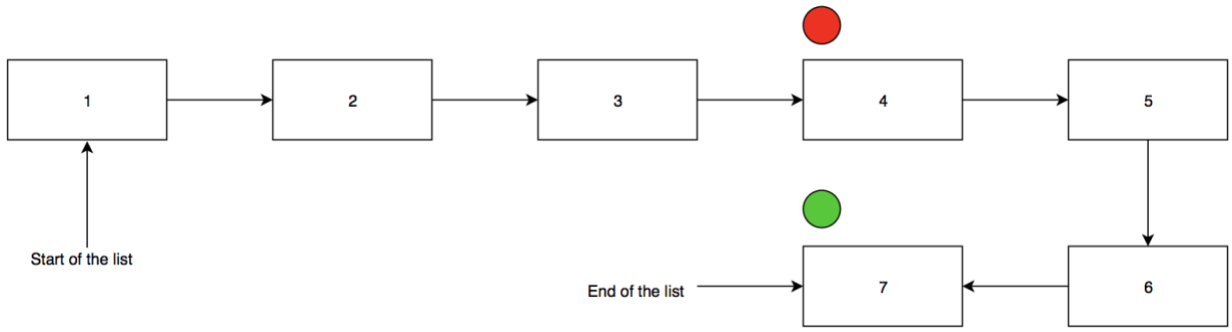
For each step the tortoise takes, the hare will take 2:



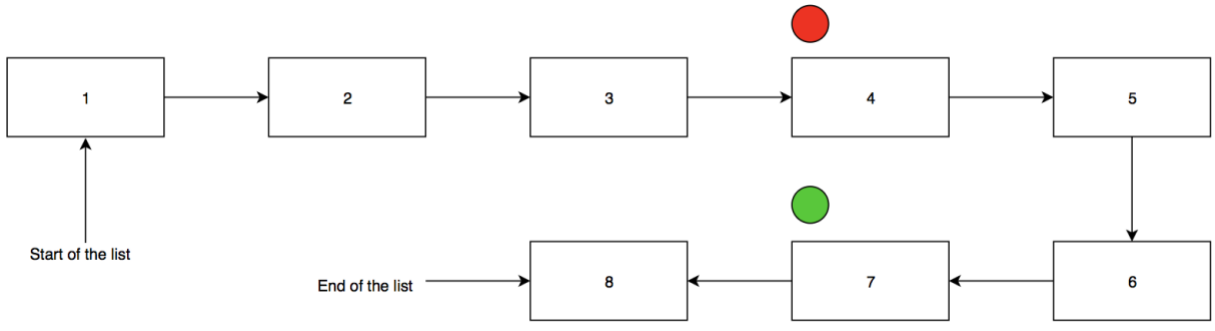
Thus, the tortoise will always be halfway as far as the hare:



When the hare reaches the end of the list, the tortoise will be right at the middle:



There are 3 list elements before the tortoise, and 3 after, thus the tortoise is in the middle. This is the case for a list of odd length, however, suppose the list actually has eight elements instead of 7:



The hare will not move forward only one step, since he can only hop two steps at a time, as explained in the code walkthrough. Since he does not have two more steps to hop, this is where the program will end, returning the value at the tortoise's current location, which is 4.

Code Walkthrough (Dafny)

```
method Main() //returns (returnVal: int)
{
   var s: seq := [1,2,3,4,5];

   var i,j := 0,0;

   while ((j + 2) < |s| )
   invariant 0 <= i <= j;
   invariant j < |s|;
   {
    j := j + 2;
    i := i + 1;
   }
   assert 0 <= i <= j < |s|;
    var returnVal := s[i];
    print returnVal;
}
```

The code for this program is all contained inside of a main method. Initialize a sequence (Dafny's version of a list), and give it a name, s. This is the list that will be traversed during the program. Initialize two variables, i and j, for the tortoise and the hare respectively. Since the hare will be jumping two steps forward at each iteration, it is important to check that his next jump will not go out of bounds before each iteration. Therefore, check whether the hare's current index plus 2 would be within the list. As long as it is, have the hare take two steps, and the tortoise take one. The program will end once the hare can no longer move two steps forward. At that point, the value at the tortoise's location will be returned.

There are two invariants and an assertion in this code. An invariant is a fact that must be maintained for the duration of the loops. An assertion serves as a check to ensure that certain facts are still true.

The first invariant says that neither the tortoise nor the hare can be at a negative index, and that the tortoise cannot be ahead of the hare. Both of these points must always remain true. Neither iterator should ever drop below 0, as it would then be out of bounds. The tortoise should also never surpass the hare, given that it only takes one step forward for every two the hare takes.

The second invariant ensures that the hare iterator is always within the bounds of the list. Admittedly, this is not entirely necessary, due to the loop guard making sure that the hare Is always at least two jumps away from the end of the list. The final assertion at the end.

The assertion at the end ensures that the facts from the invariant still hold after the loop terminates. Both iterators and the list length should still be positive. The tortoise still cannot be ahead of the hare, and the hare must still be within the bounds of the list.

Obstacles

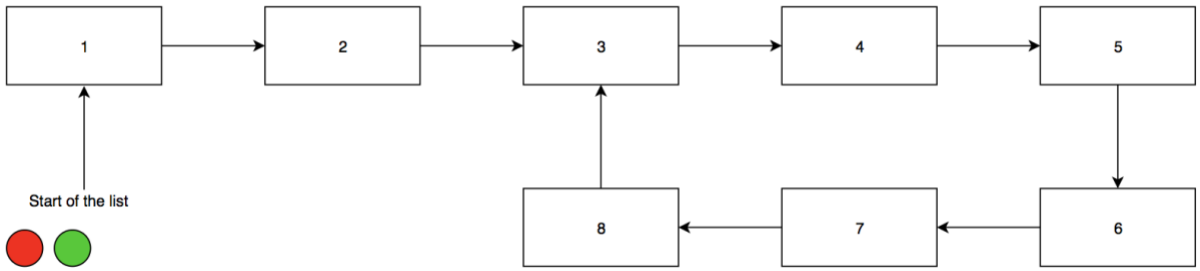
Ideally, this program, would run without a pre-defined list length. Currently, it doesn't quite make sense to use this algorithm as it would be more efficient to just divide the length of a list in half to ascertain the middle of the list. The value at that element can then be returned. This algorithm would work best on a list of unknown length, however, the way to make that work would be to check that the hare would not be encountering a null value if he hopped two steps forward. As far as my research proved, there is no way to check whether an element in a list is null. Dr. Sekerinski therefore advised to work with lists, assuming the length is known.

Cycle Detection

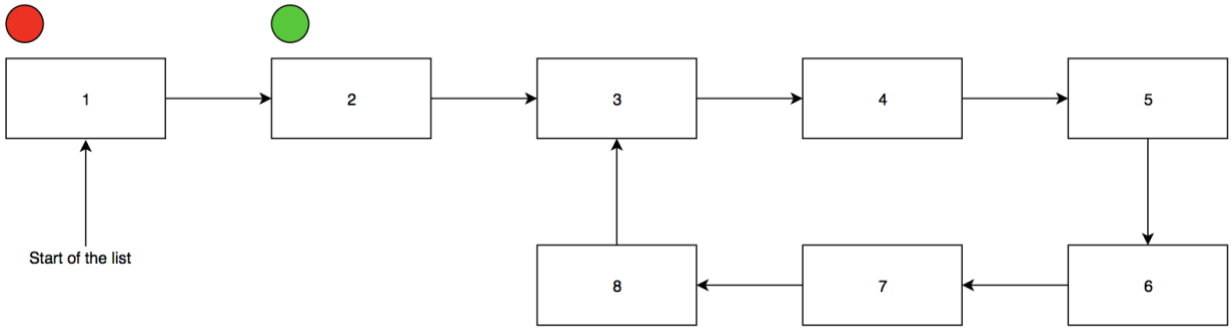
Illustrated Overview

The cycle detection algorithm works by having the hare move twice as fast as the tortoise. If the hare reaches the end of the list, there is no loop. However, if the hare and tortoise ever end up on the same list element, there is a loop. The logic behind this is that, the hare is moving faster than the tortoise. So in a linear list, he would just reach the end. But the fact that he overlapped with the tortoise means that he somehow ended up behind the tortoise at some point, thus implying the existence of a loop.

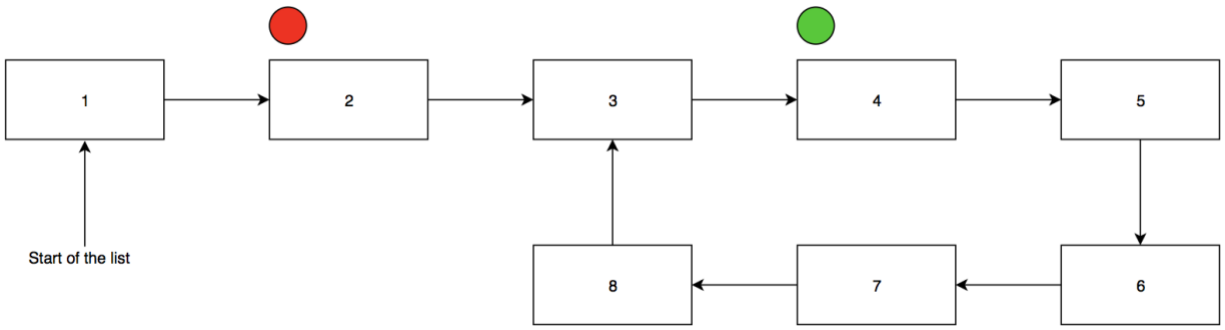
The hare and the tortoise will both start at the same spot:



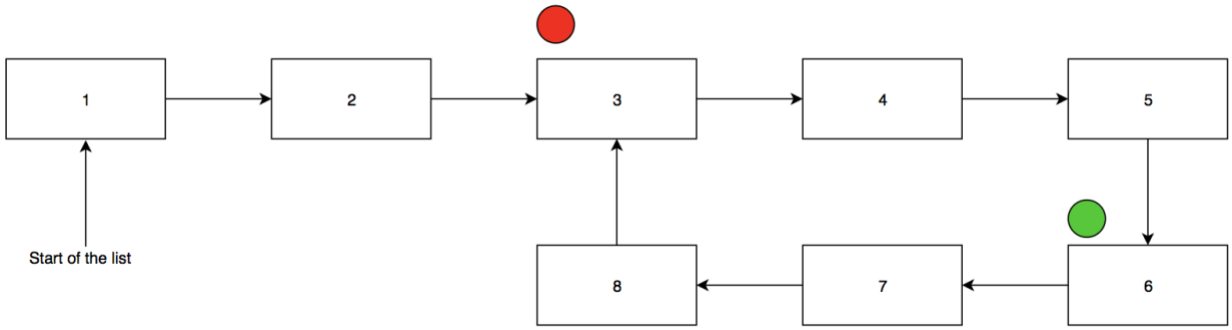
The tortoise will take one step forward, and the hare will take two:



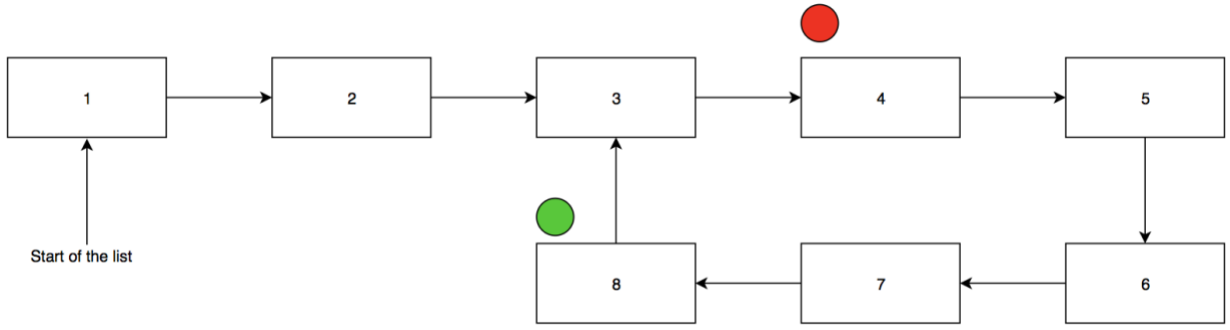
This will happen for many iterations:



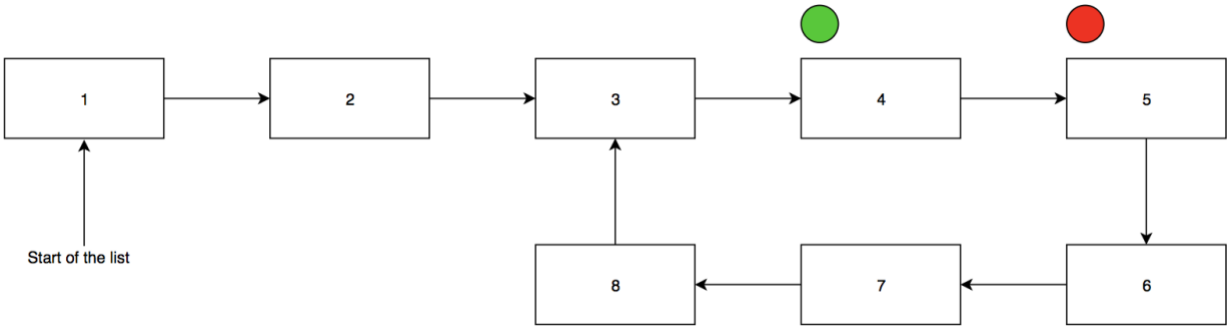
This will happen for many iterations:



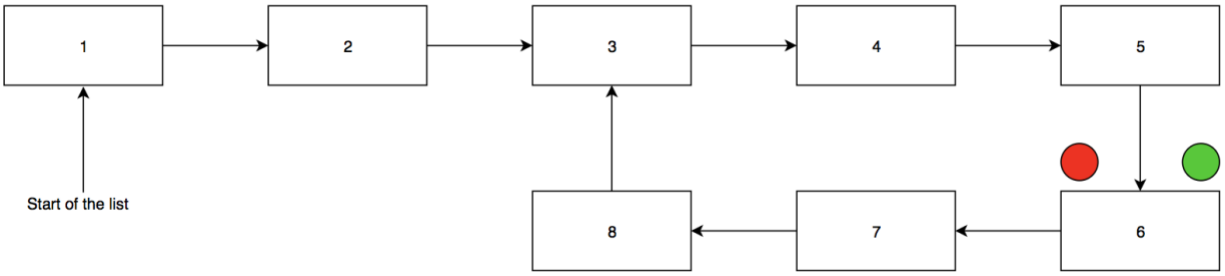
(continue hare iterating two, tortoise iterating one)



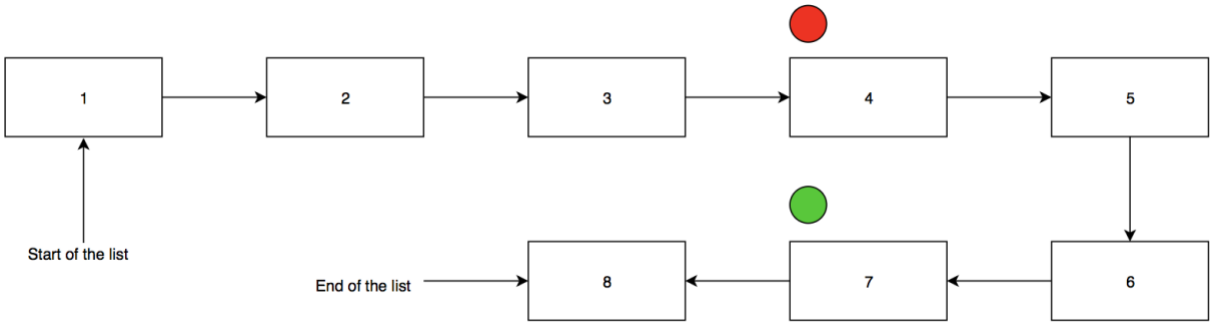
(continue hare iterating two, tortoise iterating one)



Until the tortoise and the hare end up on the same element. Once they do, the program will return that a cycle has been proven to exist:



If the list ends and the hare cannot move forward, there was no cycle:



The program will return that there is no cycle to detect and then terminate.

Code Walkthrough (Alloy)

```
/* Impose an ordering on the State. */
open util/ordering[Node]

sig LinkedList {
head : lone Node
}
sig Node {
link : lone Node
}
pred true {}
pred false { not true }

pred hasLoop(slow: Node, fast: Node) {
   slow = LinkedList.head
   fast = LinkedList.head
   fast = fast.next

   !no fast.next && !no fast.next.next && fast != slow {
      slow = slow.next         // 1 hop
      fast = fast.next.next     // 2 hops

      slow == fast {
          true // cycle exists
          }
   }
  false  // fast reached null, so no cycle
}

run hasLoop
```

This code begins by stating that the nodes are ordered, so that the language knows that they are connected and can automatically traverse the elements using the built in next function. The linked list is initialized, with its head as a lone node. The rest of the nodes in the link have been initialized as links.

Alloy does not currently support Boolean values in the way that Java does. Since this code is implemented in a way that requires true and false values, the true and false predicates must be defined, as empty, which defaults to true for true and not true for false.

The work for this program is done in a predicate function called hasLoop. It works with two node pointers – slow to represent the tortoise, and fast to represent the hare. Both are initialized to point at the head of the list at first, although the fast iterator is bumped over to the next element of the list before entering the loop. This is because of the way the loop is set up to return true if the fast and slow are at the same point. It is important that the two don't start off at the same spot, to avoid returning a false positive for the cycle detection. Therefore, the fast iterator takes one step forward before the cycle begins

Then, as long as the two elements after the fast pointer are not null, fast will move forward two elements (using .next.next) and slow will move forward one element (using .next). If fast and slow ever point to the same element, then it is true that a cycle exists. However, if the two don't overlap and the hare cannot go any further (i.e., if the next two elements are not not-null) then it is false that there is a cycle in the list.

There are no invariants or assertions in the current code. However, in terms of an informal formalisation, there should be at least one invariant and two assertions. The invariant would be at the beginning of the while loop body, right before fast and slow do their hops. It would simply state that fast does not equal slow. Another invariant could state that slow.next is not null. It is important to note though, that while the fast iterator will have already passed that

spot, and fast only goes where there are no nulls, there still needs to be an explicit instruction for slow as well.

Of the two assertions, the first one would be where true is being called. Right before calling true, there should be a check that neither fast nor slow are null values. The second assertion would go right before false is called. There should be a check that fast does not equal slow. Unfortunately, I couldn't figure out how to implement any of these.

Obstacles

The code needs to be able to stop running as soon as it hits either true or false, and return whichever of the two it hit. However, I was unable to figure out how to do this, so there is currently no result being returned from this function. The output is as follows:

```
Executing "Run hasLoop"
  Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
  0 vars. 0 primary vars. 0 clauses. 51ms.
  No instance found. Predicate may be inconsistent. 0ms.
```

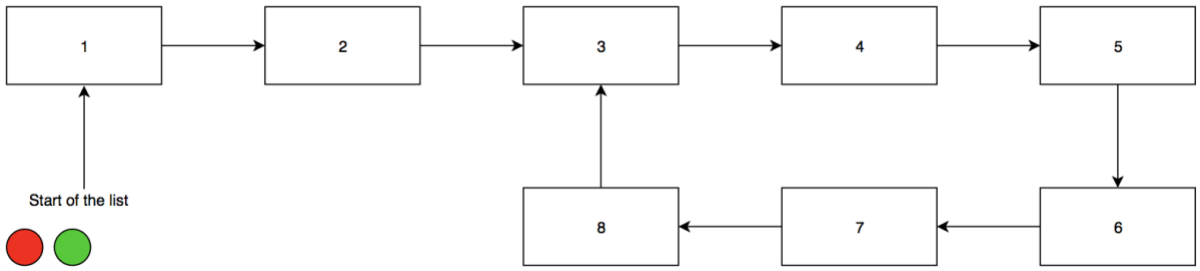
The predicate is being classified as inconsistent due to the fact that is missing assertions. The assertions I would have liked to include in this program are listed in the section above.

