# Verifying the Worst-Case Time Complexity of Hash Tables

Ashley Bruce
Computer Science Department
University of California, Santa Barbara
ashleybruce@ucsb.edu

## Abstract

Hash-tables are data structures that have constant time complexity for best and average-case input and linear time complexity for worst case input. This paper presents the verification of this worst-case time complexity by synthesizing input that always triggers worst-case addition. Due to the nature of how hash-tables work, this input is only achieved if the item being inserted collides with every item that is inserted before it. In order to achieve this for every value, this means that all items must hash to the same initial value. Thus, input is synthesized based off the initial hash function.

To go about synthesizing this input, an implementation of a hash-table is written in Racket and a solver is defined in Rosette to generate a model for worst-case input.

***Keywords:*** Hash tables, linear time complexity. Rosette, verification, synthesis

## 1 Introduction

The hash-table data structure is a fixed-size array that contains items of some type. Each item in the hash table contains some data that is used for searching; this is called the "key." If we consider the size of the hash table to be known as TableSize, each key is mapped to some number in the range of 0 to TableSize − 1. This mapping, also known as a "hash function," maps all the keys to indices within the Hash Table.

The hash function should ideally map each key so that all the keys are evenly distributed among the array and distinct keys are hashed to different indices. But, due to the array only having a finite number of slots, while the number of keys is immeasurable, distinct keys will occasionally be mapped to the same index, known as "collision" [1]. Theoretically, we could maintain an array that is large enough such that most the keys would be mapped to their own slot, but in practicality, this array would end up as mostly empty space. Out of all the possible keys, only a subset of keys is used in actuality [2]. Keys usually follow the same syntax rules as words in a language. Take, for example, the English language. Out of all the endless possible combinations for the 26 letters of the alphabet, only a small subset is found in the English language [2]. For this reason, arrays of a much smaller size are used, resulting in inevitable collisions.

There are a few ways to set up hash tables to resolve hash collisions. Separate chaining is one method that takes all the values hashed to the same index and maintains them in a linked list at that index. When a collision occurs, the value is just added to the end of the linked list. In this case, all values hashed to a specific index are maintained at the linked list pointed to at that index.

Another method is closed hashing, also known as open addressing. Unlike separate chaining, closed hashing only allows one value to be stored at that index in the array. If two values hash to the same index, only one value is placed at that index. The other value has to be put in a slot at a different index. Where this placement results depends on what method was used to resolve the collision.

If linear probing is used, then when a collision occurs, the next index is tried. Until an empty slot is found, the algorithm will continue looking through the array in a linear manner, starting from the index where the item was initially hashed to. If quadratic probing is used, then the items placement is determined by what collision resolution attempt it is on. The formula for this can be seen as follows:

$$\mathrm{H(x)} + i^2$$

where H(x) is the hash function and i is the probe number. If a collision happens, then quadratic probing checks the next index (i = 1; $1^2$ = 1). If that index is occupied, quadratic probing advances 4 indices (i = 2; $2^2$ = 4) and checks that one. This continues until an empty slot is found.

Even though there are multiple ways to deal with collisions when they happen, the ultimate goal is to minimize the number of collisions that do happen. A well-designed hash function and table size will minimize the probability of collisions occurring. The hash function depends on the table size, as it has to be known what values are available to map keys to. Ideally, the load factor (number of items in the hash table / number of total slots in the array) should not exceed 0.75. If we know the size of the data we are working


