HASH TABLE

BY AKARSH KUMAR

HASH TABLE PURPOSE

- The purpose of a HashTable is to serve as an ArrayList type object that has O(1) set, get, and lookup times.
- It also maps keys to values and includes put and get methods to control them.
- It achieves O(1) time by using a hashing function to store information. Going back, it knows exactly where to find the information based on the hash function

MY IMPLEMENTATION

- My implementation used an array of objects to hold all the information
- The hash function hashes the key into an integer and based off that it decides where in the array to store the information
- My implementation was programmed to double the size of the array whenever the load factor (percentage of the array full) exceeded 70%

HASHING FUNCTION

- The hashing function for my HashTable was very simple
- It used Java's built in hashCode() function on the key object
- It then modded the given integer with the size of the array to choose where to place the information in the array
- Code:

```
public int hashFunc(K key) {
        return Math.abs(key.hashCode()) % table.length;
}
```

HANDLING COLLISIONS

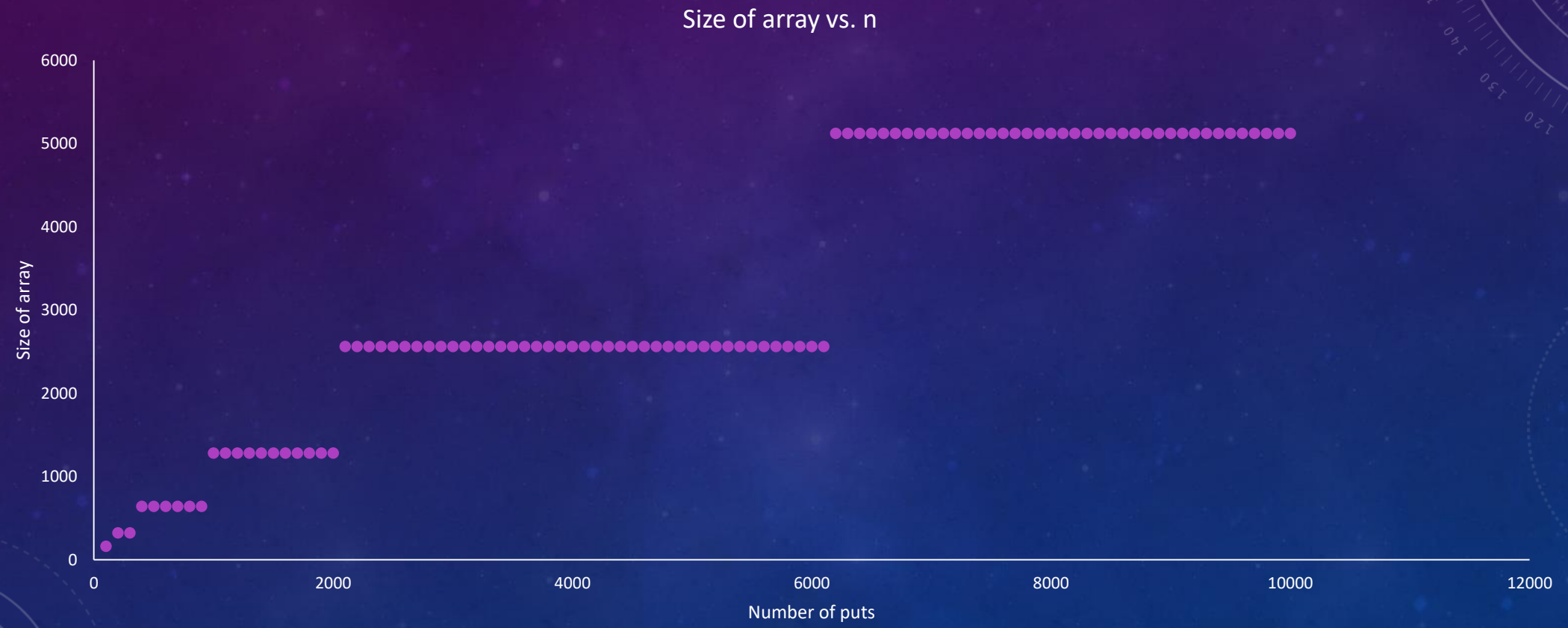
- Using that hashing function, there will be many objects that are mapped to the same stop.
- To handle this, instead of moving the object to the next available slot, the array at that index was a LinkedList of entries
- Multiple objects in the same index were just stored in a List of entries
- Since this list will be very short because of the distribution of information throughout the array, it will be fast to scan through the array when looking for a given key.

