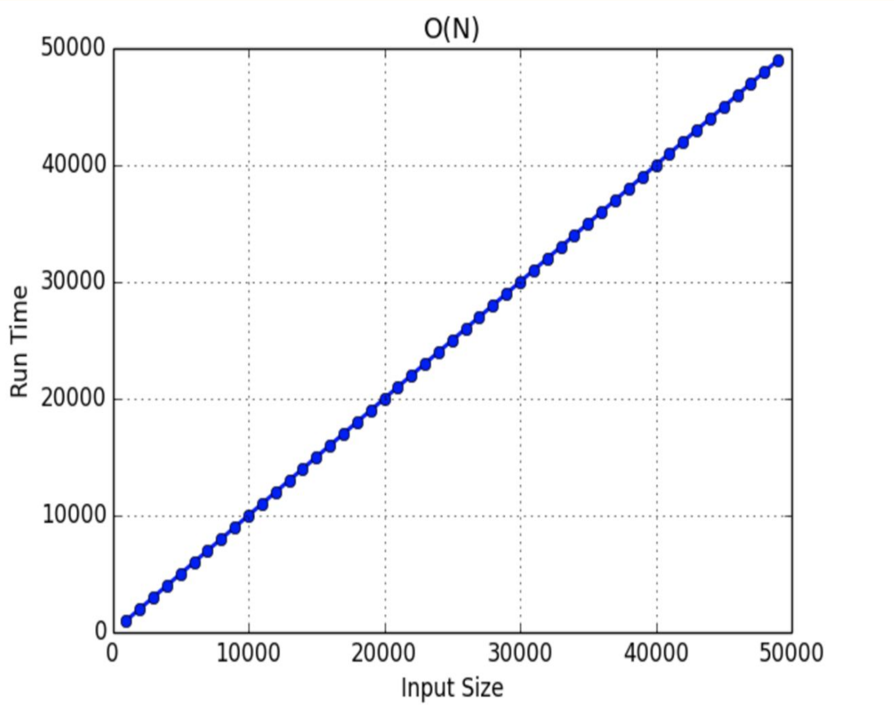
Hash Tables and Hashing Algorithms

By Fotis Antonatos, Joseph Lieberman, and Brad Estus

Algorithm Purpose

- Hashing algorithms allow us to create unsorted arrays which support insert, search, and deletion in *close to* O(1) time.

- Unfortunately, some implementations can significantly increase runtime, with search, delete, and insert taking O(n) time.



The Algorithm

- By comparing different collision resolution strategies and different hash function types, we aim to determine the best Hash Table configuration for use in a 75-100% full hash table.
- We are primarily concerned with runtime and collision count

- How does changing h(key,i) change our collision rate?

```
Hash-Table-Insert(T, key):
    collisions = 0
    i = 0
    x = 0
    repeat:
        if(i == m):
            error: “table overflow”
        x = h(key,i)
        i = i + 1
    until T[x] == NIL or T[x] == DELETED
    collisions = i - 1
    return collisions
```

The Algorithm

```
Hash-Table-Delete(T, key):
    collisions = 0
    i = 0
    x = 0
    repeat:
        if(i == m):
            error: “table overflow”
        x = h(key,i)
        i = i + 1
    until T[x] == NIL or T[x] == key
    collisions = i - 1
    if T[x] == key:
        T[x] = DELETED
    return collisions
```

```
Hash-Table-Search(T, key):
    collisions = 0
    i = 0
    x = 0
    repeat:
        if(i == m):
            error: “table overflow”
        x = h(key,i)
        i = i + 1
    until T[x] == NIL or T[x] == key
    collisions = i - 1
    return collisions
```

Algorithm Analysis

- Hash tables and their hashing algorithms are designed to support common array operations in constant time.
- For sparsely populated tables, this can be easily achieved, but as tables fill, collisions cause significant increases in time complexity.
- In the worst-case, insert, delete, and search take O(n) time!
- As we are testing n inserts, deletes, and searches, O(n) time complexity would result in an O(n^2) time graph for our tests. We want a linear graph.