*Woodstock '18, June 03–05, 2018, Woodstock, NY*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

```java
public class Hash {

    public static void main(String[] args) {

        Integer [] itemsAndWorstCase = {0,0};
        // itemsPlaced index 0
        // worstCase index 1

        String[] hashTable = new String[11];
        Scanner userInput = new Scanner(System.in);

        for (int i = 0; i <= 7; i++) {
            System.out.print("Key: ");
            String key = userInput.next();
            collisionResolution(key, hashTable, itemsAndWorstCase);
        }}

    private static void collisionResolution(String key, String[] hashArray, Integer [] IAWC) {

        boolean notPlaced = true;
        int counter = 0;
        int hashedIndex = Math.abs(key.hashCode()) % (hashArray.length);

        int itemsInArray = IAWC[0];
        int worstCase = IAWC[1];


        while (notPlaced) {
            int currentIndex = Math.abs((hashedIndex + counter) % hashArray.length);
            if (hashArray[currentIndex] != null) {
                counter++;
            } else {
                hashArray[currentIndex] = key;
                IAWC[0] = ++itemsInArray;
                if (++counter == itemsInArray) {
                    IAWC[1] = ++worstCase;
                }
                notPlaced = false;
            }}}}
```

**Figure 1.** Java implementation of a Hash Table with user input.

with in advance, then choosing a table size should be relatively straight forward. Otherwise, we have to make our best guess and resize the hash table as necessary. Additionally, the size of the array should be a prime number. This becomes apparent when looking at the hash function.

The hash function transforms a key of some type into an integer that can be used to index an array. This number should be as unique as possible to reduce the number of collisions. The first step to doing so is to transform the key into an integer value, if not in one already. For strings, it is conventional to use their corresponding ASCII values. But it is not enough to just take these values and add / multiply them together. Take for example the following two words and their ASCII values:

STUDY – 83 84 85 68 89
DUSTY – 68 85 83 84 89

**Example 1.** ASCII values for two words with the same letters.

Although the two words are distinct and should be hashed to different values, because they are made up of the same letters, they will ultimately get hashed to the same value if the numbers are only added or multiplied together. A better function would be to somehow factor in the order to end up with a unique value. Consider the following equation.

$$a_0 + a_1X + a_2X^2 + ... + a_{n-1}X^{n-1}$$

**Equation 1.** Polynomial accumulation.

If X is given one of the following values – 33, 37, 39, 41 – the number of collisions in a vocabulary of the English language is at most 6 [2]. As can be seen, each value is multiplied to a multiple of X, which is determined by the placement of the value within the key.

## 2 Overview

Now that there is a better general understanding of how hash-tables work, the rest of the paper is laid out as follows.

First, the problem of worst-case insertion is introduced. An implementation of a hash-table is then provided in Java and the problem is expanded on with a motivating example. Following this, the corresponding implementation of a hash-table in Rosette is provided. This section is followed by a challenges section, which addresses the difficulties faced when going about this project. The synthesis procedure is then presented. Evaluation of the effectiveness of this solution is then reviewed, and the paper concludes by taking a look at work related to this project.

## 3 Problem Formulation

In an ideal data set, all the keys are hashed to a different and unique index and there is no collision or collision resolution necessary. Since the hash function takes constant (O(1)) time and an array lookup for a specific index takes constant time, then we know that in this case, entering the item into the

array takes constant time. This is the case because the item will be inserted in the first place it is hashed to. Search, insert, and delete will also take constant time for the same reasons. Thus, this give us our best-case scenario.

But what if we were given a data set such that all keys hashed to the same slot? A collision results from every call to the hash function. While the hash function is still done in constant time, finding an empty slot would take n tries, with n being the item number. This leads to a linear time complexity (O(n)) for insertion, search, and deletion.

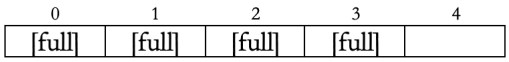
The only way for an item to collide with every item inserted before it every time is if each value was initially hashed to the same index. Given a simple hash-function and a known array size, calculating this by hand should not take too long. But what if the hash-function was more complex? Trying and calculating this by hand becomes much more non-trivial. Can input that triggered worst-case addition every time be synthesized?

## 4 Motivating Example

Consider the code in Figure 1, which depicts a partial hash-table implementation written in Java. For simplicity of this example, the built-in hashCode method was used, which calculates the hash code in a way similar to that shown in Example 1. Search and delete methods were omitted from this implementation, as the purpose of this project is to investigate the worst-case time complexity of hash tables, and the time complexity of search and deletion reflect that of insertion, thus only one of the mentioned methods needs to be verified.

