– Independent Work Report Fall 2018 –

Generating Functions for Linear Recurrences Using Program Synthesis

Nebil Ibrahim
Adviser: Aarti Gupta

Abstract

This paper shows the results of attempting to generate formulas for linear recurrences via the programming language Sketch. Sketch is a Java-based programming language that allows for holes in a program which are places where Sketch will try to put values to satisfy user specifications. Methods to produce the nth term in a linear recurrence were implemented recursively as well as iteratively with doubles and integers. The goal of this paper is to present the pros and cons of both implementations of linear recurrences to see which one is favored more by Sketch for the desired result and to see if either are viable in practice.

1. Introduction

There are several different problems in computer science with solutions that have different time and space complexities. For example, algorithms that sort arrays have different time complexities that range from O(n log(n)) such as merge sort and quicksort to O(n^2) such as selection sort and insertion sort. COS 226 instilled the idea in me that efficiency of computation in both time and space is not to be neglected when applied on a large level. COS 340 was my first introduction to linear recurrences and their applications. It amazed me that a problem like finding the nth term in a linear recurrence which is often implemented as taking exponential time has an implementation that can take linear time.

There are several uses of recurrences in probability, computer science, and other engineering fields. Analysis of the complexity of algorithms is often done by computing a closed form of linear recurrences that describe recursive algorithms[3]. In probability theory, there are events whose probability are linearly dependent on previous events. Similarly, linear recurrences are used to determine populations based on mortality and birth rates. Any function whose value can be determined by summing up previous function outputs with some constant multiplier can be represented as a recurrence.

For example the definition of the nth Fibonacci number is

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0$ and $F_1 = 1$. One algorithm to find the nth term is to recursively call Fibonacci until the base cases are reached. If this algorithm is implemented without the use of a dictionary to cache results, the time complexity is O(2^n) and the space complexity is O(n) regardless of if there is a dictionary in use. The iterative implementation of Fibonacci takes O(n) and only uses constant space. The latter of these is implementations is about as good as you can ask for, but its optimization of the former relies on the recursive definition of the nth term being known.

If for example, the only thing known about the recursive definition of a linear recurrence is that it is the sum of two terms that previously occurred in the sequence and an example of an input and output (such as the F(5) = 11), the problem becomes much harder. Finding an optimized iterative form of Fibonacci becomes much harder if all that is known is that appears in the form

$$F_n = F_{n-x} + F_{n-y}$$

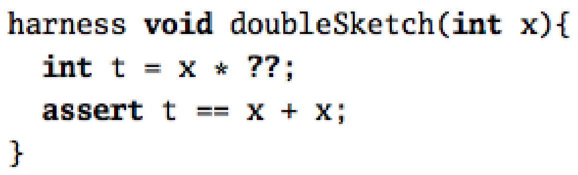
where x and y are integers. This also comes with the assumption that the coefficients of the linear recurrence are one. If the assumption is that the coefficients are one is taken away, then the sequence is of the form

$$F_n = a \cdot F_{n-x} + b \cdot F_{n-y}$$

where a and b are real numbers and x and y are integers. Another assumption made was that there are only two terms in this linear recurrence when there could be more. If this assumption is removed, the sequence is of the form

$$F_n = a \cdot F_{n-x} + b \cdot F_{n-y} + c \cdot F_{n-z} ...$$

In general, finding a recursive definition of a linear recurrence, let alone an iterative optimization, is a hard problem when there is limited information about a linear recurrence. This is where program synthesis can be used to aid in solving for the unknowns of a linear recurrence such as the coefficients, base cases (at which term they occur as well as what they return), and the number of terms. Program synthesis is a subfield of computer science where programs generate other programs by satisfying specifications given by a user. To generate a function that can be used to find the nth term of linear recurrence, I used the Sketch programming language [1]. Its syntax is very similar to that of Java and C with its defining difference being that it allows for holes (represented by "??") that it can solve for when given a specification in the form of an assertion statement such as in the example pictured from the Sketch manual. The holes were then used to allow for generic implementations of a linear recurrence in which some used iteration and used others recursion to observe the trade offs and benefits of each implementation.



```
harness void doubleSketch(int x){
  int t = x * ??;
  assert t == x + x;
}
```