PERFORMANCE: TIME VS. N PUTS

TIME VS N. PUTS ANALYSIS

- Since the time to put one item is O(1), the time to put n items is O(n)
- This means the time vs. n puts graph should be linear, which it is
- The linear regression run on the graph gave a slope of 0.0014 ms/put
- This means that every put took an average of about 0.0014 ms.
- The y-intercept of the regression is the initial time that it took to get started (setup the table)
- The random bumps on the graph can be explained by the table expanding when getting too full

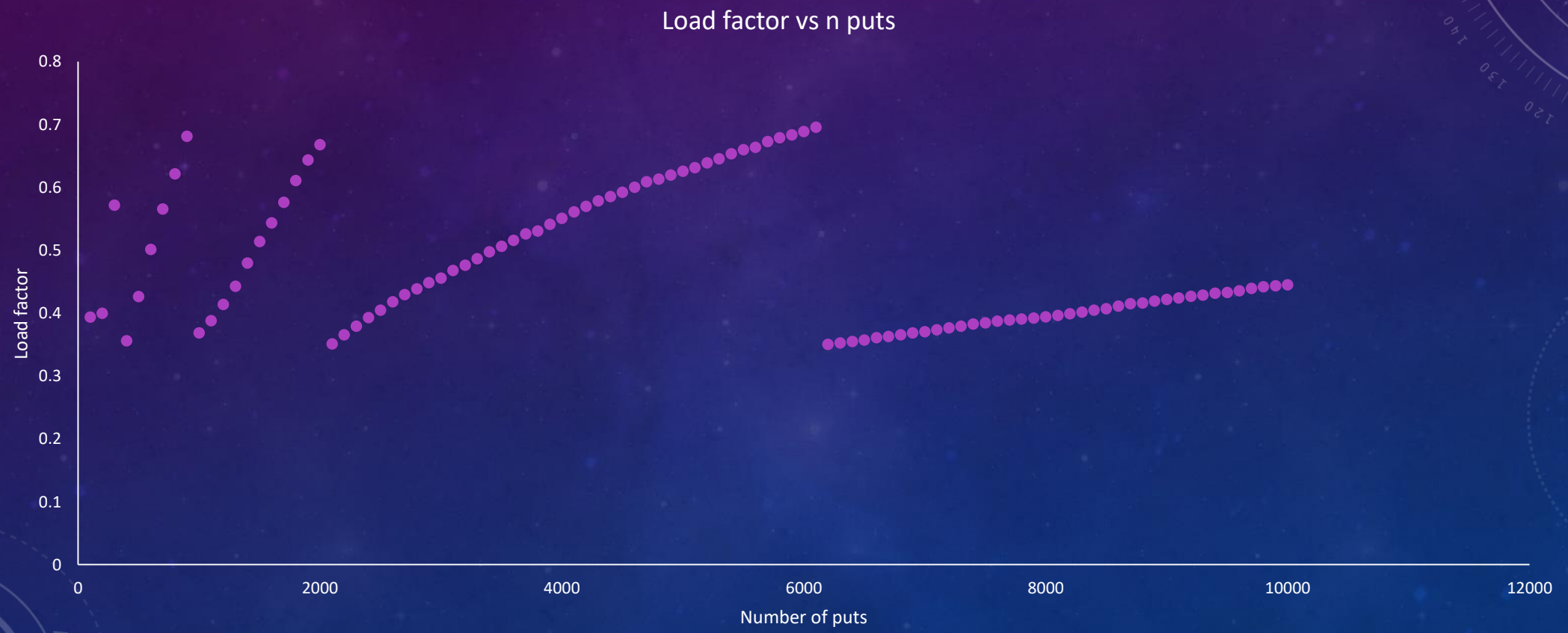
SIZE OF ARRAY VS. N PUTS



SIZE OF ARRAY VS N. PUTS ANALYSIS

- It is not possible to correlate the number of puts and the size of the table with the load factor because of the fact that words in the book may repeat (a put may simply increment the value of a key instead of making another entry)
- However, from the graph we can notice some key things
- The size of the array should double whenever the load factor exceeds 70%. This is shown in the graph by the jumps to double the value
- The lines are getting longer and longer because it requires more and more puts to get to 70% the size of the array as the size of the array grows