Problem Statement

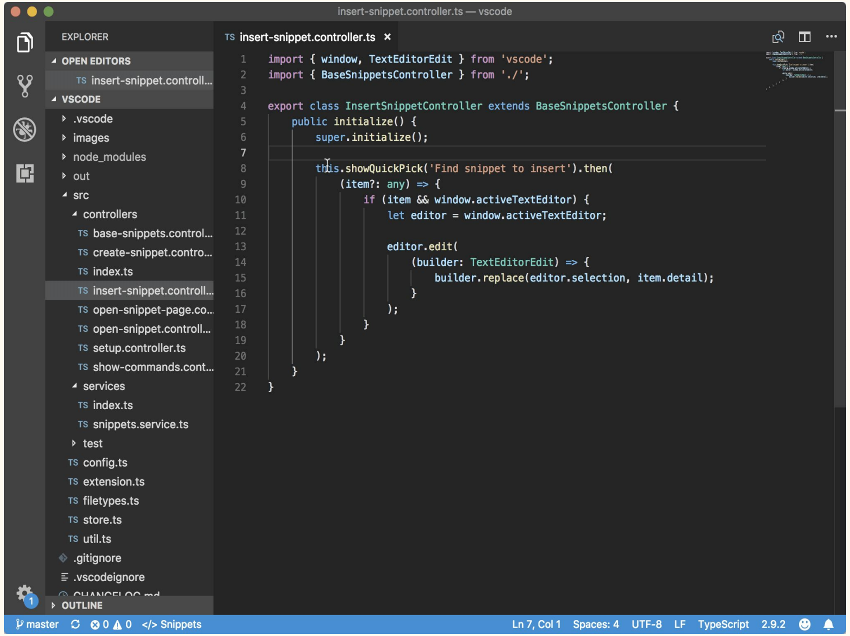
- How can we minimize collisions and runtime of hash tables?
- What parameters should we pick for the hashing algorithm?
- What form of collision resolution should we use?

Why does it matter?

- For large datasets, hash tables offer the distinct advantage of constant time access
- BUT, larger datasets are similarly likely to exhibit more collisions as the dataset size increases, reducing this advantage

Implementation

- We used C++
- For Open Addressing, we used std::vectors with integer type inputs
- For Chaining, we used an array of std::list
  - Insertions were pushed to the head of the list
  - O(1) time insertion
- Input Data
  - Uniformly random distribution
  - Varied input size
- Testing
  - Shell script which runs multiple trials with varied parameters



Demo Time!

Experimental Plan

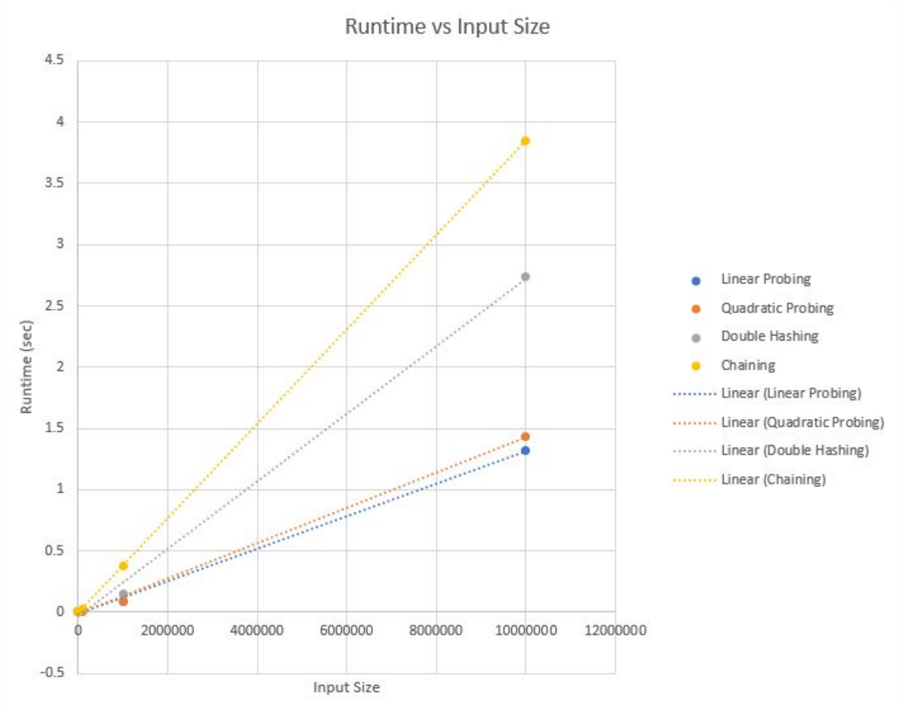
Our Dataset:

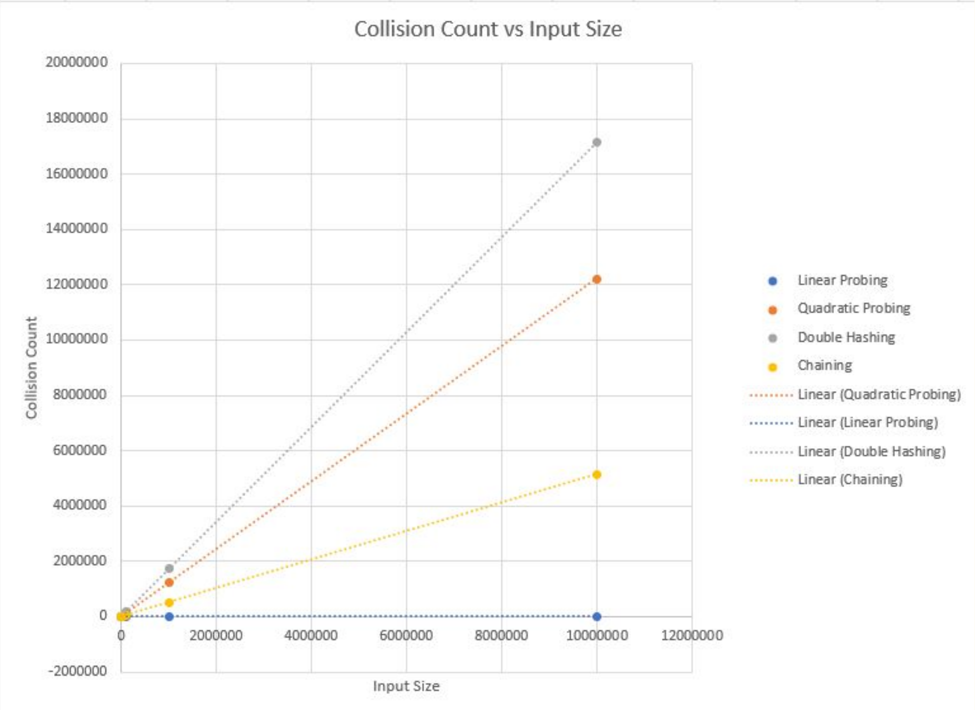
- Unsorted
- Uniformly distributed positive integers
- Size varied from [1,000, 10,000,000]
- Integer range varied from 0-10,000 and 0-2,000,000,000

Additional Constraints:

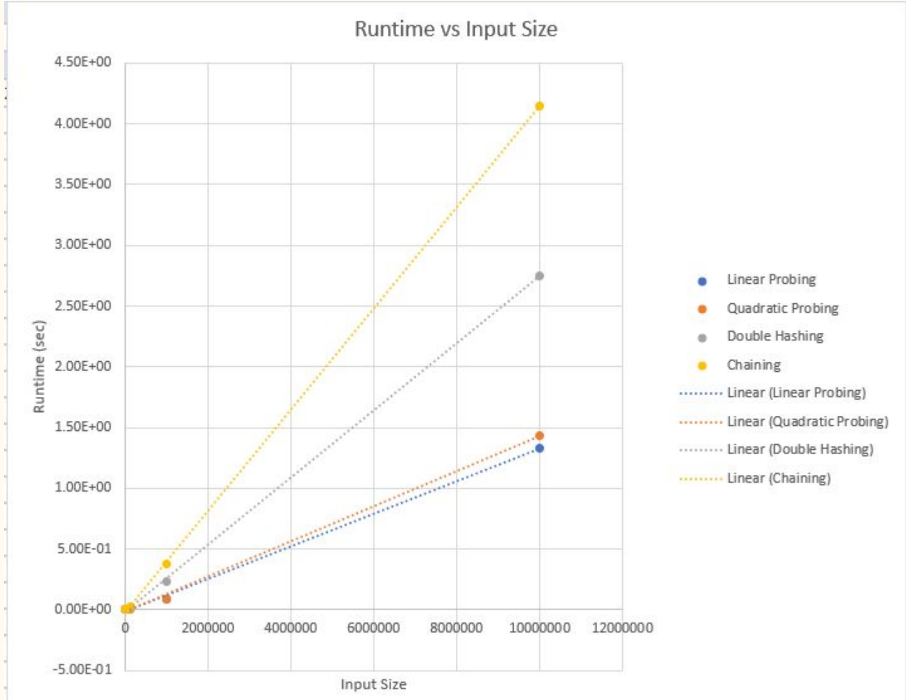
- We also varied these parameters
  - **m** (modulus)
  - **A** (multiplication factor)
  - $\mathbf{c_1}$ (quadratic constant 1)
  - $\mathbf{c_2}$ (quadratic constant 2)
- Measurable effects on collisions and load factor