Figure 1: This is an example of a Sketch program from the sketch manual. [2]

2. Related Work

2.1. Program Synthesis

An example of applied program synthesis can be observed in Microsoft Excel. Program synthesis is used to predict strings based on input and output example provided by a user [4]. One example provided is formatting names the same way after being provided with a few examples even when they are provided in a different format. The input Prof. Kathleen S. Fisher becomes Fisher, K. and the input Bill Gates, Sr. becomes Gates, B. This idea was implemented as a feature known as flash fill in Excel. The tool was received well by many and saves a lot of time for users that do not want to go through by hand and correct the differences in formatting from other users as it can also be used to correct formatting for dates, phone numbers, etc [5].

As far as work on optimization linear recurrences goes, there is not much research in the field. In a typical application of a linear recurrence, the recursive definition is already known or there is enough information about the recursive definition that a closed form is simple enough to understand. When a linear recurrence needs to be solved a typical method used is as follows. As an example, the Fibonacci sequence can be represented by $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$. For each term in the equation, it can be converted into a polynomial term whose exponent is equivalent to the argument. $x^n = x^{n-1} + x^{n-2}$ If this simplified it becomes $x^2 = x + 1$. The roots of the polynomial are

$$\frac{1+\sqrt{5}}{2} \text{ and } \frac{1-\sqrt{5}}{2}$$

. The closed form of the linear recurrence is then known to be of the form

$$F_n = a \cdot \phi^n + b \cdot \psi^n,\ \phi = \frac{1+\sqrt{5}}{2},\ \psi = \frac{1-\sqrt{5}}{2}$$

The base cases can then be solved for by substituting them into the equation. The closed form of the linear recurrence then becomes

$$F_n = \frac{\phi^n}{\sqrt{5}} + \frac{\psi^n}{\sqrt{5}},\ \phi = \frac{1+\sqrt{5}}{2},\ \psi = \frac{1-\sqrt{5}}{2}$$

While the closed form has a much better time complexity of O(n), the most computationally expensive part of getting to the closed form is finding the roots of the polynomial. There has been extensive work to develop algorithms to solve for roots of polynomials by approximations. Their have been algorithms developed take polynomial time and are similar to Newton's Method[6]. [1]

[1]The actual reported time was O(n(n$^2 + nlogMea(P) + logM(Disc(P)^{-1})$))

This at the very least means that it is possible to find the bases of the exponential functions that make up a linear recurrence.

3. Approach

The Fibonacci function serves as a good linear recurrence to start from because of its well-understood behavior, simplicity, and several implementations to work from. The implementations that were used are the recursive implementation that takes exponential time and linear memory, the iterative implementation that uses three variables to iterate through n times for the nth term, and the iterative implementation that replicates the closed form of the Fibonacci function.

The recursive implementation allows for holes to be placed at the base cases $F_0 = ??$ and $F_1 = ??$, when the base cases occur $F_{??} = 0$ and $F_{??} = 1$, and the return statement $F_{n-??} + F_{n-??}$.[2] The implementation is $O(2^n)$ time and O(n) space. The locations provided here serve as good places to use as holes since they are possible unknowns a user would have regarding a linear recurrence. At the very least there should be an input and output known by the user that they can provide as a specification for the program.

The second implementation is an iterative one that uses four variables: the value of F(0), the value of F(1), the next term in the sequence and the variable that increments over the whole loop. This implementation is tailored to the Fibonacci sequence as the constant number of variables does not allow for a linear recurrence that is dependent on more than two terms to be used. It can be modified by adding more variables, but the more variables there are, the harder it is for Sketch to solve for the holes of the program. Exploring the limits of this implementation will provide insight into how Sketch can fill in the holes for an implementation that takes O(n) time.

The third implementation is also an iterative one, but it finds an approximation of the bases of the exponential function which make up the closed form. The benefit of this implementation is that it takes O(n) time and the bases of the exponents are helpful to help the user understand the behavior of the linear recurrence.

[2]The "??" is where a hole would be placed in Sketch. Just because there are "??", does not mean they represent the same value.