As can be seen in the code, placement of a value depends on whether or not there is already a value present at that index. If there is already a value present, then the item attempting to be inserted must find another place to be inserted to, as the index is already occupied. This is known as a "collision" and "collision resolution" respectively. The time it takes to add a value to the hash table solely depends on how many collisions there are before the object is placed, which can be seen in the code starting on line 29. The counter on line 32 shows how many placement attempts there were before the object is placed and the collision is resolved.

While this counter instance only counts the number of collisions when placing a single value, it would be simple to implement a global variable along the same lines. This project is looking for input that triggers the worst-case complexity, so it is expected that every add takes O(n). In order to track this, we have to know how many items are in the array before the item is added plus one, for the item being added. While this informs us of the number of the item being added, it will also give us a goal of how many times the instruction needs to execute for it to take O(n) time. To better visualize this, consider the example array below that uses linear probing to resolve collisions.




**Example 2.** Hash table array with a space of 5, utilizing linear hashing to resolve collisions.

For simplicity of the example, collision resolution is done in a linear manner, but note that to make this code quadratic, the only difference would to be to make the following change on line 30 in Figure 1:

```
int currentIndex = (hashedIndex + (counter * counter)) % hashArray.length;
```

If another item, which we will call item, were to be added this would become the fifth item in the table. This could also be represented as the array size before the addition, 4, plus 1. In order to have worst case complexity, we expect the collision resolution line of our code, lines 29 through 35 in Figure 1, to execute 5 times. It executes 4 times when collisions occur, one for each item already in the table, plus one more to resolve the collision itself. Let's say item initially hashes to 0. The statement on line 31 executes as true and the counter increments. The next index, 1, is checked, but because that slot is full, the statement again executes as true and the loop continues. This happens again with indices 2 and 3. On the fifth time through the loop, the item will be placed in the fourth index.

To generalize this, the item that is being added can thought of as the items currently in the array plus 1. This gives us the total number of items in the array, which we will call itemsInArray. In order to make sure that we have achieved worst case time complexity, the lines in code 29 through 35 should execute itemsInArray times. This is because the number of times the code should execute should be the number of collisions plus the final resolution. If this is the case, then a global variable, worstCase is incremented. This variable keeps track of how many items had linear time for insertion. If, at the end of inserting each item, worstCase is equal to itemsInArray, then the hash-table has worst-case time complexity for addition.

## 5 Implementation

A similar implementation of the hash-table presented in Figure 1 was created in Racket. Generation of worst-case input was done in Rosette. Rosette is a solver-aided language that extends Racket [3]. Implementation was done in Racket so that Rosette verification tools could be utilized. The code for this implementation is shown in Figure 2.

The code begins by declaring a few global variables, most of which are used in the collisionResolution function. Initial arraySize is declared here to be 5, but this is an arbitrary value and changing this variable will change the array size everywhere. In the evaluation section, it differing vector sizes are used so that more items can be verified and added.

```
; Declarations
(define itemsInArray 0)
(define worstCase 0)
(define counter 0)
(define arraySize 5)
(define sameVal 0)
(define v (make-vector arraySize))
(vector-fill! v -1)

; Inserts a key to a given index
(define (insert index key)
  (vector-set! v index key))

; Method that increases by one
(define (addOne num)
  (+ num 1))

; Simple hash function method
(define (hashFunction key)
  (abs (remainder key arraySize))
  )

; Determines whether the worst case counter
; needs to be incremented
(define (isWorstCase)
  (define newCounter (addOne counter))
  (cond
    [(= newCounter itemsInArray)
     (set! worstCase (addOne worstCase))]
  ))

; Hash function
(define (hash key)
  (collisionResolution (hashFunction key) key)
  )

; Resolves collisions within table
(define (collisionResolution index key)
  (define num (vector-ref v index))

  (cond
       [(= num key)
        (set! sameVal (addOne sameVal))
        (set! counter 0)]
       [(= -1 num)
         (insert index key)
         (set! itemsInArray (addOne itemsInArray))
         (isWorstCase)
         (set! counter 0)]
       [else
         (set! counter (addOne counter))

         (collisionResolution (remainder (addOne index) arraySize) key)

         ])
  )

; Method for adding three numbers into
; the hash table in one call
(define (addNums a b c)

  (hash a)
  (hash b)
  (hash c)

  (display "worstCase: ")
  (display worstCase)
  (display " itemsInArray: ")
  (display itemsInArray)
  )
```

