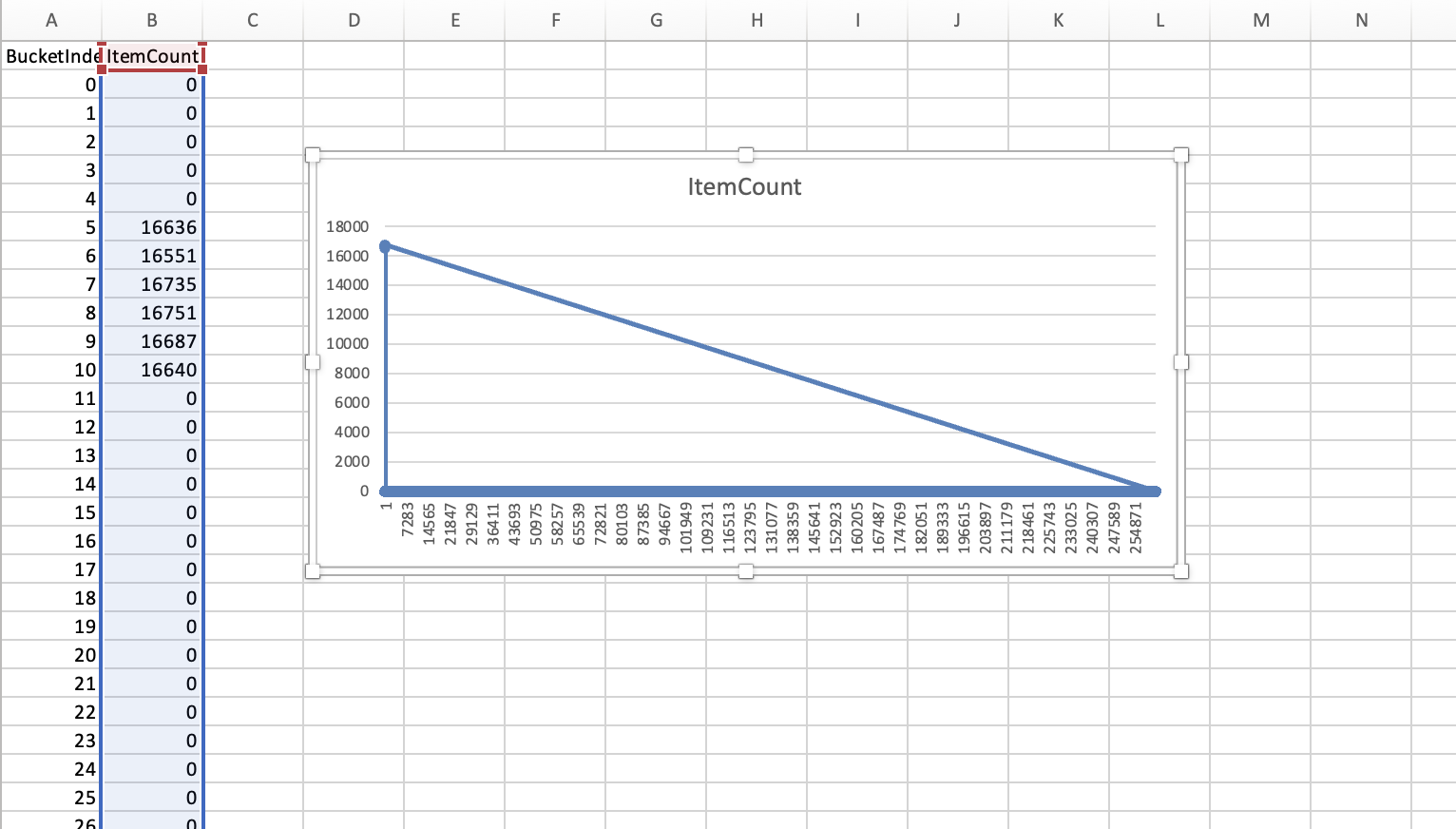
### **Graph Analysis**

#### **1. Bucket Distribution**



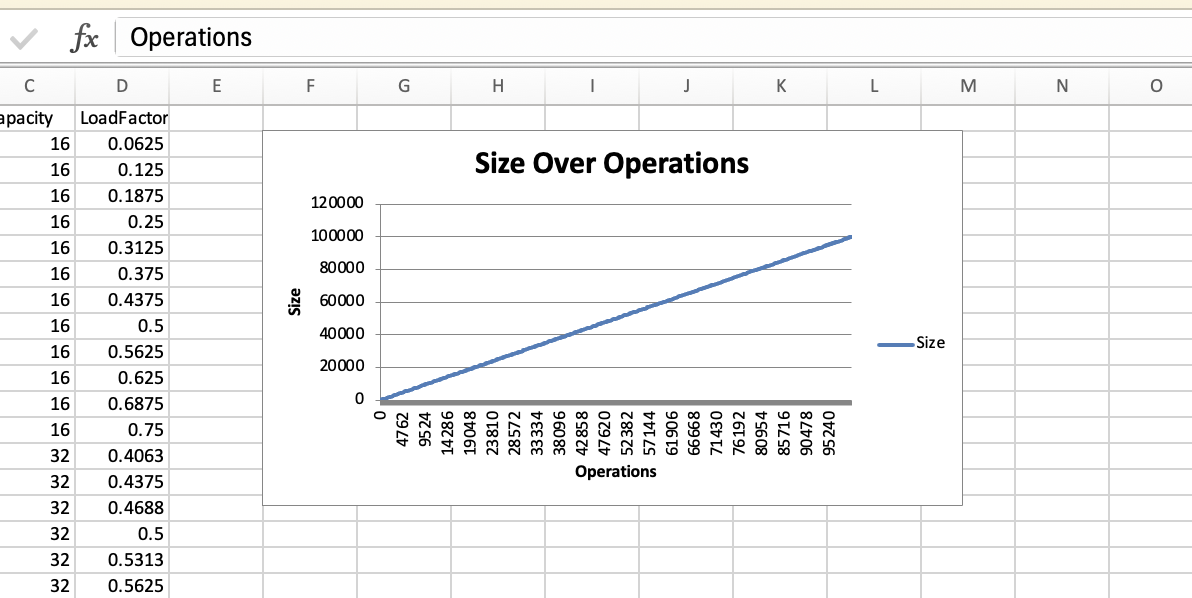
**Insight:**  
 This bar chart shows how items are distributed across different bucket indices in a hash map.  
 Observation:

* Most bucket indices have 0 items, suggesting underutilization or high sparsity in the hash table.
* If this hash map is part of a custom implementation, it might suffer from poor hash function design, leading to inefficient space usage and clustering elsewhere.