The goal is to identify the benefits and drawbacks of each implementation to see if any of them are viable when there are holes placed in the program and the user provides a specification for the linear recurrence to hold. Ideally, the user would be able to just provide the number of terms required to be added and at least that many input and output examples. If this could be achieved with the third implementation, then the user would gain more useful information about the behavior of the recurrence than it would from the other two implementations.

4. Implementation

4.1. Iterative Implementation (tailored to Fibonacci)

The first implementation to look at was the iterative one that adds the previous two terms in sequence to get the next one. Even though I was not sure if there was much to learn from this implementation, it became clear there was a lot to learn from the kinds of solutions Sketch would return. In one trial all four variables were set as holes and the only assertion statement given was that the fourth term of the Fibonacci sequence equals five.[3] Sketch found the results quickly, but the variables were not initialized to what they were desired to be. The variable that was to be returned was initially set to the return value of five. The iterator variable was then set to nine and because the while loop is only entered if the iterator variable is less than the argument to the function, which in this case was four, the function just returns five as the answer.[4] A similar observation was made with assertions checking the third and fifth Fibonacci numbers. The iterator would be set at a number larger than what was needed to enter the for loop and then it would never enter the loop. While this answer is correct, this problem is known as under specification. It occurs when there is not enough specification for the program to fulfill so it returns an answer that satisfies the given criteria, but will not work for general inputs.

There are combinations of holes for which Sketch can generate a function that satisfies the

[3]The variables were referenced above. They are the base cases (0 and 1), the next term in the sequence (F(n)), and the iterator for the loop.

[4]The base cases did not appear in the result, but they would have been irrelevant since they only matter once the loop is entered which did not happen in this case.

constraints. With just the base cases as holes, Sketch has no problem finding the holes to be zero and one. This is probably due to the iterator not being provided as a hole. Since there are a fixed number of iterations, there is only one set of solutions which can satisfy the constraint which are zero and one. In one trial where the only variable that was not a hole was the nth term in the sequence and the assertion was for the eighth term in the sequence, Sketch returned a program that satisfied the assertion but filled in the holes with values that would not work on general inputs. The iterator was set to five and the two base cases were set to two and ten. This illustrates the importance of the iterator in finding the correct solution from the program. If the iterator is left as an unknown, Sketch can find a value for it and then fit the bases accordingly to provide an answer that fits the specifications provided but not work in the general case.

The specifications for this program were in the form of assertions that compared the output of the program with holes to one without holes to make it easier to test other inputs and outputs.[5] This also allowed for loops to iterate through assertions with different inputs to easily test how well the solver could fill in the holes and if adding more assertions helped. However, there is a limit to the number of assertions that Sketch will allow. Even if the assertions should be satisfiable such as asserting the third Fibonacci number as three and the fourth Fibonacci number as five, Sketch could return an error called unsatisfiable assertion. Sketch might think the assertions are too strict so as soon as the program is executed it will return this error. This does not happen every time there are several assertions. In fact, it does not even happen every time a program is run. If for a particular program this does happen then the program has the potential to run indefinitely when the error does not immediately show up. The implications of this are that even if a user has multiple data points for a linear recurrence, they might be unable to use all of the data points due to the unsatisfiable assertion error and be restricted to using a few data points which could lead to under specification.

4.2. Recursive Implementation

The recursive implementation allows for several more holes than the iterative one above, so there is a lot more freedom to observe its behavior. When almost every part of the function was turned

[5] As an example (assert fib(7) == realfib(7))

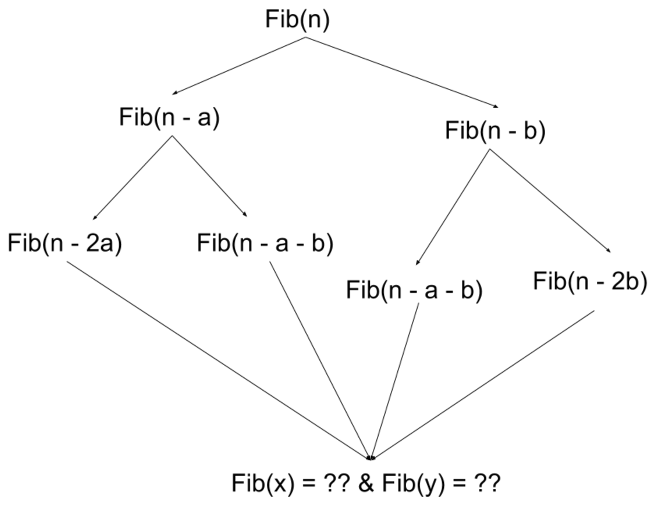


Figure 2: This is a representation of what happens for Sketch to generate a program when every variable is a hole.