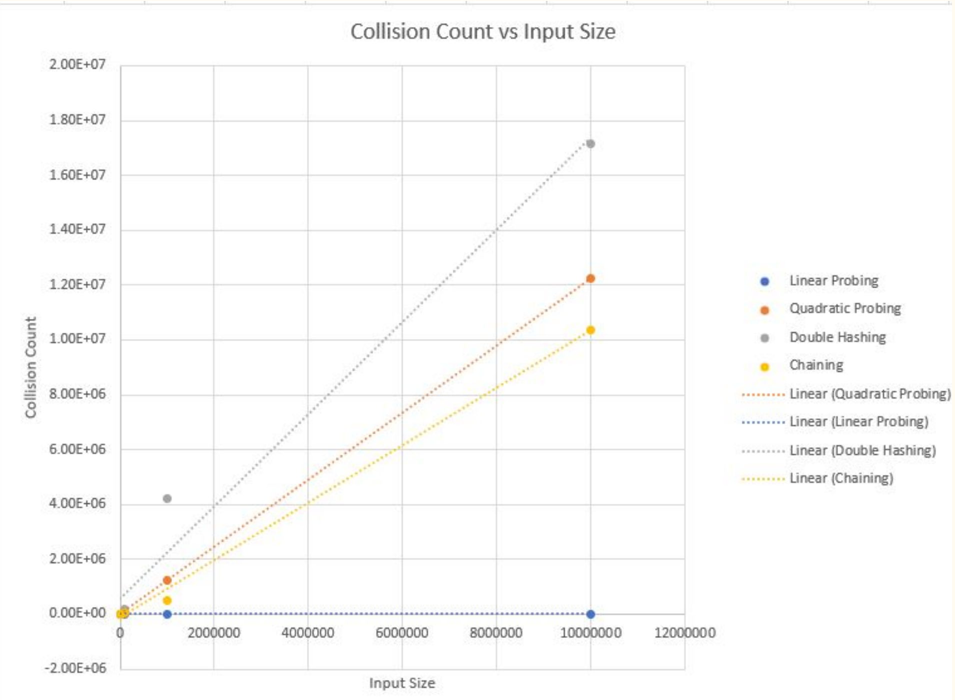
Results (division, m=size)