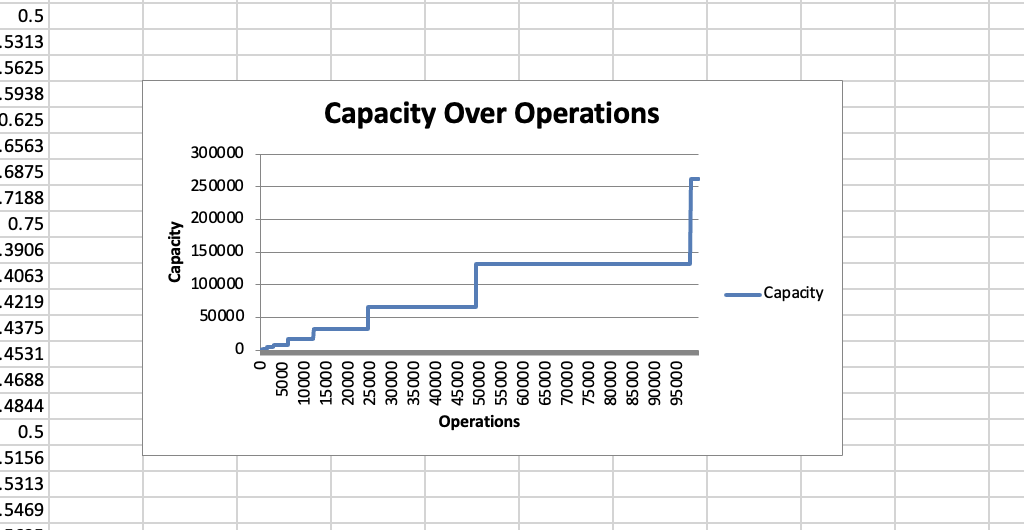
#### **2. Resizing Behavior**

**Insights:**  
 This workbook includes three line charts tracking hash table behavior during insertions:

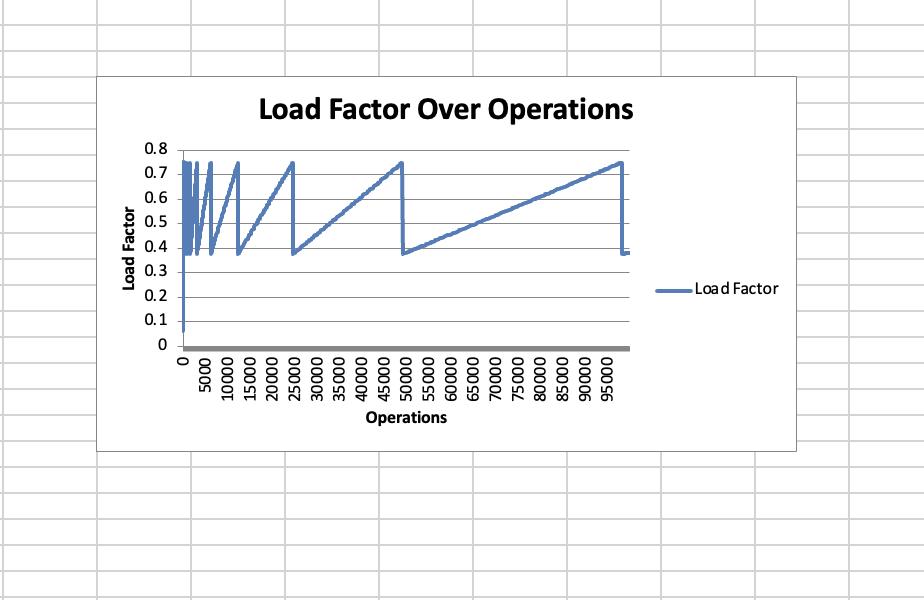
* Size vs Operations

  
 → The number of items increases steadily with each operation, as expected.