into a hole such that it was in the form $F_n = F_{n-x} + F_{n-y}$ with unknown base cases including when those base cases occur, the solver had no way to find the desired solution. This gives the function six holes.[6] When the specification was for the fifth and fourth Fibonacci number, Sketch returned a program whose base cases were at those terms and had the correct value at those terms, but all of the other holes had wildly incorrect values in place. For example in a trial with these specific holes, the x value as shown in the equation was 29 and the y value was three. Sketch seems unable to handle so many holes as having built a function as generic as this implementation means several assertion statements are needed. Having the number of assertions required causes Sketch to simply reject the program at run-time. The only time the program is ever willing to run is when there are only two assertions to be made in which case a similar problem occurs as mentioned previously where the function generated would only return the correct value for the base cases.

When the holes in the program are reduced to just figuring out which two terms to sum up in order to receive the next term in the sequence, Sketch does fill in those holes correctly. In order to do this accurately, Sketch needs to know at least two assertions in order to not be subject to under specification. For any user who is aware of the base cases of their sequence, so long as they have

[6]Two of the holes are for when the base cases occur. Two of them are for what value to return to those base cases. Two of them are for figuring out which term in the sequence.

the knowledge of some terms in the sequence (in the case of Fibonacci at least two), then they will be able to find the terms in the sequence necessary to get the next term.

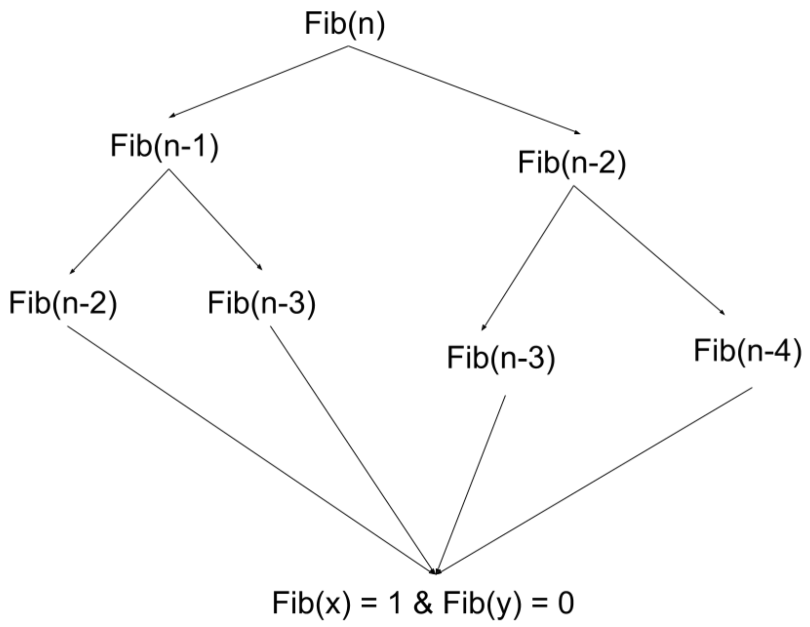


Figure 3: This is a representation of what the program has to do in order to solve for when base the cases occur.

Just filling in holes does not necessarily enable the Sketch to fill the rest of them any better. If the only holes are those for when the base cases occur, even though Sketch is provided with what to return in those cases, it still fails to fill in the hole. This might be due to the fact that it easier for Sketch to solve for base cases because it does not have to solve for them until it is at the end of recursion. On the other hand, if the holes are when the base cases occur, Sketch does not know when it will have to find the solutions to these holes and therefore throws an unsatisfiable assertion error.