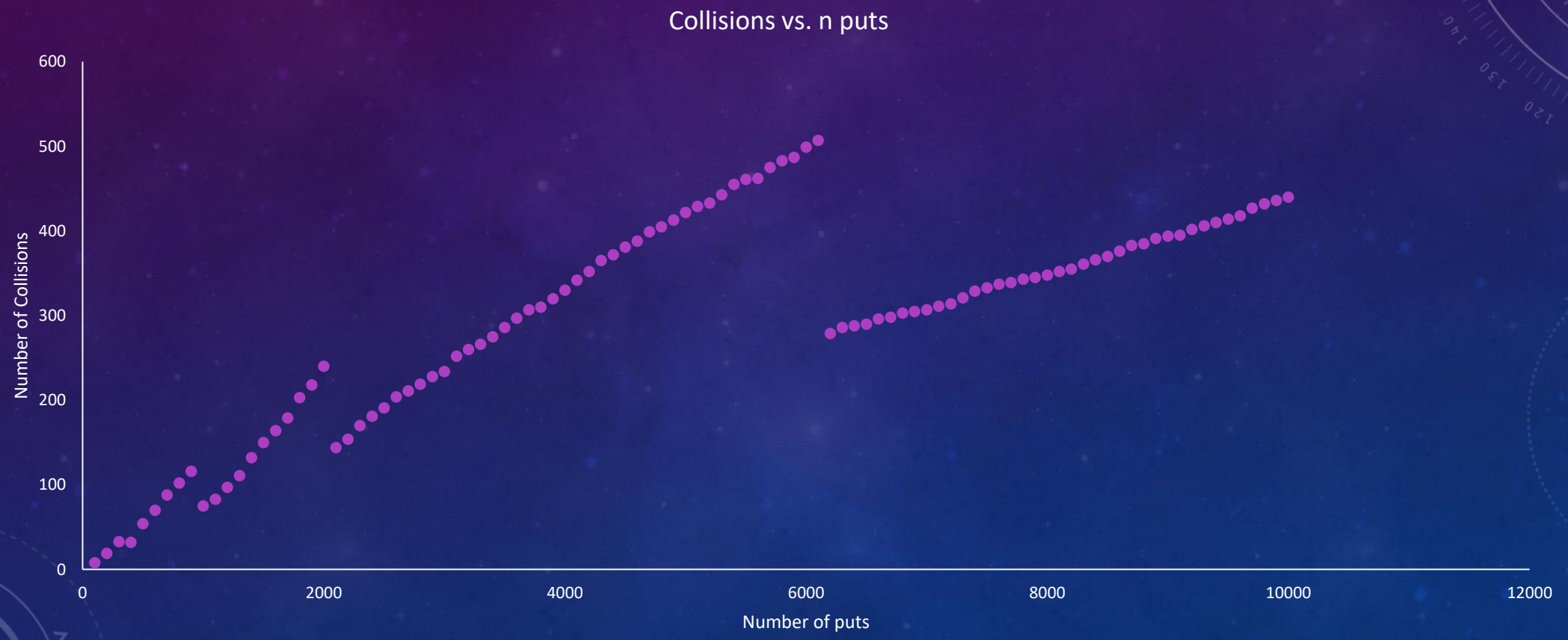
LOAD FACTOR VS. N PUTS



LOAD FACTOR VS. N PUTS ANALYSIS

- Since the load factor should not exceed 70%, the graph does not ever cross 0.7
- The graph shows that the line jumps down to 0.35 every time it passes 0.7
- This makes sense because when doubling the size of the array, the load factor (size/length of array) will be cut in half
- The lines are getting longer and longer because it takes more and more puts to get to 70% load as the size of the array grows