Floyd's cycle detection algorithm

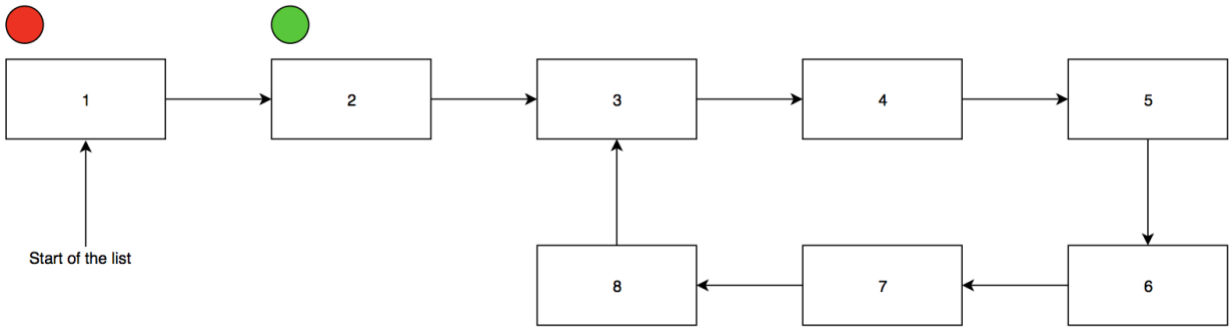
Illustrated Overview

Floyd's cycle detection algorithm works in two main sections. The first is to detect whether a cycle exists. This part uses the cycle detection algorithm from above. If no cycle exists, the program is already over. However, if a cycle does exist, the next step is to find the beginning of the loop. This is done by moving the tortoise back to the beginning of the list, and then incrementing both iterators by one until the tortoise and hare meet. The length of the loop can then be calculated by keeping the tortoise in place and incrementing the hare until he meets the tortoise again (More, 2015).

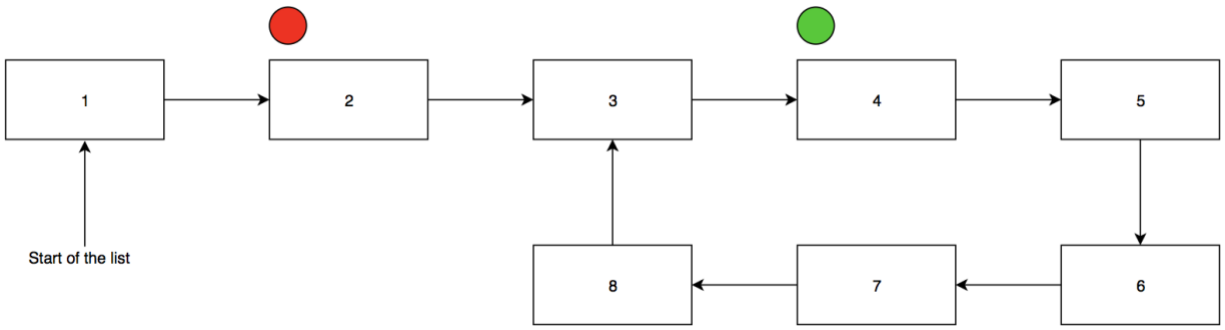
The hare and the tortoise will both start at the same spot:



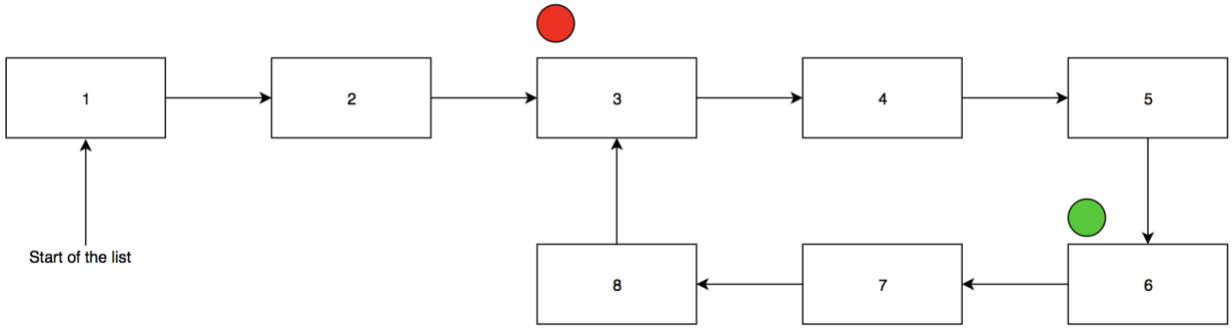
The tortoise will take one step forward, and the hare will take two:



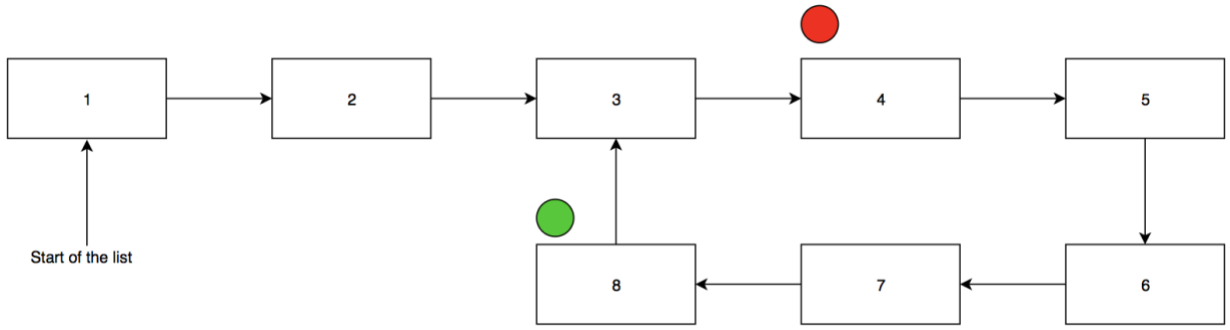
This will happen for many iterations:



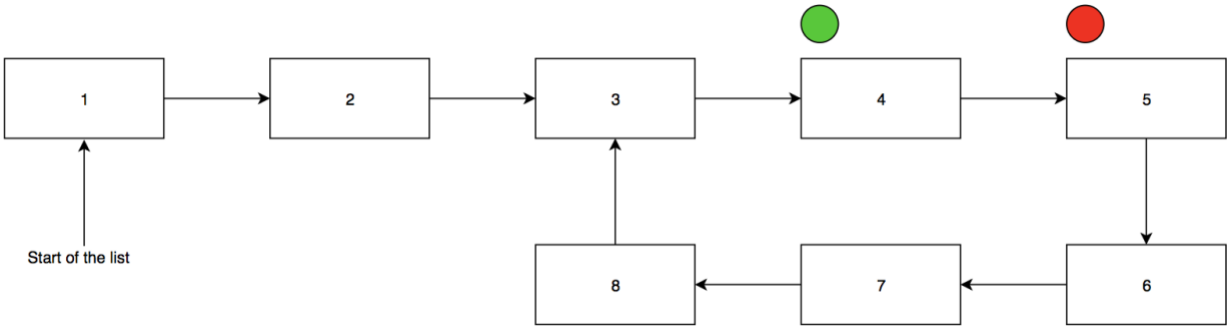
(continue hare iterating two, tortoise iterating one)



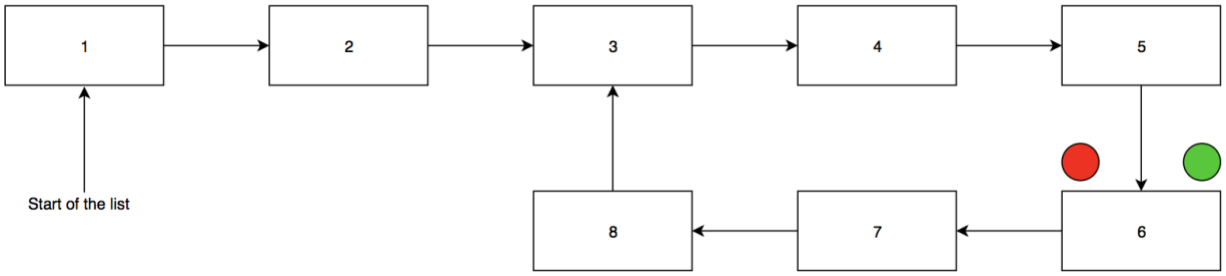
(continue hare iterating two, tortoise iterating one)



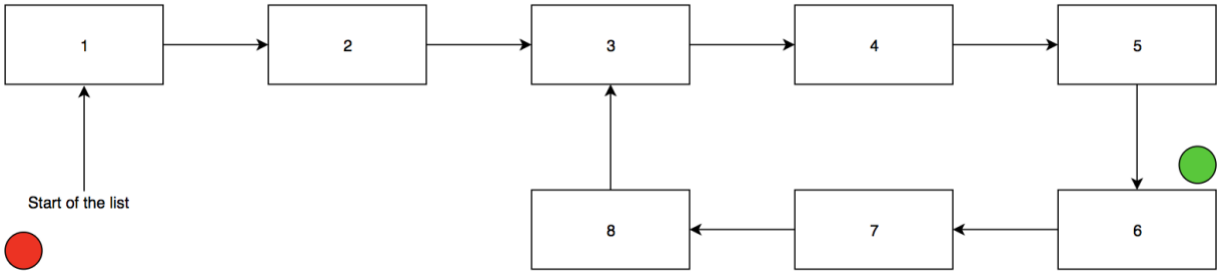
(continue hare iterating two, tortoise iterating one)