4.3. Iterative Implementation (Closed Form)

The closed form of Fibonacci is the ideal implementation for calculating the nth term as it O(n) time and uses constant space. Finding the bases of the exponents is the most computationally expensive part. In order to find them, doubles will have to be used as no integer can accurately approximate the

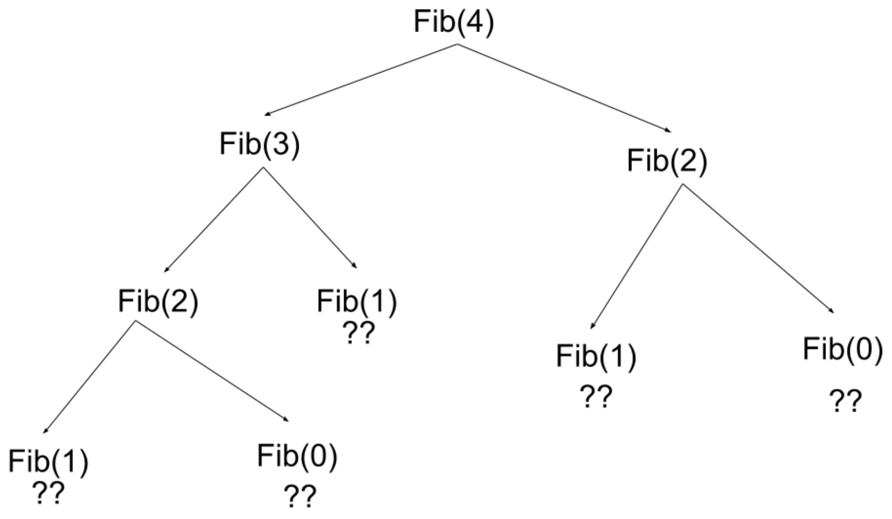


Figure 4: This is a representation of what the program has to do in order to solve for the base cases.

nth term of the Fibonacci sequence. Since Sketch does not support doubles as holes, the holes have to be represented as the ratio of two integers.[7] For every double included in this implementation, the solver has to find the holes for two numbers as opposed to one. This would be much more computationally expensive because there are very few pairs of integers whose ratio is accurate enough to find the nth term of a sequence.[8]

In order for this function to solve for any holes, the coefficient of the exponential part of the closed form could not be left as a hole. The closed form of the Fibonacci function has both terms divided by the square root of five. On trials when it was left as a hole, the program never terminated. This was probably due to Sketch not knowing how to handle so many dependent integer pairs. The square root of five was then placed as a double instead of a pair of integer holes to observe if Sketch could then solve for the other holes.

Even after providing the square root of five, the program still did not manage to terminate. Finding the holes of four integers (two for each of the bases of the exponents) was still too difficult for Sketch

[7] Professor Armando Solar-Lezama, a professor at MIT who made the Sketch language informed me of this on a visit to Princeton.

[8] Of course any approximation will not be able to accurately find the nth term for very large n, but the Fibonacci function was only tested on relatively small inputs.

to find the holes for. To remedy this issue, I used a property of the closed form of Fibonacci. The base of the exponent in the second term of Fibonacci is less than one i.e. $\frac{1-\sqrt{5}}{2}$ and the coefficient it is multiplied by is the square root of five so the term will always be less than one. Because of this, it suffices to approximate the closed form of Fibonacci as

$$F_n = floor(\frac{\frac{1+\sqrt{5}}{2}^n}{\sqrt{5}})$$

This reduced the number of holes to solve for from four to two allowing Sketch to finally return a program that satisfied the given specifications.

Because this implementation returned a double the assertions could not be as constrained as they were in previous implementations. The constraints had to have error bounds included in order for Sketch to satisfy the specifications which would be near impossible other. Professor Armando Solar-Lezamma also informed me that the error bounds in the asserts could not be too small. Otherwise, Sketch would not be able to provide any ratio of integers from its natural search range to provide as an answer. Sketch does provide the option to use a flag to increase the range of integers to search over, but this also increases the amount of time the function would look over integers. To combat this, I increased the bounds from +/- 0.001 to +/- 0.1. In order for the function to return a better approximation of the golden ratio larger inputs would instead be provided with the same error bounds. The limit of the ratio of integers must head towards the golden ratio as larger input output assertions are provided as specifications.