**Figure 2.** Racket Implementation of a Hash Table

A vector v is then made, and each index is initialized with -1 to represent that the index is empty.

The next four methods are all used within the hash and collisionResolution functions.

The hash function takes a key as an argument and passes the index and key to collisionResolution. To get the index from the key, hashFunction is called on the key. This is a simplistic function that performs a modulus function on the key, returning an index within the size of v.

The collisionResolution function is in charge of doing all the checks on the key before the item is inserted into the vector. If the key trying to be added into the vector already exists inside the vector, then sameVal is incremented and the item is not inserted. The counter is also reset to 0 due to the fact that the duplicate item may not be the first index the item is hashed to. Otherwise, the function then checks to see if the index is empty. If the number at the index is -1, then there is no item at that index and the key is inserted at that index. Otherwise, if the index is occupied, the counter is incremented and collisionResolution is recursively called with the next index. If an item is inserted, the global variable itemsInArray is incremented, and isWorstCase checks to see if the item being inserted had worst-case addition time. It does this by comparing the counter to the itemsInArray. If the item being inserted did have worst-case insertion time, then the worstCase global variable is incremented.

The code depicted then finishes with a method for hashing three numbers and displaying the worstCase and itemsInArray variables. Three, again, is an arbitrary value for the sake of including the code here. In the evaluation, addNums is changed to reflect the number of items that will be inserted into the hash-table. This method will be used to confirm that items hashed into the vector have worst-case insertion.

## 6 Challenges

Initially, the idea for verifying this algorithm was based on the concept that the worstCase counter would equal the itemsInArray counter during input that triggered worst-case time complexity. Trying to go about synthesizing worst-case input this way ended up presenting a few challenges.

The first challenge was that due to the recursive nature of the hash table implementation in Figure 2. It was found that when adding symbolic variables to the collisionResolution method, it failed to terminate the recursive iterations. A potential fix to this was to make it so Rosette only considered executions of a certain length [4]. In this implementation of the hash-table, a for-loop was declared, and the number of times the code executed was based on the current value of itemsInArray. At the worst case, the item will collide with everything already in the hash table (itemsInArray) and thus will have to call collisionResolution the same number of times as itemsInArray. This successfully terminated the loop in collisionResolution.