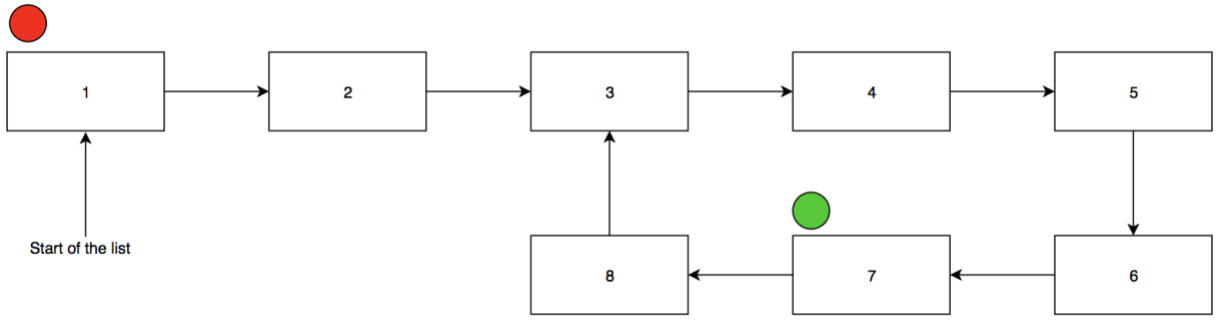
Until the tortoise and the hare end up on the same element. Once they do, a cycle has been proven to exist:



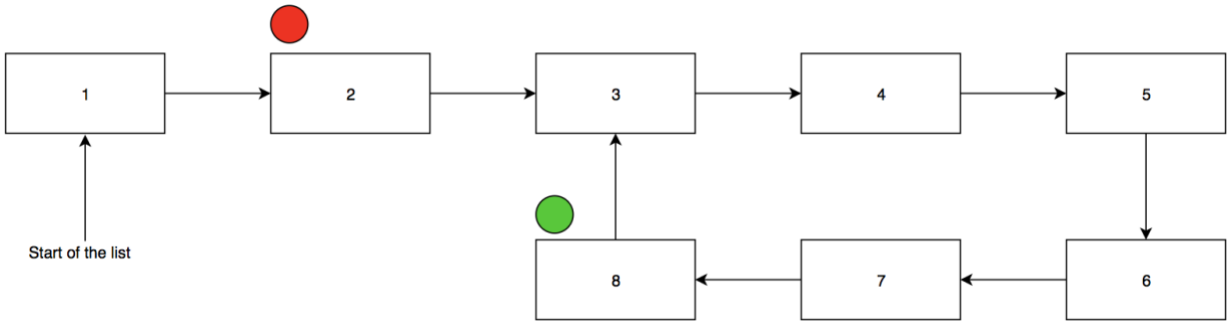
To find the first node in the cycle, move the tortoise to the start of the list:



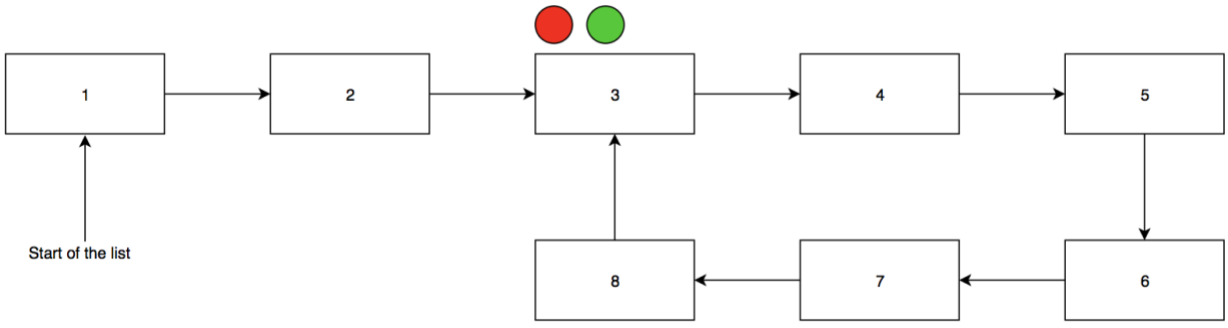
Increment each by 1 jump:



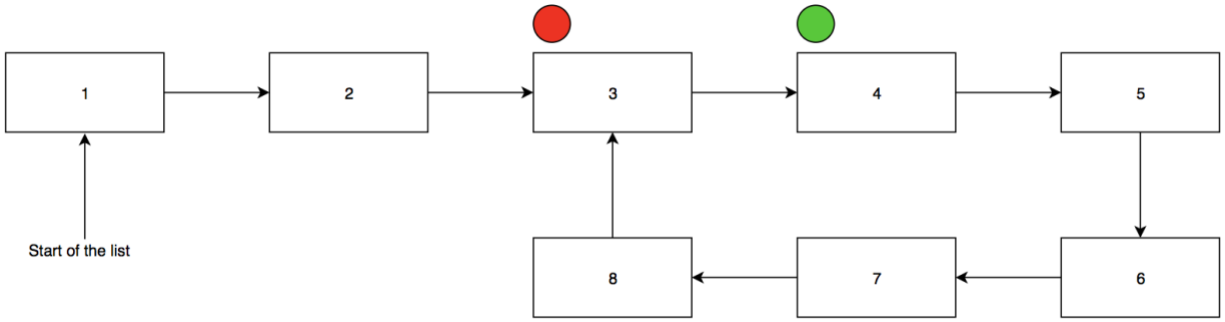
Keep incrementing:



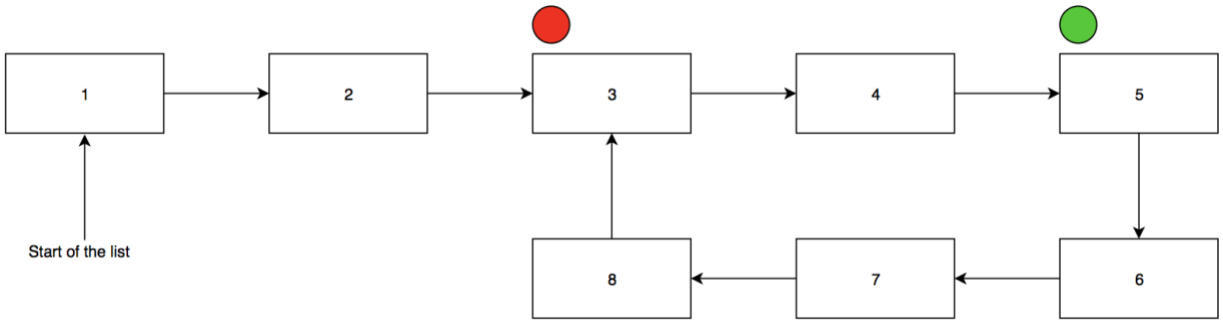
When the hare and the tortoise meet, the start of the list has been found:



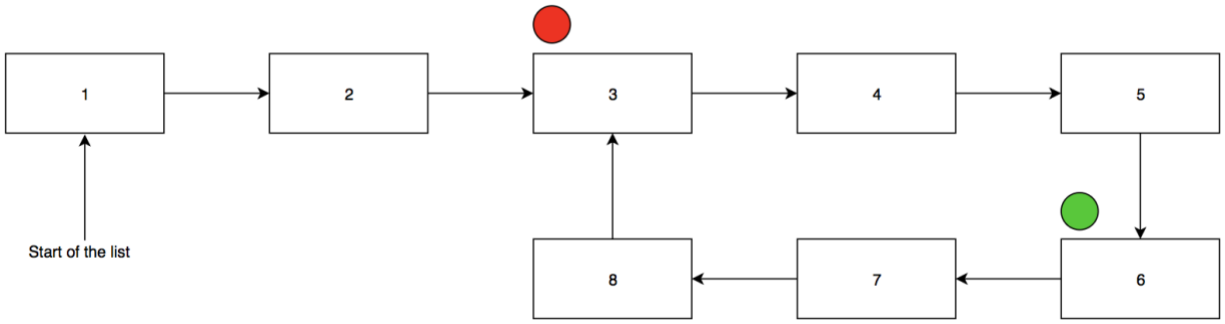
Increment the hare to find the length of the loop, initialize a counter to 1:



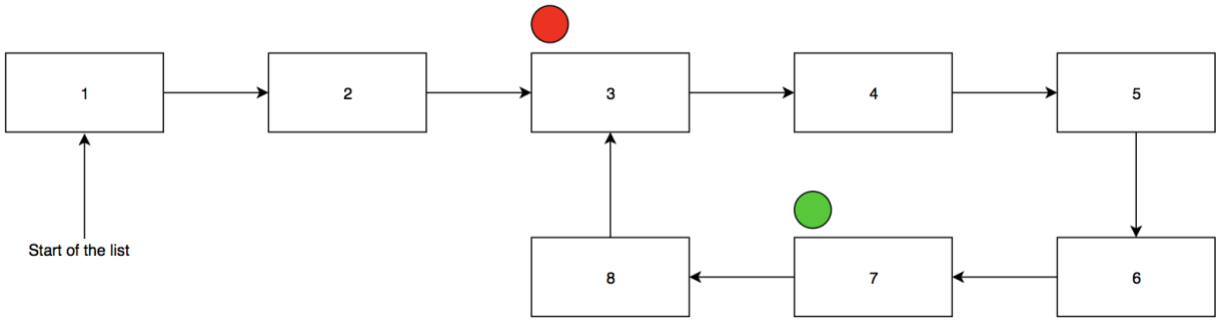
Keep incrementing the hare and the counter by 1:



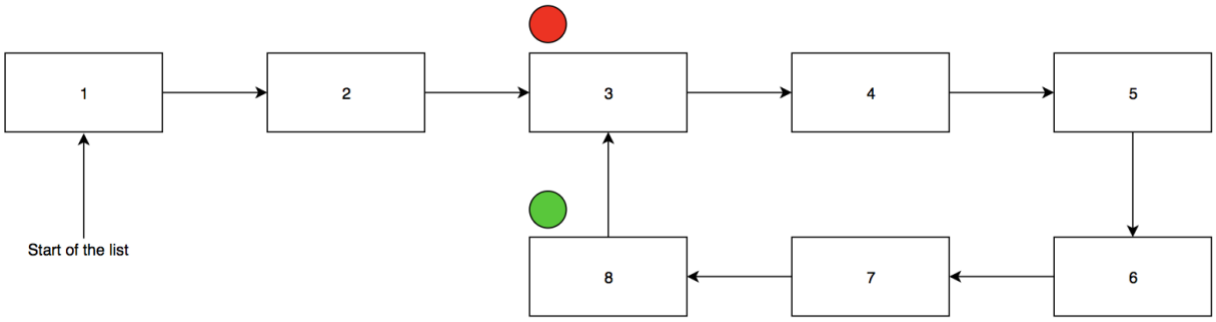
Keep incrementing the hare and the counter by 1:



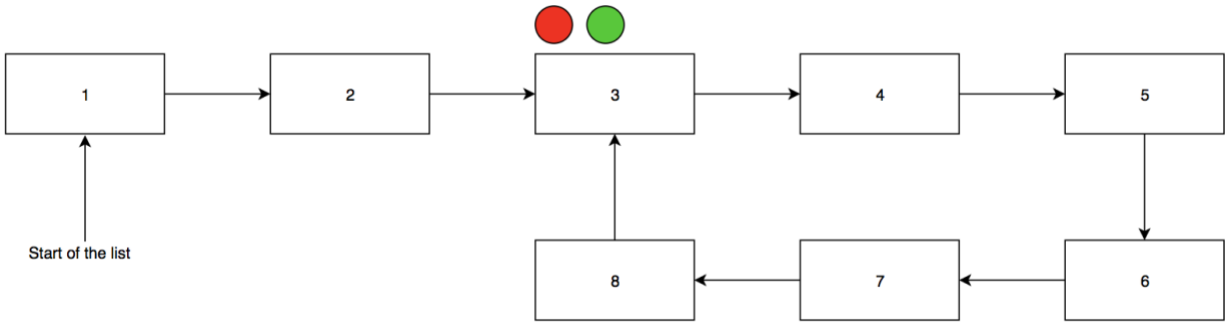
Keep incrementing the hare and the counter by 1:



Keep incrementing the hare and the counter by 1:



When the hare and tortoise meet, one loop has been completed, length should have been counted by using a counter.



Now, a loop has been detected, the start of that loop has been identified, and the length of the loop has been determined.

Code Walkthrough (Dafny)

```
method floyd() returns (lam:int, mu:int)
{
   var f: seq := [1,2,3,4,2,3,4,2,3,4,2,3,4];

   var tortoise := 1;
   var hare := 2;
   assert 0 < tortoise < |f|;
   assert 0 < hare < |f|;
   while (f[tortoise] != f[hare])
   decreases |f| - tortoise > 0;
   {
      tortoise := tortoise + 1;
      hare := hare + 2;
   }
   mu := 0;
   tortoise := 0;
   while (f[tortoise] != f[hare])
   decreases |f| - tortoise > 0;
   {
      tortoise := tortoise + 1;
      hare := hare + 1;
      mu := mu + 1;
   }
   lam := 1;
   hare := tortoise + 1;
   while (f[tortoise] != f[hare])
   decreases |f| - hare > 0;
   {
      hare := hare + 1;
      lam := lam + 1;
   }
   return lam, mu;
}
```

The code for this program will be in the Floyd method, which will return an integer lambda (the length of the loop) and an integer mu (the starting point of the loop). The very first step is to define a sequence, which is Dafny's version of a list. This is what will be traversed for the program. Initialize the tortoise to 1 and the hare to 2. While the value at the two elements is not equal, the hare will move two items forward while the tortoise moves one item forward. This will continue until either the list ends – in which case a cycle does not exist – or the hare and the tortoise meet. If the hare and the tortoise meet, then a cycle exists.

The next step is to determine where the cycle begins. A counter, mu, is initialized to 0 and the tortoise is moved back to the start of the list. The hare will stay in place. Now, while the value of the two elements (where the hare and the tortoise are pointing) is not equal, move the hare and tortoise 1 step forward each, and increment the mu counter by 1. Thus, the counter will be counting the number of steps the tortoise is taking. The hare and the tortoise will meet at the beginning of the loop. Since the tortoise was travelling from the start of the list, the counter will have kept tract of which element of the list is the start of the loop.

The final step is to determine the length of the loop. This time the tortoise will stay in place while the hare helps the counter. Initialize a new counter lam to 1 and move the hare one step ahead of the tortoise. Now, the tortoise will stay still. At each iteration, the hare will move one step forward and the lam counter will increment by one. Thus, lam will be counting the number of steps the hare takes. When the hare meets the tortoise, the iterations stop. The counter will be holding the value of the length of the loop.

There are quite a few assertions in this code. The first two are asserts, they make sure that both the hare and the tortoise are within the bounds of the list. Beyond that, each of the three while loops have a decrease clause. Recall that a bound function is traditionally in the form N-n>0.In those cases, n is usually being increased within the loop while N is staying constant, so that N-n gets smaller at every run, without dropping below 0. Similarly, the

tortoise/hare is being incremented in each run of the loops, while the length of the sequence, |f|, is staying constant. Thus, |f| - tortoise or |f| - hare should return a smaller value at each run. Once it hits 0, the loop will know to stop running.

Obstacles

The code kept returning an index out of bounds error on any call of the hare or tortoise element of the list:

		Description	Line	Column
⊗	1	index out of range	9	12
⊗	2	index out of range	9	27
⊗	3	index out of range	17	12
⊗	4	index out of range	17	27
⊗	5	index out of range	26	27

```
Dafny 2.1.1.10209
stdin.dfy(9,12): Error: index out of range
stdin.dfy(9,27): Error: index out of range
stdin.dfy(17,12): Error: index out of range
stdin.dfy(17,27): Error: index out of range
stdin.dfy(26,27): Error: index out of range

Dafny program verifier finished with 0 verified, 5 errors
```

This does not make any sense since as far as I realize, the hare and tortoise variables are bounded within the range of the list. Nevertheless, I decided to try calling the first and second elements of the list, rather than the hare and tortoise elements. However, when the variables were replaced with actual numbers, this error was replaced with a different set of errors:

		Description	Line	Column
⊗	1	decreases expression might not decrease	9	4

```
Dafny 2.1.1.10209
stdin.dfy(9,4): Error: decreases expression might not decrease

Dafny program verifier finished with 0 verified, 1 error
```

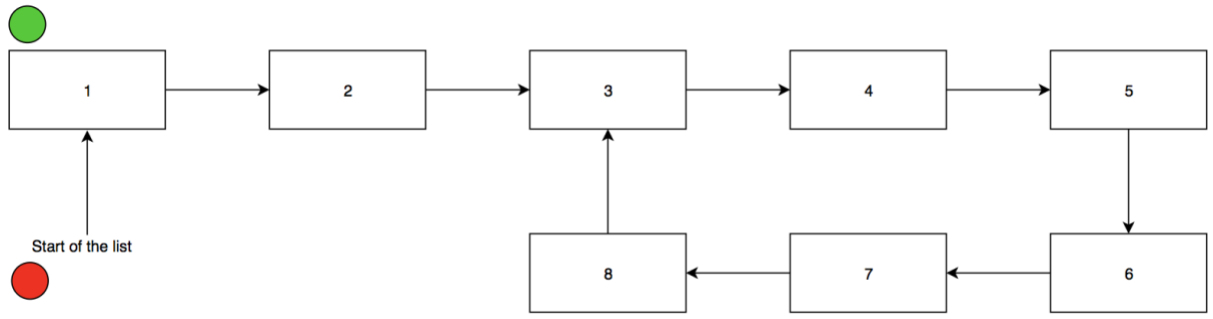
After getting this error I modified a few items, but that led to different errors. I was ultimately unable to decipher the cause of this error.

Brent's cycle detection algorithm

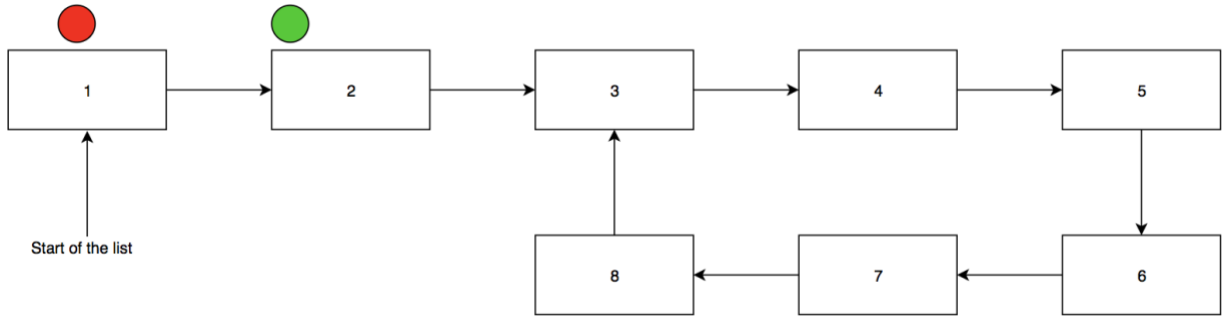
Illustrated Overview

Brent's cycle detection algorithm also works in two main sections. The first is to detect whether a cycle exists. If a cycle does exist, its length is immediately known, since it was calculated while the program was checking for loops. For this version, the hare moves one element at each iteration and the tortoise occasionally teleports to certain parts of the list. A cycle is detected if the two iterators overlap. If no cycle exists, the program is already over. However, if a cycle does exist, the next step is to find the beginning of the loop. This is done by moving the tortoise back to the beginning of the list, and then incrementing the hare iterator by the loop length. Then both iterators move ahead by one hop at each iteration until the tortoise and hare meet. This spot will be the beginning of the list (MBo, 2012).