5. Evaluation

5.1. Results of Closed Form Implementation

In order to test the effectiveness of the closed form implementation, the function was provided with different specifications as arguments. As stated earlier, larger arguments to the function require a much more accurate approximation of the golden ratio in order for the it to return the correct value for the nth term. As the argument size increase the range of ratios the are acceptable decrease. This

Argument (n)	Error Bounds (+/-)	Elapsed Time (seconds)	Percent Error (to Golden Ratio)
1	0.1	0.298	27.73
1	0.05	0.293	31.33
3	0.1	0.3	15.37
3	0.05	0.243	16.74
5	0.1	25.046	9.87
5	0.075	26.203	9.87
7	0.1	NA	NA

Table 1: Result from testing the closed form implementation of Fibonacci.

did of course increase the amount of time the function took, but the time increased much more than expected.

For small values the function ran quickly. The function returned a value that was within the error bounds which were set at both +/- 0.1 and +/- 0.05 respectively. The function would sometimes fill in the integer holes with consecutive Fibonacci numbers such as 13 and 8 which is an accurate way to represent the golden ratio because

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \frac{1+\sqrt{5}}{2}$$

Sketch would not be able to consistently provide pairs of integers with this property because of the limited range of integers it is allowed to use. There only so many terms of the Fibonacci sequence less than its default search range which goes up to 32. Using the flag to increase the search range of integers doubles the range, so allowing sketch to search over more integers would lead to a more accurate result, but would also dramatically increase search time to the point where it is no longer possible to fill in the holes in a reasonable amount of time.

When the argument was five, the function took much longer to produce a result. What initially did not appear to be exponential time, was now definitely greater than any reasonable polynomial time I was hoping for. The function did however return a much more accurate approximation of the golden ratio which was 1.77 and has an error of only 9.87% error from the golden ratio. For the input of size five, the bounds could not be tightened to +/- 0.05 because Sketch could not find a pair of integers which were that close to the fifth Fibonacci number. Instead, the bounds were tightened to +/- 0.75 for which Sketch failed to return a more accurate result but still took slightly more time to find.

Inputs greater than five could simply not be handled by Sketch. The program would either run without ever terminating or the program would almost instantly conclude the specifications were unsatisfiable. The only way the program would terminate is if it were provided with large error bounds. This defeated the purpose of incrementing the argument size since Sketch could not return a more accurate value than the one return when the specification input was five.

5.2. Results of Iterative Implementation

Argument(n)	Elapsed Time (sec)
0	0.145
2	0.147
4	0.304
6	0.644
8	6.945
10	95.281

Table 2: Results from testing the iterative implementation tailored to Fibonacci.

The iterative form tailored to Fibonacci runs better on every assertion that was also run for the implementation that replicates the closed form. For inputs greater than size eight, a flag had to be raised so that Sketch could unroll the loop more than eight times. A jump in run time was seen from an input specification of six to eight and similarly from eight to ten. This might be due to how Sketch performs when it is near or beyond its default unrolling limit. If a user only knew large input and output of a linear recurrence, even this method would not run in reasonable time.

5.3. Results of Recursive Implementation

The recursive implementation worked quite well when only one of the three possible places to put holes i.e. (when the base cases occur, the base cases, and what two terms in the sequence need to be added for the nth term) had holes. Under specification was most prominently a problem with this implementation. Due to the holes placed in the conditions of the if statements there was often more than two specifications required to get the right answer since the were two if statements. For any linear recurrence whose nth term is the summation of m previous terms, the recurrence would need at least m terms to to not be subject to under specification.

Unspecified	Specified	Elapsed Time (seconds)
Return Statement	Base Cases, When Base Cases Occur	0.257
Base Cases	Return Statement, When Base Cases Occur	0.152
When Base Cases Occur	Return Statement, Base Cases	0.136

Table 3: Results from testing the iterative implementation tailored to Fibonacci.