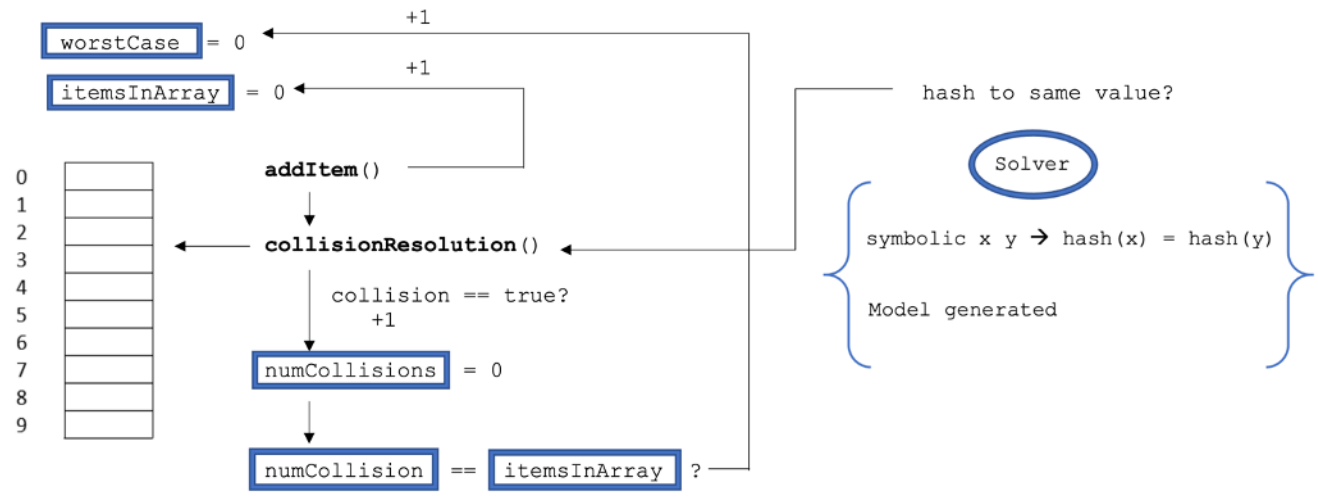



**Figure 3.** Hash table implementation and solver overview.

But, although the loop successfully terminated, solving for when worstCase equaled itemsInArray was still unsatisfiable, which led to the next challenge.

Only a subset of Racket's language is lifted to Rosette [5]. The term "lifted" refers to parts of the Racket language that can be used with symbolic values [6]. When a form is lifted, it means that it has the same meaning in Rosette programs as they do in Racket programs. When writing programs in Rosette, using this subset is known as using the safe dialect of the full Racket language. Trying to use unlifted constructs of the Racket language results in unsafe operations being performed [7]. Because the symbolic virtual machine (SVM) only controls the execution of lifted procedures, unlifted code falls to the Racket interpreter, and thus is subject to side-effects caused by this interpreter in the presence of symbolic values [7].

This is what appeared to happen with the implementation of the hash table that was used for this project. When attempting to insert symbolic values into the collisionResolution function, a few lines of this code were escaping to the Racket interpreter and executing incorrectly. Most notably, this included code that included comparison with the symbolic value, such as when the insert method tried to compare the symbolic key to the key already present in the hash table to make sure a duplicate wasn't being added:

[( = num key )]

And when checking to see if the index in the array was unoccupied:

[( = -1 num )]

This shed more light into why the recursive implementation did not work initially, as the solver was unable to properly make sense of the comparisons in the code before that, and thus kept entering the last else, and kept calling itself.

Unfortunately, given the timeline of this class, I was unable to figure out how to properly lift this subset of code or, alternatively, how to ensure that the unlifted subset continued to behave correctly in the presence of the Racket interpreter. Thus, instead of verifying that worstCase = itemsInArray, the hash function method was verified instead.

## 7 Synthesis

As seen from the motivating example, in order to achieve linear time complexity, the item being inserted has to collide with every item inserted before it. Essentially, this means that all items must hash to the same initial value.

Knowing this, generating input that would trigger worst-case time complexity is pretty straightforward. All that is needed to do is make sure the items given to hash table hash to the same initial index.

The implementation for this is shown in Figure 4. In this code, symbolic integers are first declared, and a new solver instance is generated, named hashSame [8]. Symbolic variables are inserted into the hashFunction method and an assertion is made that the result of each hashFunction call equals the other. A model is then generated. This model can be checked by calling addNums with the values generated from the solver.

An alternative way of writing this same solver is depicted at the bottom of Figure 4. In this implementation, an assertion is explicitly made that the hashFunction values of symbolic integers x, y, and z are equal. The function sol is then defined to solve this assertion. To evaluate the values

of each symbolic variable, evaluate has to be called, as seen below the definition for sol.

## 8 Evaluation

Evaluation of the success of the synthesis method was done in two different steps. The first was checking to make sure the values generated in the model were correct and hashed to the same index. This was confirmed by calling the hashFunction method on each item generated. If each item returned the same number as a result, then then we know that each item will initially hash to the same value.