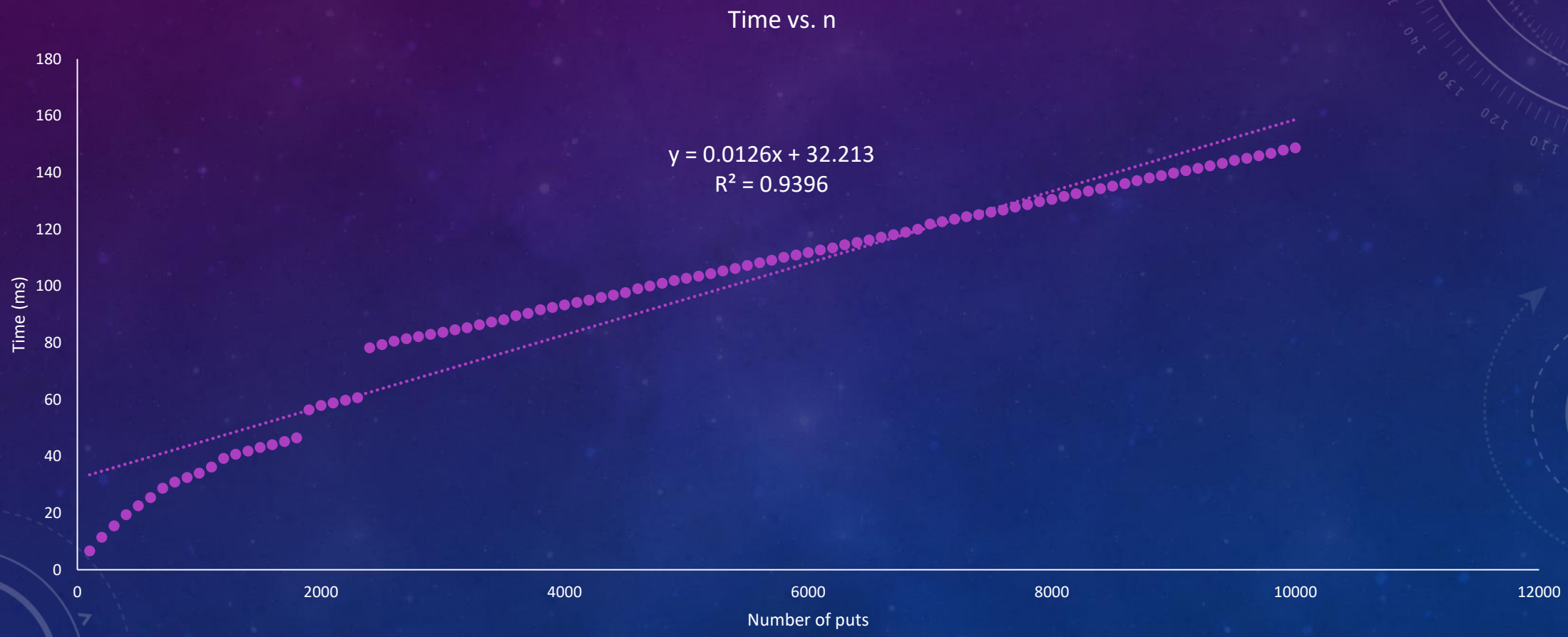
COLLISIONS VS. N PUTS



COLLISIONS VS. N PUTS ANALYSIS

- The number of collisions is measured as being the number of extra items stored in one index of the array
- The graph is a discontinuous graph because it also jumps down whenever the size of the array doubles
- The number of collisions should decrease as the array doubles because there is more room to fit all the data resulting in less overlap

JAVA REFERENCE (HASHMAP) TIME VS. N PUTS



REFERENCE TIME VS. N PUTS ANALYSIS

- The graph appears to linear as expected, besides some jumps in the graph
- These jumps are probably when the HashMap is resized to hold more entries
- Somehow, my implementation is much faster than the reference HashMap
- I cannot pinpoint the reason for this but it is most likely because mines is optimized for this assignment and inputting words, while the HashMap is probably optimized for some other feature

FINDING THE TOP 100 WORDS

- To find the 100 most common words with my implementation, the HashTable was first filled with all the words
- A loop ran through the entire HashTable, entry by entry, and found the most common word
- That was repeated 99 more times finding the most common word in each pass
- The list of words were saved in “top 100 words.txt”