5.4. Comparison

The results illustrate how difficult it is to solve for the closed form a sequence, even with a sequence as simple as the Fibonacci sequence. Considering that the implementation was provided with values a user would not know such as the fact that one of the bases of the exponents is not necessary to find the nth term and that the whole formula is to be divided by the square root of five, it still performed poorly. Having approximated the golden ratio with an error of 9.87%, it would be useless to use the approximation for large inputs. The iterative form proved much better, but it was also tailored to the Fibonacci sequence so it would prove difficult to apply to general linear recurrences. At the very least, it is viable to use Sketch to solve for a few missing holes using the recursive implementation.

6. Conclusion

The main problem with the closed form implementation was Sketch's lack of support for doubles as holes. As far as I am aware there are not any languages that support doubles as holes because there are too many possible values. My next efforts will be focused on figuring a method to use doubles as holes that is not computationally expensive on any program synthesis language.

This paper focused mainly on the Fibonacci sequence, but there are plenty of behaviors that other linear recurrences have that are not expressed by Fibonacci. There are linear recurrences whose nth term is dependent on (n/2)th term in which case the return statement which returns the (n - a)th term (where a is a constant) would not work. If the closed form of the base case has exponents with the same base, then they may appear in the form $F_n = a_0 \cdot b^n + a_1 \cdot x \cdot b^n + a_2 \cdot x^2 \cdot b^n \ldots$. There are also closed forms that have bases of exponents which are complex. Addressing more types of linear recurrences would require more functions and using objects to represent complex numbers, but would be an interesting task to solve.

I hope others will try to make languages similar to Sketch that do support doubles as holes. If a language is made that can handle more assertions as specification and reason through them better than a closed form implementation of finding the nth term of a linear recurrence becomes more viable.

7. References

[1] Sketch Programming Language by Professor Armando Solar-Lezama

[2] Solar-Lezama, Armando. “The Sketch Programmers Manual.” MIT Computer Science Artificial Intelligence Lab, people.csail.mit.edu/asolar/manual.pdf

[3] R. Sedgewick, F. Flajolet, An Introduction to the Analysis of Algorithms, Addison-Wesley, 2013

[4] Gulwani, Sumit. “Automating String Processing in Spreadsheets Using Input-Output Examples.” Contents: Using the Digital Library, ACM, 26 Jan. 2011, dl.acm.org/citation.cfm?id=1926423.

[5] Goldman, David. “Office 2013: Nice Feel, but Missing a Touch.” CNNMoney, Cable News Network, 16 July 2012, money.cnn.com/2012/07/16/technology/microsoft-office-2013/index.htm.

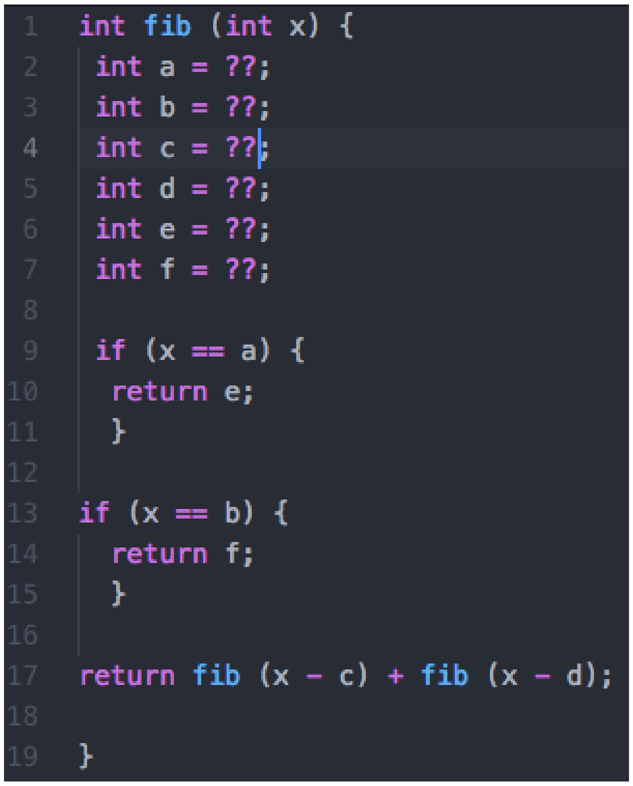
[6]Sagralof, Michael, and Kurt Mehlhorn. Computing Real Roots of Real Polynomials. 13 Oct. 2013, people.mpi-inf.mpg.de/ msagralo/RealRootComputation.pdf.