This is further confirmed by entering the values generated from the model into the addNums method. Due to the implementation of the collisionResolution method in the hash table, the number of worst-case additions is stored globally. The addNums method will take values, insert them into the hash table, and then display the number of worst-case items, as well as the items in the vector. If these are equal, then we had worst-case addition for each item inserted into the hash-table.

Initially, two symbolic values were declared and used in the hashSame function. The two values generated were then inserted into the addNums function to confirm worst-case insertion. The results generated from hashSame can be seen in Figure 5. The hashSame and addNums methods were modified according to how many symbolic variables were declared and entered. The array was also resized from 5 to 10 when the number of symbolic variables being entered got to 6.

As can be seen from the table in Figure 5, the number of items generated by the model is the same as how many symbolic variables were inserted into the function. Because all values generated were being placed into the table, this number also reflects the global variable itemsInArray. The values of the items initially generated were recorded in the middle column and tested in addNums. The worstCase variable value was recorded in the last column after calling addNums.

It is worth noting that any more symbolic numbers added after this point begun to generate duplicate numbers. Duplicate numbers are not added into the hash table, due to one of the checks before inserting. When running addNums, the worstCase number, as well as itemsInArray, only reflects the number of items that were inserted into the array. Thus, with duplicates generated in the model, the numbers for worstCase and itemsInArrray will be less than the number of values generated from the model.

One solution to this would be to make an assertion before running the model that none of the symbolic variables can equal one another. When trying to implement this solution, it was found that a statement such as (not (= a b c d e ...)) did not work. Trying to synthesize input still generated duplicate numbers. It was found that this was because this statement did not properly consider the comparisons between every variable entered into the statement. This statement was instead interpreted like so: (& (! ( = a b) ( = b c) (= c d)...)) and so on. Not all interactions between the variables were considered.

In order to properly consider all interactions, each interaction had to be explicitly stated in its own assertion, which is shown in Figure 6. This figure shows the amount of assertions necessary for comparing 8 symbolic variables. As can be seen from this code, the number of assertions is equal to nCr, with n being the number of symbolic variables being entered and r being the number of items being compared, or 2.

It is also worth noting that a simpler solution to this problem was not able to be found. If there is a better solution to this problem, it was unknown to the author of this paper. Due to the exponential increase in the number of assertions required to compare more symbolic values, data was not collected for more than 8 symbolic variables.

Using this solver to generate input for 8 symbolic variables, the size of the vector was further changed, to ensure input could still be reliably generated for varying vector sizes. By doing this, it was found that the SVM had trouble generating models for certain table sizes. For example, trying to solve for a model for 8 symbolic integers in a vector-size of 49 generated (unsat). This issue resolved if the current-bandwidth was increased from 8 to 16.

Although it wasn't tested explicitly in this project, modifying the hashFunction such that the number being inserted

```
(current-bitwidth 8)
(define-symbolic* x y z integer?)
(define hashSame (solve+))

; Synthesis of worst-case input
(hashSame
   (= (hashFunction x) (hashFunction y) (hashFunction z)))

; Another way to synthesize
(define sol (solve (assert (= (hashFunction x) (hashFunction y) (hashFunction z)))))
(evaluate x sol)
(evaluate y sol)
(evaluate z sol)
```

**Figure 4.** Code that synthesizes input for worst-case addition of an item into a hash-table.

| Items | Input generated | Worst Case |
|---|---|---|
| 2 | 50, 0 | 2 |
| 3 | 108, -128, 113 | 3 |
| 4 | -74, 79, -9, 86 | 4 |
| 5 | -15, -70, -5, 70, 65 | 5 |
| 6 | -50, -80, -70, 90, 70, 0 | 6 |
| 7 | -70, -110, -50, 0, 70, 100, -20 | 7 |

**Figure 5.** Results generated from hashSame and addNums. First column represents how many symbolic variables were entered into the hashSame method. The second column lists the values synthesized from this function. The third column gives the result displayed on screen after calling addNums with the numbers generated. Array size was changed from 5 to 10 when 6 symbolic variables were going to be added.

went through a more arithmetic heavy operation should still generate a model when plugged into the solver method.