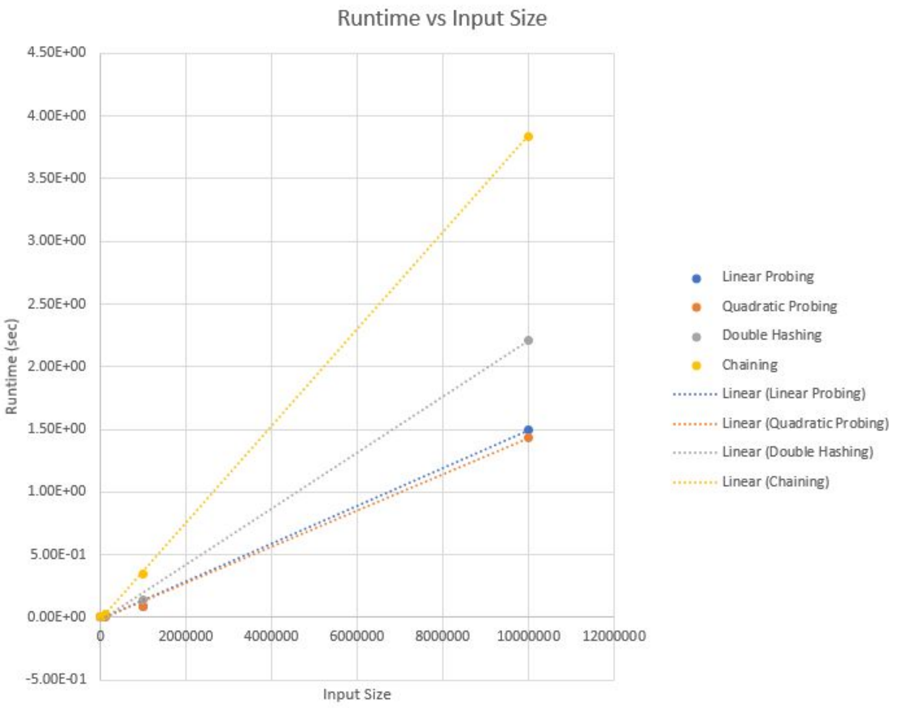


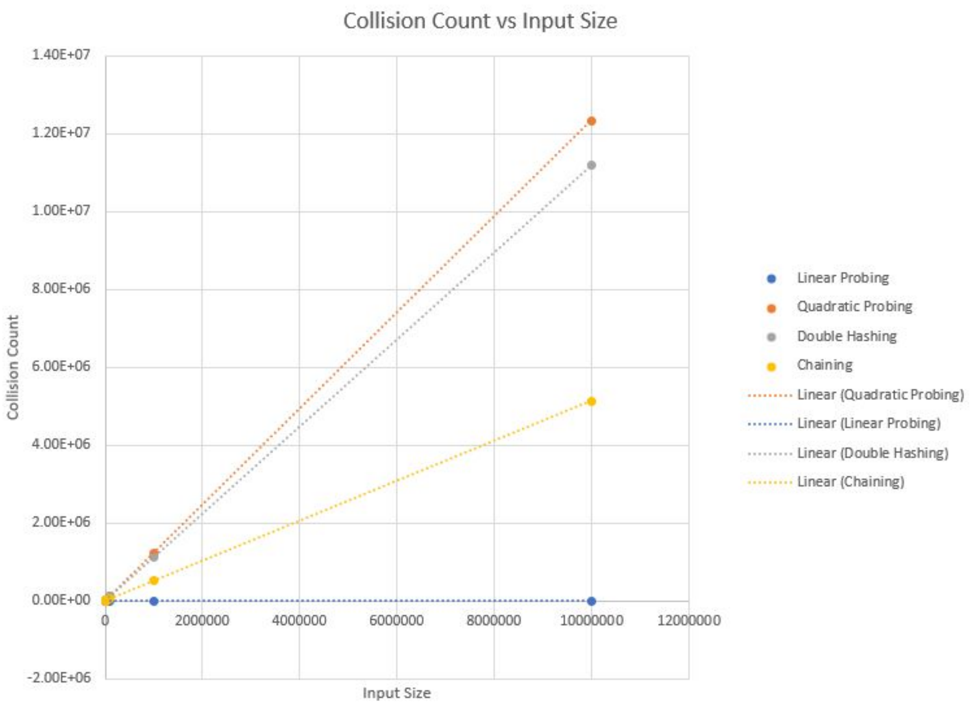
Results (multiplication, m=size, A=0.618)