8. Acknowledgements

I would like to acknowledge Professor Aarti Gupta, Professor Armando Solar-Lezama, and everyone in my independent work seminar COS IW09.

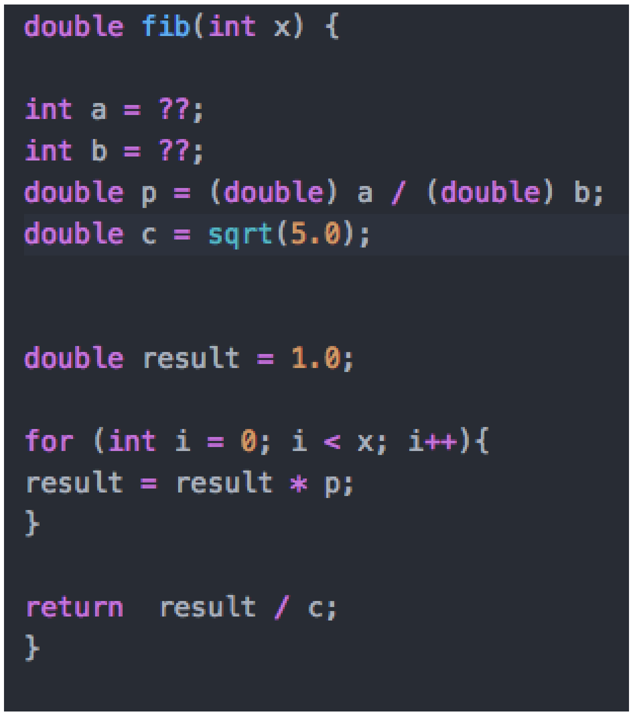
9. Appendix

The following is some of the code written that may be helpful to understanding the implementations described.



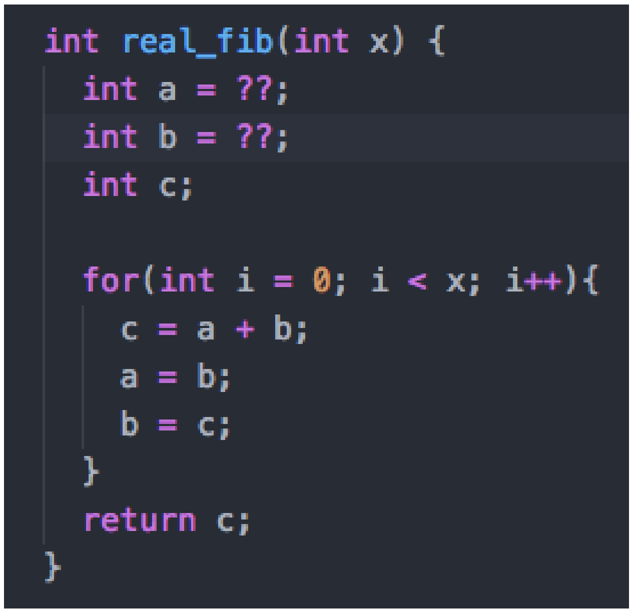
```
int fib (int x) {
 int a = ??;
 int b = ??;
 int c = ??;
 int d = ??;
 int e = ??;
 int f = ??;

 if (x == a) {
  return e;
  }

if (x == b) {
  return f;
  }

return fib (x - c) + fib (x - d);

}
```

Figure 5



```
double fib(int x) {

int a = ??;
int b = ??;
double p = (double) a / (double) b;
double c = sqrt(5.0);


double result = 1.0;

for (int i = 0; i < x; i++){
result = result * p;
}

return  result / c;
}
```

Figure 6



```
int real_fib(int x) {
  int a = ??;
  int b = ??;
  int c;

  for(int i = 0; i < x; i++){
    c = a + b;
    a = b;
    b = c;
  }
  return c;
}
```

Figure 7