* **Capacity vs Operations**

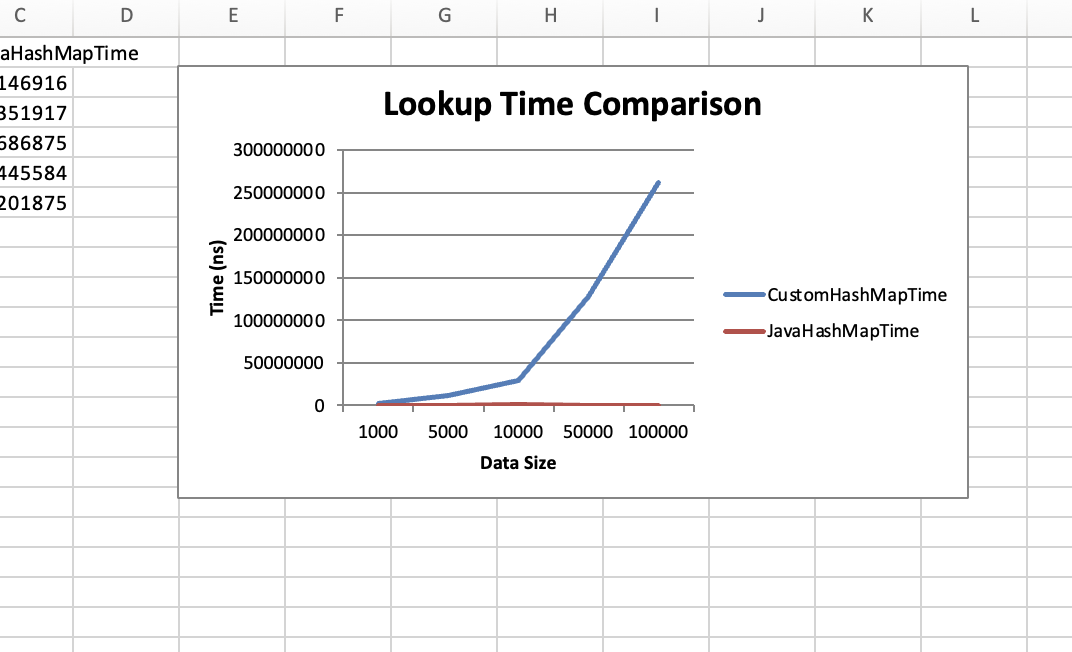
  
 → Capacity grows in discrete jumps, likely following a doubling pattern — a common resizing strategy in hash maps.

* **Load Factor vs Operations**

  
 → The load factor increases until a threshold, then drops after each resize, confirming dynamic resizing based on load.

**Conclusion:**  
 This behavior is consistent with a resizable array-backed hash table, where maintaining load factor below a threshold ensures time efficiency for lookup/insertion.

#### **3. Lookup Performance**

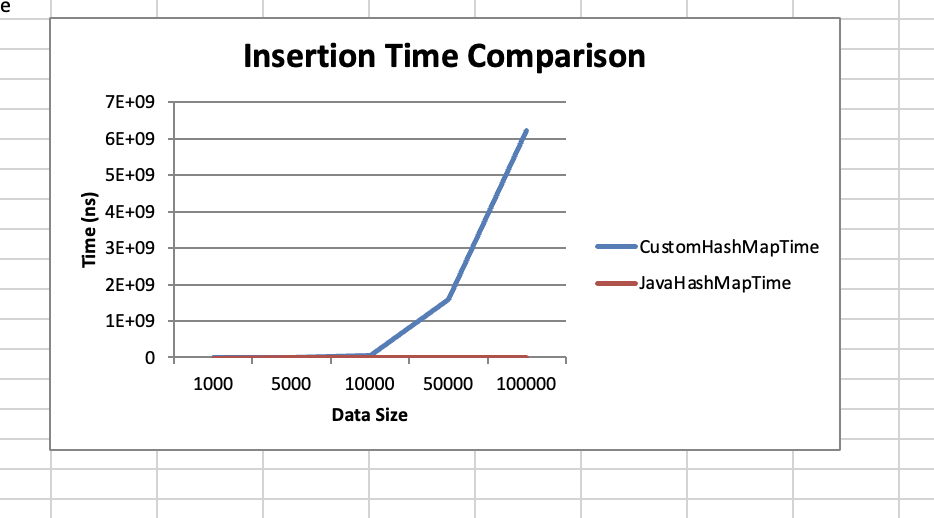


**Insights:**  
 This chart compares lookup times for a CustomHashMap vs Java’s built-in HashMap:

* Java’s HashMap remains fast and consistent, even as data size scales.
* CustomHashMap's lookup time increases significantly with size, showing inefficiency and lack of optimization.

**Conclusion:**  
 Java's native implementation clearly outperforms the custom version, emphasizing the importance of well-tuned hashing and load handling.

#### **4. Insertion Performance**



**Insights:**  
 This graph compares insertion times for both implementations:

* JavaHashMapTime scales modestly with data size.
* CustomHashMapTime shows exponential growth, especially past 10,000 items, indicating inefficient resizing or rehashing logic.

**Conclusion:**  
 The custom hash map suffers from serious performance bottlenecks during insertions, likely due to resizing cost, poor memory management, or suboptimal collision resolution.

### 

### 

### **Overall Summary**

* Java’s HashMap consistently outperforms the custom implementation in lookup and insertion efficiency.
* The custom hash map lacks balance in bucket utilization and struggles to scale.
* Resizing logic works, but performance degradation signals the need for algorithmic improvements.
* This analysis validates the importance of using load factor thresholds, efficient hashing, and proper data structure design when building scalable hash-based data systems.