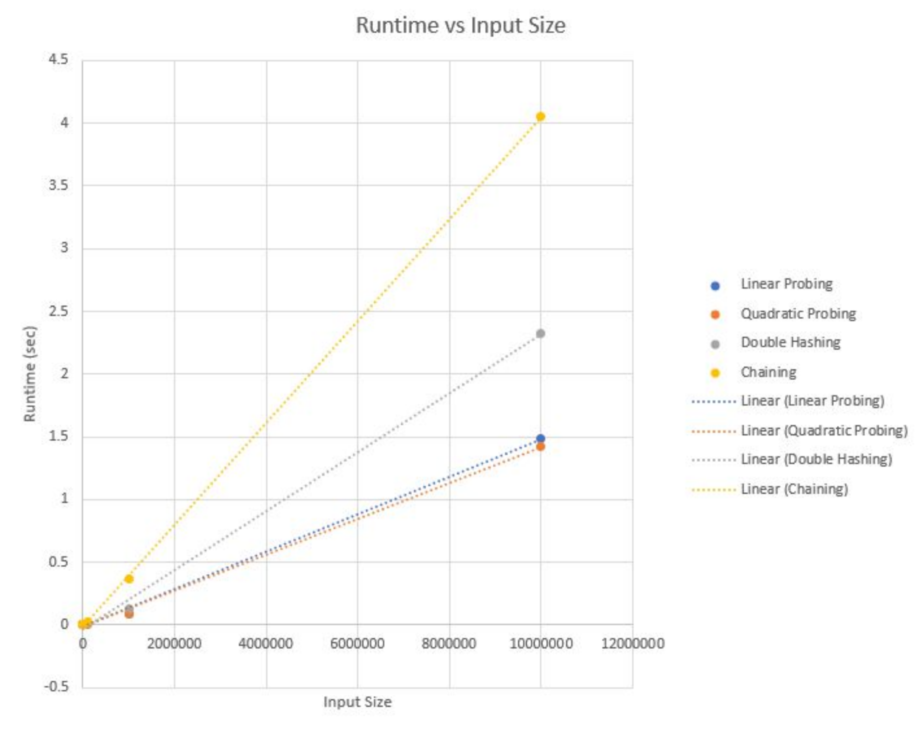


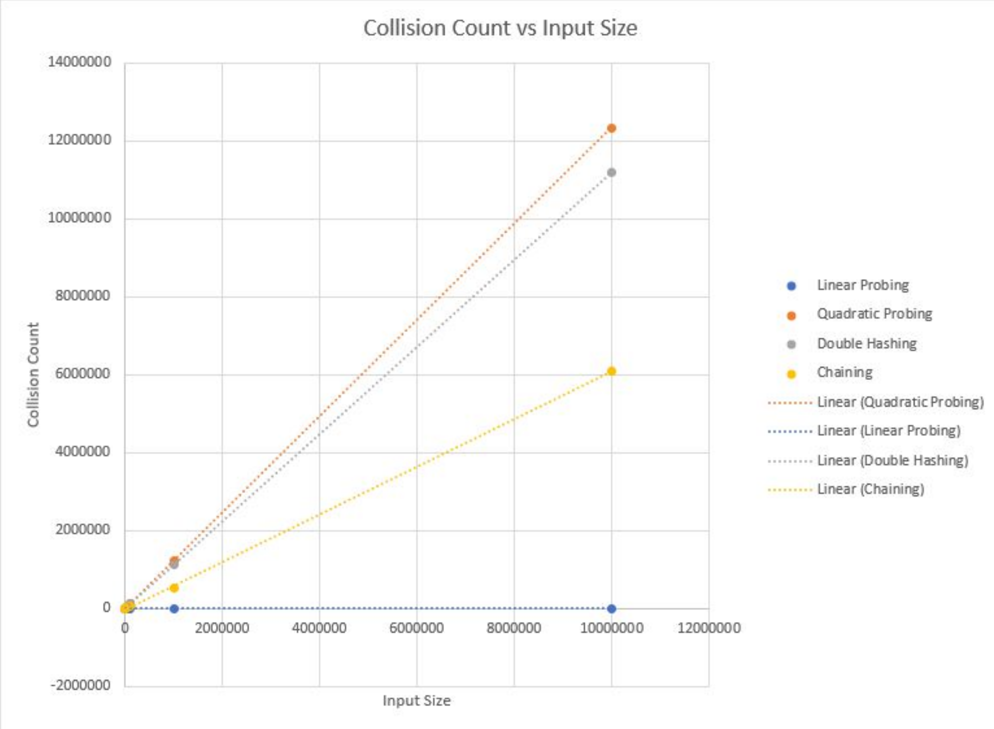
Results (division, m as a prime)