Overall, the solution presented in this paper appears to be an adequate solution for generating input that triggers worst-case addition time into a hash table.

## 9 Limitations

As discussed in the previous section, the main limitation of the code presented in Figure 4 is that more duplicates are generated with more symbolic variables being used to generate a model. While the hash-table implementation in Figure 2 prevents against the insertion of these duplicate values, increasingly more symbolic variables would be required to generate unique input. This problem was solved by asserting that each symbolic variable differed from one another. This solution had its own limitation though, as the more symbolic variables introduced, the more assertions were necessary to ensure all of the values would be difficult, as shown in Figure 7.

The number of assertions necessary to encompass all interactions between each variable can be represented by the equation nCr, where n is the number of symbolic variables being solved for, and r being 2, for the number of symbolic variables being compared. As n grows linearly, with r staying the same, the number of assertions necessary (nCr) grows exponentially, as shown in Figure 6.

Another limitation to this project is due to the hash-table implementation being unlifted. Because of this, symbolic variables could not be directly inserted into the hash-table. Worst-case input had to be generated from symbolic variables input into the hashFunction method. The numbers generated from this model were then placed into the hash-table. Being unable to insert these symbolic variables directly into the table made the methods available for verification limited to the hashFunction. If this project were carried out over a longer period of time, it would be worth considering to create an implementation of a hash-table that was properly lifted.

## 10 Related work

Research found to be related to the verification of worst-case insertion in hash tables is discussed in this next section. Although no prior research was found to be done on this exact topic, work that is closely related will be explored.

*Lifted Implementation of a Hash Table:* This Github Repository depicts a lifted implementation of a hash-table written in Rosette for keys that are strings. This lifted hash table was written in part of a larger project that checks the equivalence of two contracts regarding self-defined transactions [9]. As this project is still under development, information regarding the hash-table's role in the project is minimal.

*Pattern Fuzzing for Worst Case Complexity:* This research presents a testing technique that looks into determining worst-case asymptomatic complexity for a given application [10]. Instead of looking for concrete input, this paper focuses on identifying input patterns that maximize resource usage

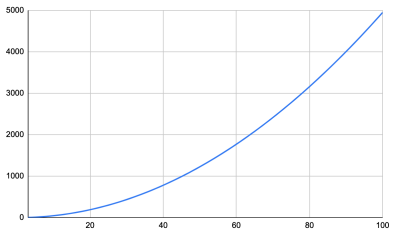



**Figure 6.** Exponential growth of nCr, with r being held constant at 2. For 100 symbolic variables (the x axis), 4950 assertions would be needed.

```
(assert (= (hashFunction x) (hashFunction y)
           (hashFunction z) (hashFunction a)
           (hashFunction b) (hashFunction c)
           (hashFunction d) (hashFunction e)))
(assert (not (= x y)))
(assert (not (= x z)))
(assert (not (= x a)))
(assert (not (= x b)))
(assert (not (= x c)))
(assert (not (= x d)))
(assert (not (= x e)))
(assert (not (= y z)))
(assert (not (= y a)))
(assert (not (= y b)))
(assert (not (= y c)))
(assert (not (= y d)))
(assert (not (= y e)))
(assert (not (= z a)))
(assert (not (= z b)))
(assert (not (= z c)))
(assert (not (= z d)))
(assert (not (= z e)))
(assert (not (= a b)))
(assert (not (= a c)))
(assert (not (= a d)))
(assert (not (= a e)))
(assert (not (= b c)))
(assert (not (= b d)))
(assert (not (= b e)))
(assert (not (= c d)))
(assert (not (= c e)))
(assert (not (= d e)))
```

```
Welcome to DrRacket, version 7.8 [3m].
Language: rosette, with debugging; memory limit: 256 MB.
> (solve asserts)
(model
 [x$0 -8]
 [y$0 88]
 [z$0 -48]
 [a$0 68]
 [b$0 28]
 [c$0 -128]
 [d$0 -108]
 [e$0 -28])
```

**Figure 7.** Assertions necessary to compare 8 symbolic variables and make sure none are equal to one another. Calling "solve asserts" generates the input shown on bottom. As can be seen, none of the values generated are equal to one another.

of the target program. In doing so, a new model of computation, called Recurrent Computation Graph (RCG), is used to express these input patterns.

