John Goza
EN.605.620.81.SP23 – Algorithms for Bioinformatics
Lab 2 – Hashing
Due 12 Apr 2023 (One day extension granted by Dr. Rubey)

Abstract

In Lab 2 we implemented and compared two different methods of hashing (division based and student's choice), as well as different hash collision resolution methods (linear probing, quadratic probing, and chaining). Implementations were completed in Python. I chose to implement multiplication based hashing as the "student's choice" algorithm. While I expected a significant difference in performance between different hashing algorithms, both hashing algorithms had comparable results. When comparing collision handling methods, I found that both types of probing (linear and quadratic) were asymptotically similar, while chaining was the clear asymptotic winner (only for data insertion). Overall, I gained a stronger understanding of hashing, and a deep appreciation for hash function distribution patterns.

Data Structures and Design

The required input file for this assignment consisted of multiple blocks of integers separated by newlines. To parse the required input, I created an ingestion module called "input_handler.py" which was responsible for validation and error handling as well as ingestion. The module ingests the provided input file into a list of lists, with each inner list comprised of one block of integers. It checks for input file existence, validity of each line in the file, and for valid data as the end result of ingestion (i.e. the input file must contain at least one block of valid integers); if any of those checks fail, it throws an error with a user friendly description of the issue. The ingestion module intelligently assigns the integer lists by checking each line of the file based on the context of the previous lines. It will ignore lines in the input file that do not contain hashable values, which allows the user a significant degree of flexibility in annotation of their input file. The list of integer blocks is then returned to the main module ("Lab2.py") for hashing.

The main module references "config.py" for a full list of required hashing configurations provided in the project assignment. Each configuration is stored as an object, allowing for easy and consistent access to relevant keys (e.g. "hash_function", "collision_scheme", etc). When the main module is launched, it first parses and applies the CLI flags passed by the user (which are documented in the README). Then, it calls the ingestion module to read in the file, and then uses nested "for" loops to hash each input block with every required hashing configuration. For each block of input values, the main module initializes a table for storing hashed values, and then calls the hashing module, which, in turn, uses the hashing algorithm and collision resolution method specified in the current iteration of the configurations. The hashing module ("hasher.py") and the collision handling module ("collision_handlers.py") are described later this in report. After the specified hashing algorithm has been applied to each value in the current input list, the main module takes the aggregate statistics returned from the hashing function and feeds the final resulting table, aggregated statistics, list of successfully inserted values, and list of failed values to the output module ("output_handler.py"). This logic is repeated for each hashing configuration, and then the process is repeated for the next iteration of input values.

The hashing module contains four methods: "hash_values", "hash_by_multiplication", "hash_by_division", and "format_result". I shall describe the simplest functions first. The function "format_result" is a convenience method that is self descriptive. I format all the results into dictionaries in order to make output generation reliable and easy to read. The function "hash_values" is also very simple. It serves as the conductor for the values passed by the main module and routes them to the correct hashing function. It also aggregates the results returned by each hashing function call.

The most important methods in the hashing module are the “hash_by” methods. Starting with “hash_by_division”, we take a provided value, ensure that it can be converted to an integer, and then apply the simple modulus divisor hash specified in the assignment. If a value hashes to a key that is not currently in use, or that has available space in its bucket, we simply insert the value at that key and return the key and relevant metadata. If the key is currently occupied, or the bucket is full, then we apply the correct collision resolution function to the key and return results from that. Similarly, “hash_by_multiplication” checks the value for integer conversion, hashes the given value, and applies collision resolution if needed, but it has a very different hashing algorithm. I based the multiplication hashing algorithm off of the one found in page 264 of Cormen (Cormen); the full function can be seen in Figure 1. I used two logarithms to limit the range of the output to the table size, which was critical to avoid using a modulus operator. The “hash_by_multiplication” and “hash_by_division” methods both use the same collision resolution methods, which we will discuss now.

$$h(k)=\lfloor m(kA(kA-\lfloor kA\rfloor))\rfloor \qquad m=\ln(\text{table\_size})/\ln(2) \qquad A=(\sqrt{5}-1)/2$$

Figure 1) Equation for multiplication based hashing. Note that by using the ratio of logarithms the algorithm projects values into the correct range without the use of a modulus operator.

The probing collision resolution functions are defined in the “collision_handlers.py” file and both use the “probe” function. The collision resolution for chaining was extremely simple and straightforward, so that logic was included as part of both hashing functions. The “probe” function maintains a list of keys that the current value has collided with, and an integer count of the number of comparisons the current value has undergone. It handles both linear and quadratic probing through the use of defaulted parameters, and uses a modulus operator with the table size to prevent the exceeding of the table bounds.

The main module, hashing module, and collision resolution modules contain very little validation and error handling logic, as the input is thoroughly checked during the ingestion of the file. This follows the generally accepted good programming principle of “keeping error handling at the edges”, and allows the “business logic” of the program to remain clean and readable. Inversely, the ingestion module and output module both contain a significant amount of error handling in order to prevent errors from spreading beyond their scopes.