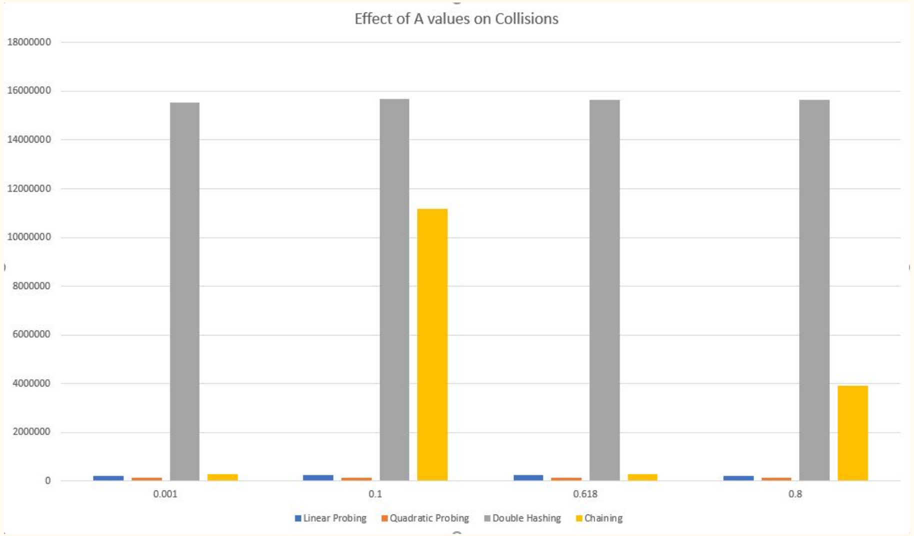


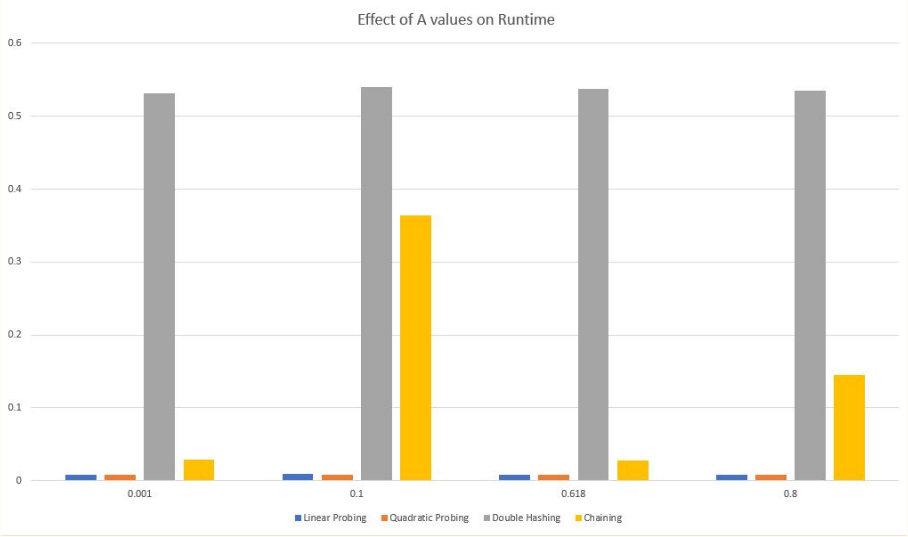
Results (multiplication, m is prime, A=0.618)





Results (Varying A values between 0.001 and 0.800)





Other Results

- Modifying m_1, m_2, c_1, and c_2 values slightly affected runtime and number of collisions:
  - For a dataset of 1,000 mostly-unique integers, quadratic probing exhibited little change in the collision count or runtime
  - For a dataset of 100,000 mostly-unique integers, double-hashing also exhibited little change, though prime numbers caused fewer collisions than non-primes
- Linear Probing ***really*** struggles with non-unique datasets.

Division method

	Collisions	Runtime
Linear Probing	215967999	34.2142s
Quadratic Probing	13732370	0.134439s
Double Hashing	2281994	0.055598s
Chaining	90896	0.0220713s

Multiplication method

	Collisions	Runtime
Linear Probing	214975379	34.7859s
Quadratic Probing	13724986	0.14852s
Double Hashing	2277108	0.0575159s
Chaining	92823	0.0264668s

Limitations and Future Work

What didn't we test?

- Different input distributions (i.e. normal distribution)
- Deleting different proportions of the input dataset
- Inserting different proportions of the input dataset

Our Thoughts About the Algorithms

Linear Probing

- Bad for data that contains duplicates
- Small hidden constants

Double Hashing and Quadratic Probing

- When m = size, double hashing did better
- When m = prime, quadratic probing did better
- Higher hidden constants

Chaining

- The best for mid-to-large, mostly unique, datasets

Overall, all Algorithms performed well, with close-to-constant time operations.

Concluding Remarks

- For datasets with few or repeated values, any form of collision resolution will produce “good enough”
- Chaining has the most overhead for smaller datasets
- If datasets are likely to contain repeated values, linear probing will be extremely inefficient.
    - It’s runtime increases significantly
- Prime number modulus values significantly reduce the likelihood of collisions
- Poorly selected constants in Quadratic Probing and Double Hashing can lead to unaddressable “holes” in the hash table!
- Chaining is heavily reliant on fitting the table size to the size of the dataset