This paper mentions that there is a large body of literature out there that just looks at the worst-case complexity analysis without looking into producing worst performance inputs (WPIs). To efficiently generate input that triggers worst-case performance of a given program, a new black-box complexity testing technique is proposed. This is based off the idea that WPIs almost always follow a specific pattern that can be expressed as a simple program [10]. The goal of this paper is to find a program that expresses the common pattern shared by these WPIs [10].

*Verifying OpenJDK's Sort Method for Generic Collections:* In the Java standard library, and many other frameworks, TimSort is the sorting algorithm provided. This paper looks at a case study of this sort function in which a crash-causing bug occurs due to an uncaught exception. This bug was found when attempting to verify TimSort via mechanical proofs [11].

As can been in this small subset of research related to the ideas presented in this paper, the verification of well-known algorithms is of great use. Even with well known and widespread algorithms, such as TimSort, previously unknown bugs can be found with verification. Verification of certain aspects of these well-known algorithms, such as worst-case time complexity, can also shed insight into bugs related to performance.

## 11 Conclusion

Knowing a program's worst-case complexity is important, as it can shed insight into bug detection and the identification of security vulnerabilities [10].

Hash-tables are an interesting data-structure in which operations take constant time in the best case and average case scenarios and linear time in the worst case. It is very rare for input to trigger worst case time complexity, given the way hash tables are structured and used.

This paper took a deeper look into hash-table insertion in order to generate input that triggered worst-case insertion time. Rosette, solver-aided language extending Racket, was used so that solver-aided tools could be utilized. A partial implementation of a hash-table was created in Racket language. As only insertion was being investigated in this project, only methods that were a part of the insertion process were included in the implementation. Search and delete methods were omitted.

To generate worst-case input, a solver was created and symbolic input was inserted into the hashFunction. Input generated thus hashed to the same initial value, ensuring that worst-case insertion was achieved.

## 12 Acknowledgements

The project was carried out under the instruction of Professor Yu Feng for the Fall 2020 CS292C class at the University of California, Santa Barbara.

## 13 Works Cited

1. Weiss, Mark Allen. (2012). *Data Structures and Algorithms Analysis in Java. 3rd Edition.* Pearson.
2. Agrawal, Divy. (2020). *Data Structures and Algorithms: Hashing.* Personal Collection of Divy Agrawal, UC Santa Barbara, Santa Barbara, CA.

3. Torlak, E., & Bodik, R. (2013). *A lightweight symbolic virtual machine for solver-aided host languages.* UC Berkeley.
4. Torlak, Emina. Lifted Forms, Rosette Github Repository. 2018.
5. *Racket Documentation.* V7.9. Lifted Racket Forms.
6. *Racket Documentation.* V7.9. Syntactic Forms.
7. *Racket Documentation.* V7.9. Unsafe Operations.
8. *Racket Documentation.* V7.9. Solver-Aided Forms.
9. Chen, Yanju. Venti - Verifying Smart Contracts via Yule. Github Repository. 2020.
10. Wei, J., Chen, J., Feng, Y., Ferles, K., Dillig, I. (2018). *Singularity: Pattern Fuzzing for Worst Case Complexity.* University of Texas at Austin.
11. Gouw, S„ Boer, F., Bubel, R., Hahnle, R., Rot, J., Steinhofel, D. (2019). *Verifying OpenJDK's Sort Method for Generic Collections.* Journal of Automated Reasoning.