Results

This project provided a significant amount of insight into the inner workings of hashing and collision resolution methods, and provided an opportunity to compare and contrast each method. I began with running the required input file through the program. As the required input file had six lists to hash for each scheme specified in the assignment, I aggregated the results by scheme, which can be seen in Table 1. The results showed that my implementation of multiplication hashing encountered significantly more collisions than the division hashing implementation. More collisions necessitated more comparisons, leading to multiplication based hashing also having the highest average number of comparisons. A factor that may have negatively influenced the results of multiplication based hashing is float to integer conversion. My implementation of the hashing method aggressively converted values to integers, but a more “float friendly” version of the algorithm would theoretically have a domain with

	Hash Function	Modulo	Bucket Size	Collision Handling Scheme	Comparisons (Average)	Primary Collisions (Sum)	Secondary Collisions (Sum)	Load Factor (Average)
1	Division	120	1	Linear Probing	11.50	4	5	0.08
2	Division	120	1	Quadratic Probing	11.50	4	5	0.08
3	Division	120	1	Chaining	10	0	0	0.08
4	Division	113	1	Linear Probing	10.33	1	1	0.08
5	Division	113	1	Quadratic Probing	10.33	1	1	0.08
6	Division	113	1	Chaining	10	0	0	0.08
7	Division	41	3	Linear Probing	11.83	1	3	0.08
8	Division	41	3	Quadratic Probing	11.33	1	2	0.08
9	Multiplication	n/a	1	Linear Probing	14.67	27	1	0.08
10	Multiplication	n/a	1	Quadratic Probing	14.67	27	1	0.08
11	Multiplication	n/a	1	Chaining	10	0	0	0.08

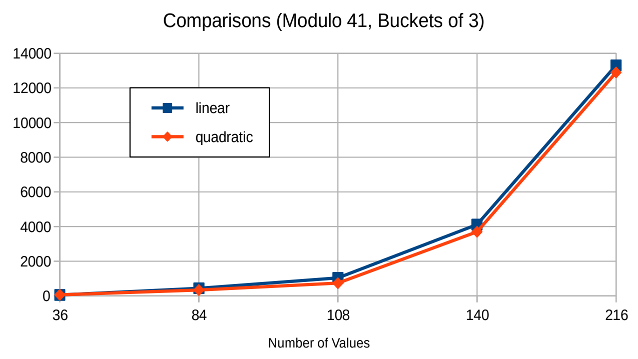
Table 1) Results of hashing required input for multiple schemes. Schemes are listed in the order specified by the problem statement. Required input had six lists to hash per scheme; values in this table are aggregated for all six lists.

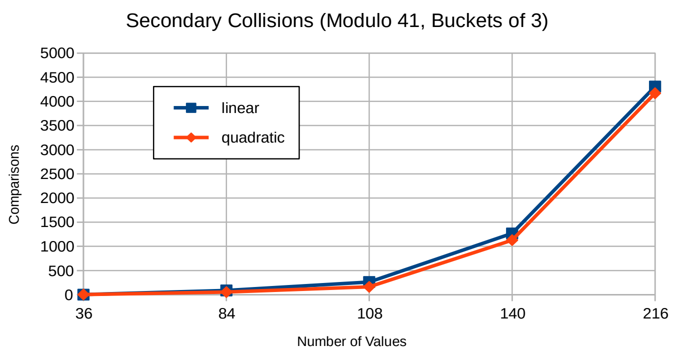
higher resolution. This could make an important difference in collision rates, and therefor reduce the number of comparisons.

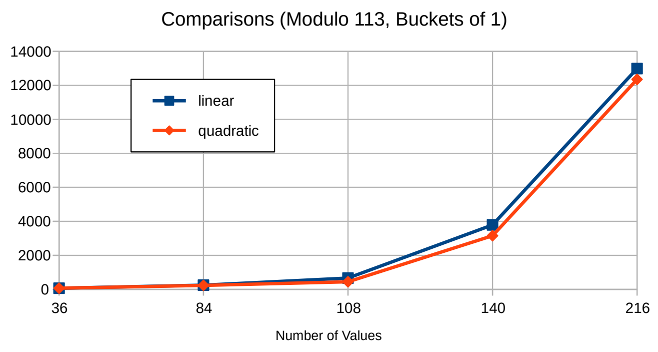
The evaluation of the division based hashing results suggests that modulo choice is an important factor for the efficiency of division based hashing. Division based hashing with modulo 113 outperformed both modulo 120 and modulo 41, mostly due to fewer collisions. These outcomes were heavily influenced by a low load factor. If the load factor had been higher, collisions would have been more common for all schemes. If I was to repeat this work, I would continue to investigate division based hashing by testing more modulos. It would be interesting to see if we could find a pattern where the best modulos can be found given a range 1 to N, and then determine the mathematical representation of that pattern.