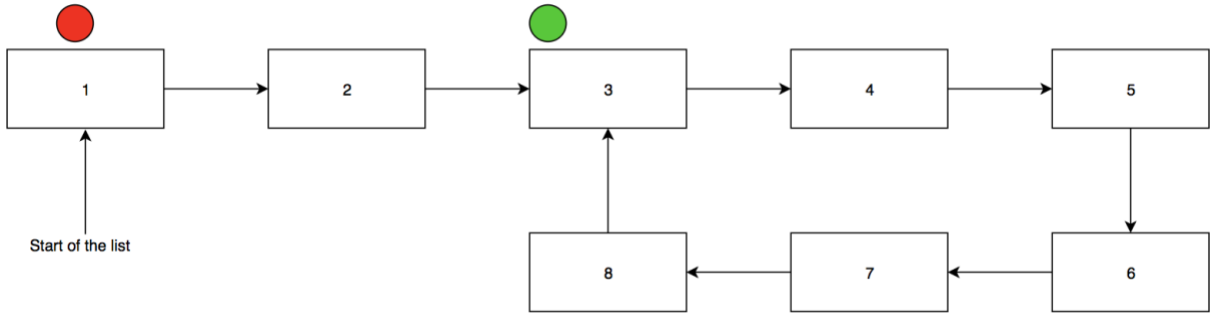
Start tortoise at 0 and hare at 1, power = lam = 1:



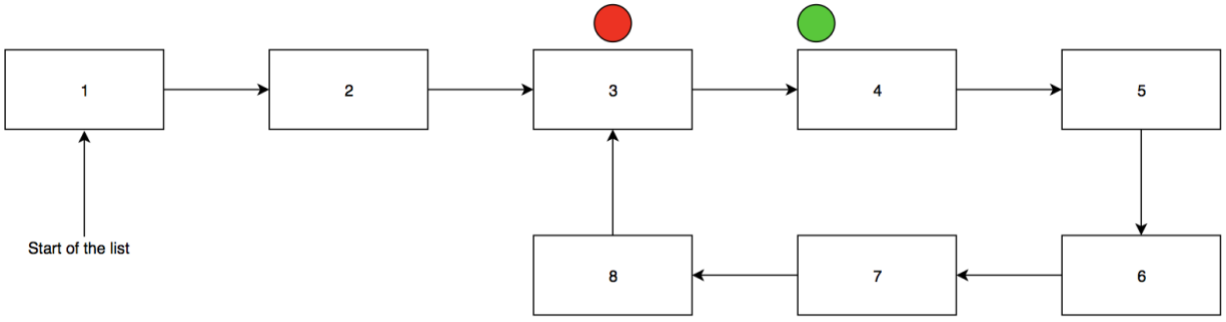
Since power == lam, move tortoise to hare. Double power to 2, set lam to 0. Move hare forward one, increment lam to 1:



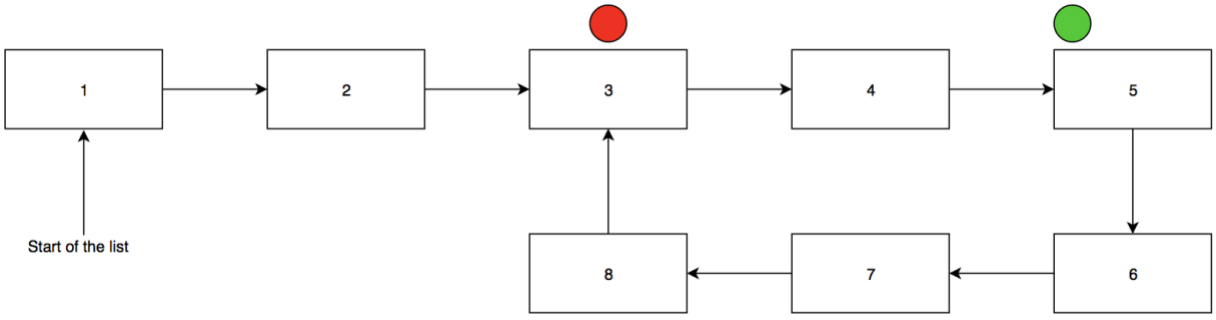
Move hare forward one, increment lam to 2:



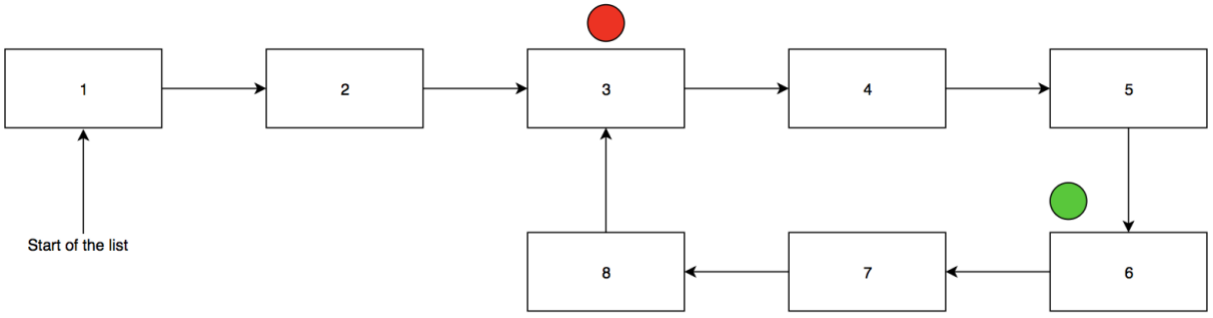
Since power == lam, move tortoise to hare. Double power to 4, set lam to 0. Move hare forward one, increment lam to 1:



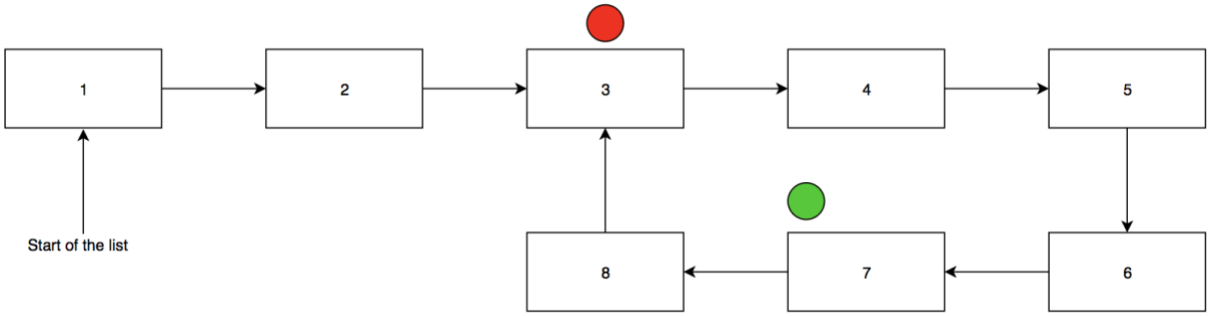
Move hare forward one, increment lam to 2:



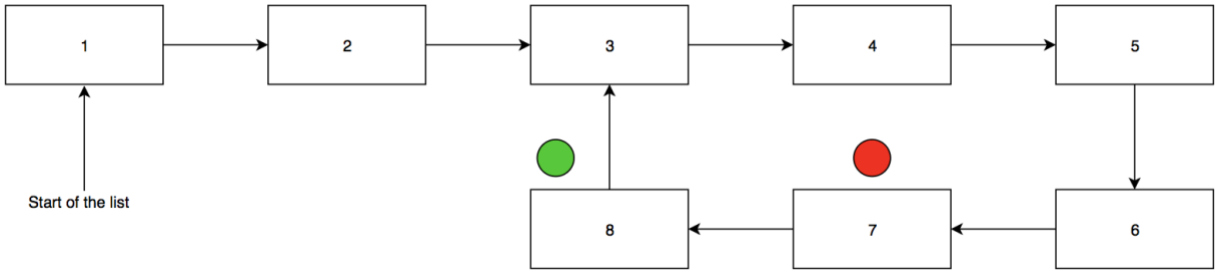
Move hare forward one, increment lam to 3:



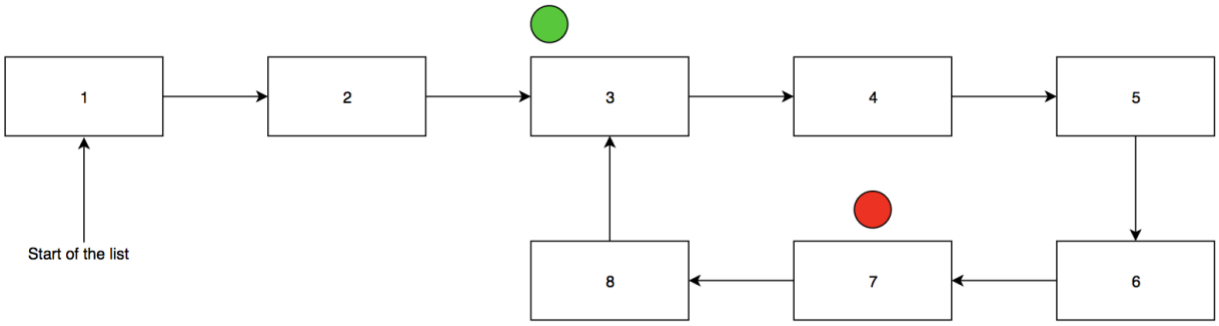
Move hare forward one, increment lam to 4:



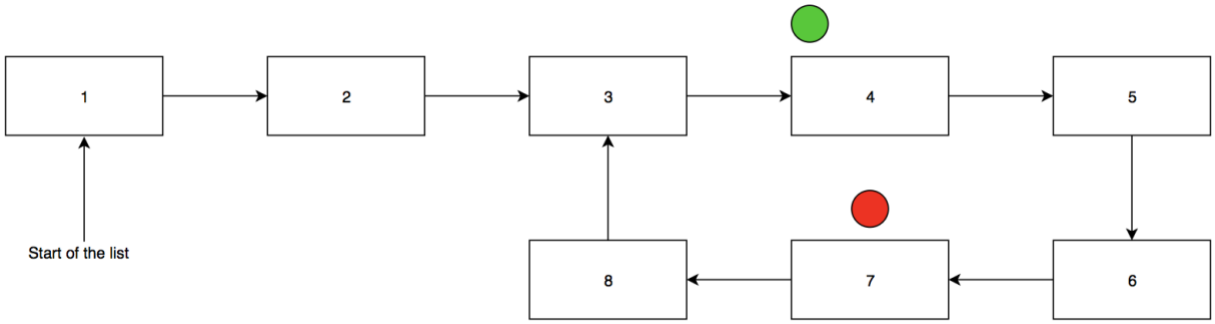
Since power == lam, move tortoise to hare. Double power to 8, set lam to 0. Move hare forward one, increment lam to 1:



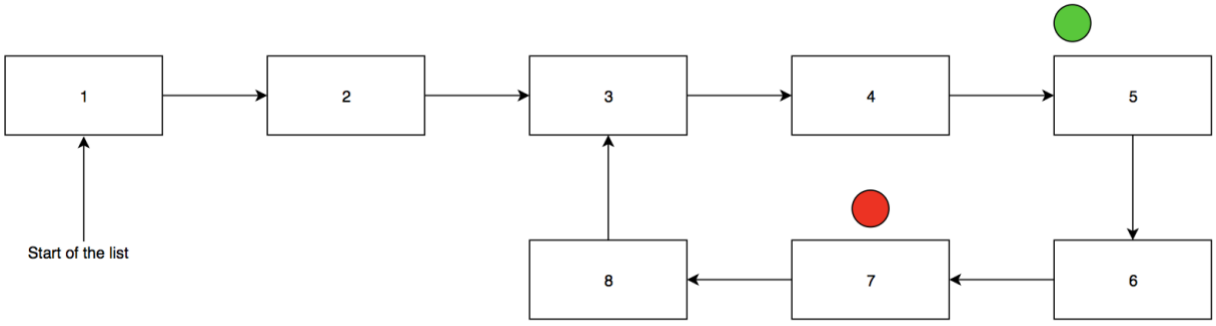
Move hare forward one, increment lam to 2:



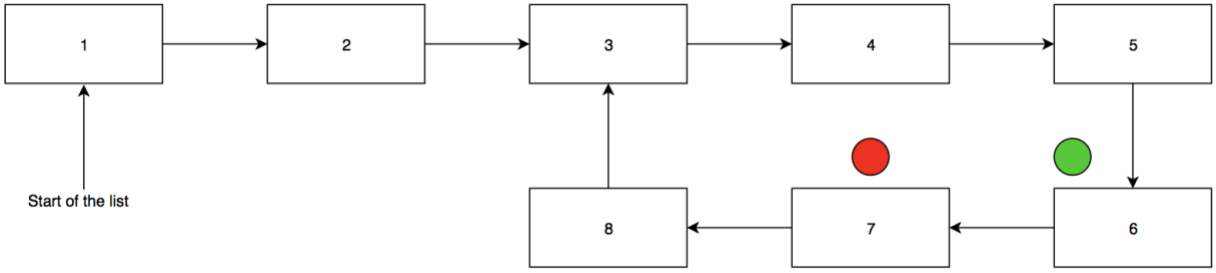
Move hare forward one, increment lam to 3:



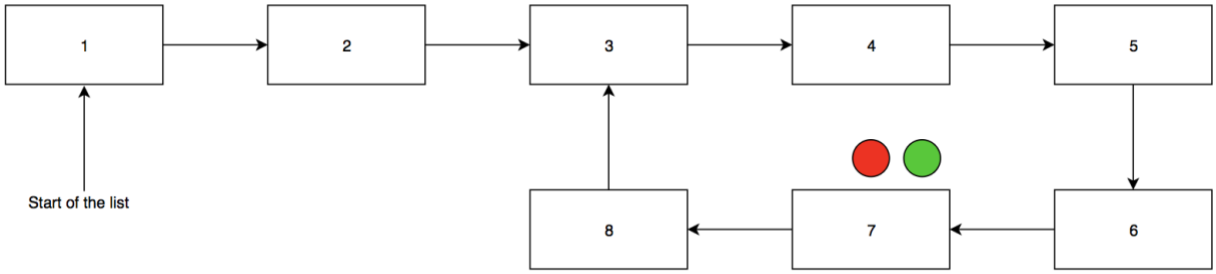
Move hare forward one, increment lam to 4:



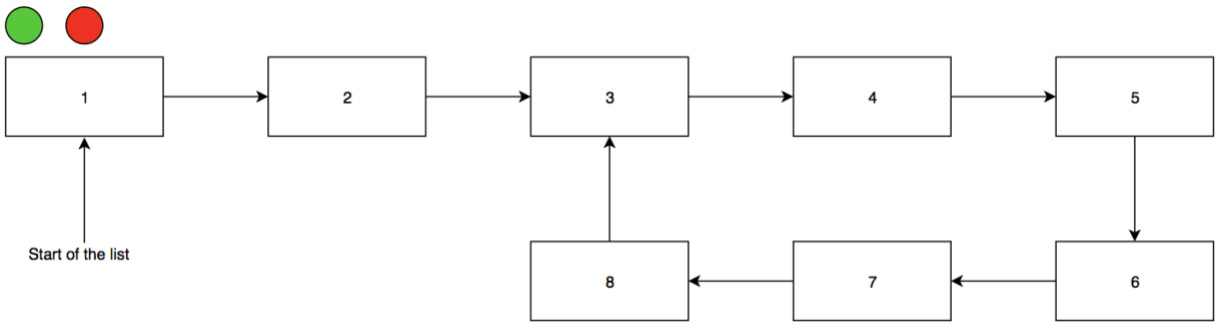
Move hare forward one, increment lam to 5:



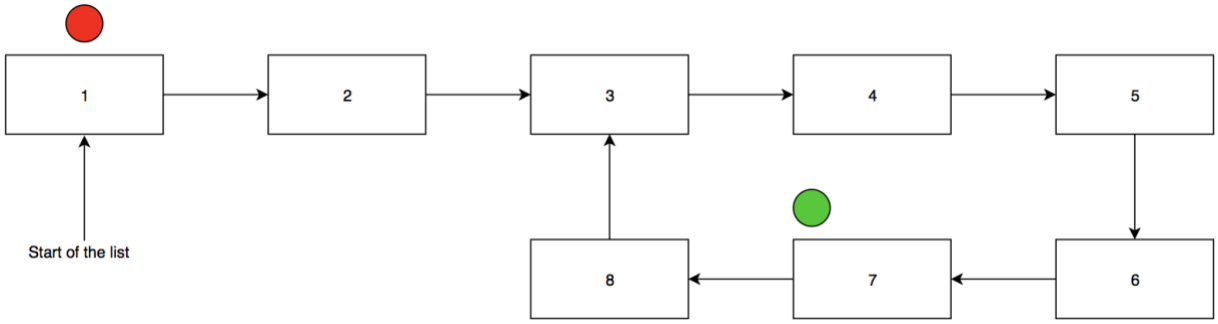
Move hare forward one, increment lam to 6:



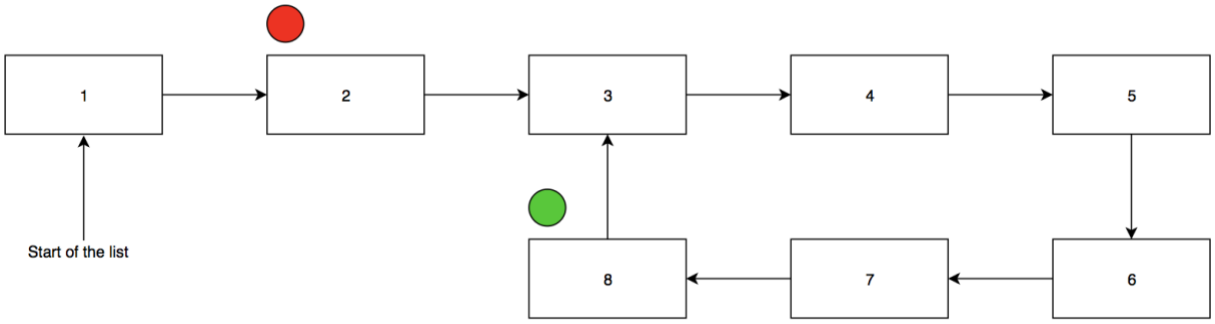
If tortoise and hare overlap, loop exists, lam = loop length = 6. Move tortoise and hare back to beginning of list:



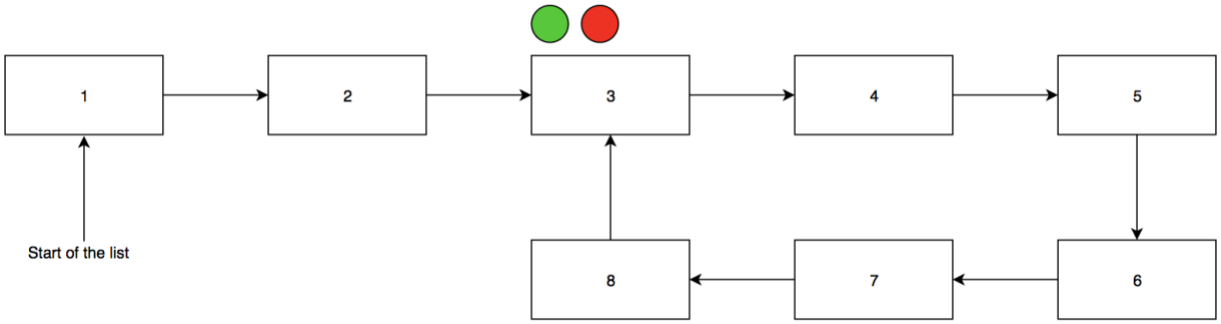
Move hare lam steps forward:



Increment tortoise and hare:



When they meet, that is the starting point of the loop:



*Note: I am aware that this is a **very** extensive walkthrough of this algorithm. However, in my research for this assignment I didn't find any similar walkthroughs. Therefore, this diagram will ideally be quite beneficial to others trying to understand Brent's cycle detection algorithm.