Chaining outperformed both linear probing and quadratic probing for both division and multiplication based hashing, which was expected as the required implementation of chaining never had to perform addition key checks. The lack of key checks led to exactly ten comparisons for each scheme using chaining, as well as zero collisions. While chaining performed very well for insertion, the downside to this implementation would be evident upon item retrieval. As our in-table lists grew to accept more values at a given key, the efficiency of item search and retrieval would decrease. In a worst-case scenario, if all of our values clustered into the same key, retrieval time would become O(n) (assuming a linear search through the values). Chaining was also the most space inefficient, as the program was required to maintain a dynamic array inside of each table slot, rather than a simple string or integer as required by the probing schemes.

As an extra effort, I designed five additional input files of list sizes 36, 84, 108, 140, and 216 in order to determine asymptotic efficiencies of each scheme. I ran these files through the application for all of the required schemes, and assembled metadata reports together from their output. Finally, I created graphs of the most relevant result metadata, seen in Figure 2. The graphed output seems to show that the asymptotic efficiencies did not vary based on either the hashing algorithm or the probing method. This ran counter to my assumption that quadratic probing would have a significant performance hit when compared to linear, and that division and multiplication hashing would be asymptotically unique from each other. The load factor between these algorithms and probing methods were consistently equal, and varied solely on the number of values being hashed. This made sense, as the load on the hash table is calculated only from the table size and values stored, which did not change as a function of hashing algorithm or probing method. Chain comparison was not produced into a graph as the asymptotic efficiency of chain inserts remained constant at O(1). As previously discussed, the asymptotic efficiency of retrieval from a hash table constructed with chaining would be significantly worse than one constructed with probing algorithms, but that was not included as a part of this assignment.







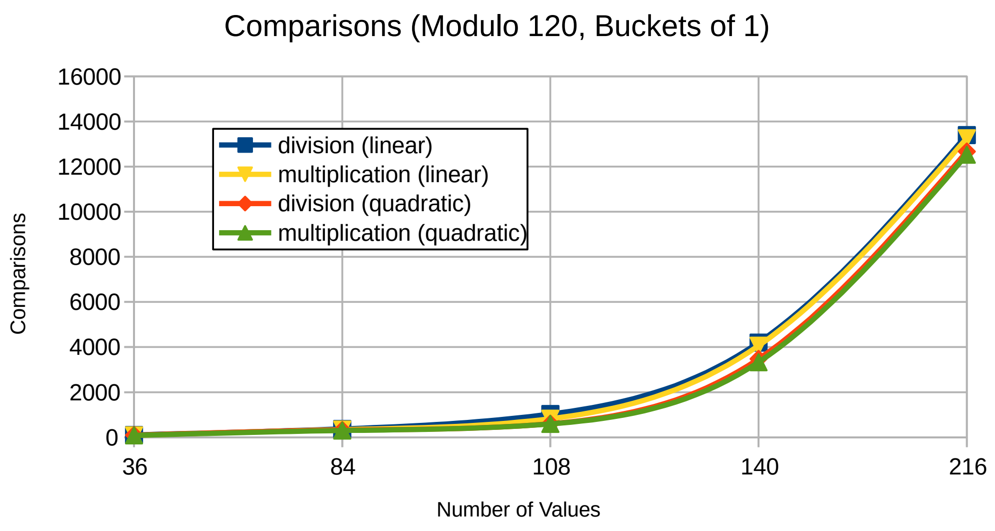


Figure 2) Comparisons of relevant statistics for similar schemes as number of values increases

Commentary on Bioinformatics Usefulness

Hash tables are essential to many modern computing problems, and their use cases in the context of Bioinformatics are numerous and diverse. We will discuss only two in this section as compiling a comprehensive list would be a substantial undertaking. Many bioinformatics applications perform sequence analysis by counting or otherwise grouping known patterns together. For example, codon to amino acid analysis requires splitting a given sequence into trimers and matching them to known codons. This problem can be simplified to a more generic statement: codon to amino acid analysis requires applying a reducing function to an input, and then bucketing reduced values. This is the core application of a hash table, making this structure a perfect choice for problems of this kind.

When handling sequence data as described above, sometimes the user of a given bioinformatics software will have multiple instances of the exact same sequence. To eliminate duplicates in a naive way would require the direct full-length string comparison of all sequences in the list, which is extraordinarily computationally expensive. A more intelligent software would use hashing. By applying a well-known hash function (such as SHA-1/2/3 or MD5), the software can significantly reduce the size of the comparison items, as it would only have to compare the hash results for equality rather than the full sequence. This would of course incur the time cost of the hashing algorithm, but for the purposes of this assignment that analysis is left as an exercise for the reader.

References (APA)

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Dynamic Programming. In *Introduction to algorithms* (pp. 405–405). essay, Mit Press