Code Walkthrough (Dafny)

```
method brent() returns (lam:int, mu:int)
{
   var f: seq := [1,2,3,4,2,3,4,2,3,4,2,3,4];
   lam := 1;
   var power := lam;
   var tortoise := 0;
   var hare := 1;
   while f[tortoise] != f[hare]
   decreases |f| - hare > 0;
   {
      if power == lam
      {
         tortoise := hare;
         power := power * 2;
         lam := 0;
      }
      hare := hare + 1;
      lam := lam + 1;
   }
   mu := 0;
   hare := 0;
   var i := 1;
   tortoise := 0;
   while i < lam
   decreases lam - i;
   {
      hare := hare + 1;
      i := i + 1;
   }
}
   while f[tortoise] != f[hare]
   decreases |f| - hare > 0;
   {
      tortoise := tortoise + 1;
      hare := hare + 1;
      mu := mu + 1;
   }
   return lam, mu;
}
```

The code for this program will be in the Brent method, which will return an integer lambda (the length of the loop) and an integer mu (the starting point of the loop). The very first step is to define a sequence, which is Dafny's version of a list. This is what will be traversed for the program. Initialize lam to 1 and a new variable, power, to 1 as well. Start the tortoise at position 0 and the hare at position 1. While the element at tortoise and the element at hare are not equal but power and lam are equal, move the tortoise to the hare's location, double the power, reset lam to 0, move hare one position ahead, and increment lam. If power and lam are not equal, simply move hare one position ahead, and increment lam. This ends one iteration of the loop. The loop will end when the element at tortoise is equal to the element at hare. Lam will be the distance between the hare and the tortoise, which will also be the length of the loop when the two overlap.

The tortoise and the hare are both brought back to the beginning of the list. Initialize variable mu to 0. Move the hare lam distance away from the tortoise. Then, as long as the elements at the hare and the tortoise are not equal, iterate both and increment the counter for mu. Wherever the hare and the meet is the beginning of the loop. Mu will have kept track of which element it is.

This algorithm does not have any assertions besides the decreases clauses. Recall that a bound function is traditionally in the form N-n>0. In those cases, n is usually being increased within the loop while N is staying constant, so that N-n gets smaller at every run, without dropping below 0. Similarly, the tortoise/hare/i iterator is being incremented in each run of the loops, while the length of the sequence, |f|/lam is staying constant. Thus, |f| - hare and lam – i should return a smaller value at each run. Once it hits 0, the loop will know to stop running.

Obstacles

The code returned quite a few errors for which I could not decipher a fix:

	Description	Line	Column
1	index out of range	8	11
2	index out of range	8	26
3	decreases expression might not decrease	30	4
4	index out of range	30	11
5	index out of range	30	26

```
Dafny 2.1.1.10209
stdin.dfy(8,11): Error: index out of range
stdin.dfy(8,26): Error: index out of range
stdin.dfy(30,4): Error: decreases expression might not decrease
stdin.dfy(30,11): Error: index out of range
stdin.dfy(30,26): Error: index out of range

Dafny program verifier finished with 0 verified, 5 errors
```

This does not make any sense since as far as I realize, the hare and tortoise variables are bounded within the range of the list. I decided to focus on the error that was asking for a decreases clause. My solution was to add an assertion before the final while loop to look as follows:

```
assert f[tortoise] != f[hare];
while f[tortoise] != f[hare]
decreases |f| - hare > 0;
{
   tortoise := tortoise + 1;
   hare := hare + 1;
   mu := mu + 1;
}
```

However, this led me to the following error:

	Description	Line	Column
1	index out of range	8	11
2	index out of range	8	26
3	assertion violation	30	23
4	index out of range	30	27
5	index out of range	31	11

```
Dafny 2.1.1.10209
stdin.dfy(8,11): Error: index out of range
stdin.dfy(8,26): Error: index out of range
stdin.dfy(30,23): Error: assertion violation
stdin.dfy(30,27): Error: index out of range
stdin.dfy(31,11): Error: index out of range

Dafny program verifier finished with 0 verified, 5 errors
```

I ultimately was unable to find an alternative solution which didn't result in more errors.

Test Cases

Selection Algorithm

1. When the sequence is: “var s: seq := [1,2,3,4,5];” the output is 3:

```
Dafny 2.1.1.10209

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
Running...

3
```

2. When the sequence is: “var s: seq := [5,6,7,8,5,4,5,9,2];” the output is 5:

```
Dafny 2.1.1.10209

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
Running...

5
```

3. When the sequence is: “var s: seq := [9,8,6,4,2,1,6,8,9,0,7,4,2,3];” the output is 6:

```
Dafny 2.1.1.10209

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
Running...

6
```

Each of these three test cases returned the expected result. In each sequence, the value at the exact middle element is returned. Thus, the selection algorithm passes its test cases.

Cycle Detection

Code does not work, but a hypothetical test case might be as follows (assuming the code works ideally the way it was intended to work):

1. When the sequence is: “var s: seq := [1,2,3,4,5];”
   the output should be false

2. When the sequence is: “var s: seq := [1,2,3,4,5,1,2,3,4,5];”
   the output should be true

3. When the sequence is: “var s: seq := [1,2,3,4,3,4,3,4,3,4];”
   the output should be true

Floyd’s cycle detection algorithm

Code does not work, but a hypothetical test case might be as follows (assuming the code works ideally the way it was intended to work):

1. When the sequence is: “var s: seq := [1,2,3,4,5];”
   the output should be false

2. When the sequence is: “var s: seq := [1,2,3,4,5,1,2,3,4,5];”
   the output should be mu = 4th element and lam = 5

3. When the sequence is: “var s: seq := [1,2,3,4,3,4,3,4,3,4];”
   the output should be mu = 3rd element and lam = 2

Brent’s cycle detection algorithm

Code does not work, but a hypothetical test case might be as follows (assuming the code works ideally the way it was intended to work):

1. When the sequence is: “var s: seq := [1,2,3,4,5];”
   the output should be 3:

2. When the sequence is: “var s: seq := [5,6,7,8,5,4,5,9,2];”
   the output should be mu = 4th element and lam = 5

3. When the sequence is: “var s: seq := [9,8,6,4,2,1,6,8,9,0,7,4,2,3];”
   the output should be mu = 3rd element and lam = 2

Relationship between Algorithms

The selection algorithm was done to introduce the concept of the tortoise and the hare, and demonstrate how the two iterators work. It does not have any direct relation to cycle detection.

Of the three algorithms dealing with cycle detection, each algorithm introduced was an improvement of the previous algorithm. Floyd's algorithm took the basic cycle detection two steps further by locating the start of the loop and calculating the length of the loop. Brent's algorithm, though it did not add further functionality to Floyd's algorithm, increased efficiency. Brent claims it can be up to 36% faster for relevant algorithms than Floyd's algorithm (Brent's Cycle Detection Algorithm (The Teleporting Turtle), 2010).

Given the list of potential applications of cycle detection, these algorithms do currently – and will continue to – prove useful to many programmers. However, Brent's algorithm is the most efficient and has the most functions. In my opinion, after extensive research on these algorithms, instances of Floyd's algorithm should be replaced with Brent's algorithm.

Conclusion

The concept of working with two iterators is a genius one, as it not only makes certain algorithms more efficient, it also provides functionality that would have been impossible with only one iterator. For example, in the cycle detection algorithm, a single iterator would make it very difficult to find a loop within a list. The lone iterator could enter the loop, and then be stuck in the loop forever. Introducing a second iterator at a different speed ensured a collision, and thus a fool proof way of returning the information that the loop does indeed exist.

The algorithms are all correct, because although this report verified them informally, the overall concept is very common. The formal proofs for the correctness of each of these algorithms already exists online. They will be supplemental to the proofs above.

This project helped me learn about invariants and about how bound functions translate over to Dafny code. From a non-verification standpoint, I also learned the general differences behind the execution of each of these algorithms. This will ideally prove to be useful in future interviews.

Additionally, I believe that this report pushed me to contribute to existing literature on this topic. When I was doing research for the Brent cycle algorithm, I had a hard time understanding how it worked. The walkthroughs I found online consisted of some code and very short synopses of how they work. A diagram or a more detailed explanation of the steps taken in the algorithm would have proved to be very helpful in helping me understand this concept for the report.

Therefore, the diagrams I created for my report will ideally prove helpful to many others who are working on problems within this topic.

Works Cited

Brent's Cycle Detection Algorithm (The Teleporting Turtle). (2010, January 22). Retrieved from Siafoo: http://www.siafoo.net/algorithm/11

Cycle detection. (2018, February 16). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Cycle_detection#Floyd's_Tortoise_and_Hare

Isaacson, D. (2010, January 22). *Floyd's Cycle Detection Algorithm (The Tortoise and the Hare)*. Retrieved from Siafoo: http://www.siafoo.net/algorithm/10

Jackson, D. (2012). *about alloy*. Retrieved from alloy: a language & tool for relational models: http://alloy.lcs.mit.edu/alloy/

Leino, R. (2018, April 10). *dafny*. Retrieved from GitHub: https://github.com/Microsoft/dafny

MBo. (2012, May 29). *Brent's cycle detection algorithm*. Retrieved from Stack Overflow: https://stackoverflow.com/questions/10798012/brents-cycle-detection-algorithm

More, N. (2015, September 2). *Detect a loop in a linked list and find the node where the loop starts*. Retrieved from iDeserve : https://www.ideserve.co.in/learn/detect-a-loop-in-a-linked-list