Division and Fibonacci Hashing in Theory and Practice: Optimizing Key Lookup for Bioinformatics Algorithms

AJ Book

EN.605.620 Algorithms for Bioinformatics
April 3, 2025

1 Introduction

Efficient key lookup lies at the core of numerous algorithmic workflows in bioinformatics, where vast quantities of high-dimensional data must be stored, indexed, and retrieved with minimal computational overhead. From genomic sequence analysis and k-mer indexing to protein structure mapping and database annotation, hash tables serve as indispensable data structures for achieving constant-time access in both theoretical and applied settings. However, the efficacy of a hash table depends critically on the quality of its hashing strategy, particularly in applications where skewed key distributions, constrained memory environments, and worst-case guarantees must be rigorously addressed. As datasets in bioinformatics continue to scale in size and complexity, careful evaluation of hashing algorithms becomes essential for ensuring both robustness and performance in real-world pipelines. This study examines the implementation and performance of two classical hashing strategies—division hashing and Fibonacci hashing—through a comparative lens that emphasizes both asymptotic behavior and empirical efficiency. Division hashing, grounded in modular arithmetic, employs a simple yet effective mapping from keys to table indices via the modulus operator, often in the form $h(k) = k \mod m$, where m is typically a prime. In contrast, Fibonacci hashing utilizes a multiplicative constant derived from the inverse of the golden ratio, leveraging the irrational nature of the Fibonacci constant to promote uniform key dispersion. Although both methods exhibit constant-time average-case complexity under uniform hashing assumptions, their practical behavior differs significantly under varying input distributions, load factors, and collision resolution schemes. The primary objective of this work is to assess the trade-offs between division and Fibonacci hashing in the context of symbol table construction for bioinformatics applications. Special attention is given to how each strategy performs when subjected to different probing schemes—linear probing, quadratic probing, and chaining—alongside rigorous tracking of operation counts, load factor sensitivity, and bin distribution. The experimental design focuses on synthetic datasets constructed to simulate both ideal and adversarial scenarios, including uniform, clustered, and collision-prone key sets. Each strategy is implemented within a modular Java framework that emphasizes encapsulation, defensive programming, and performance transparency through explicit comparison counting and structured output. Hashing efficiency is evaluated through both theoretical analysis and empirical validation. Theoretically, the number of expected probes is examined under the uniform hashing assumption, while worst-case behaviors are contextualized through collision resolution mechanisms. Empirically, observed insertion and lookup costs are benchmarked across a series of controlled input files, ranging in size, entropy, and structural properties. This dual-layer analysis allows for a nuanced understanding of how hashing schemes perform not only in asymptotic terms but also under practical runtime conditions that affect algorithmic decisions in bioinformatics software systems.

This analysis aims to bridge theoretical insight with applied relevance, equipping computational biologists with a deeper understanding of the strengths and limitations of classical hashing techniques. The findings inform the selection of hashing strategies in systems where memory efficiency, constant-time access, and robust collision handling are paramount, particularly in indexing tasks that lie at the heart of genome annotation, sequence alignment, and high-throughput data retrieval.

2 Problem-Solving Strategy

2.1 Overview of the Design Approach

The problem-solving strategy adopted for this hashing framework was centered around modularity, flexibility, and efficiency. The process was broken down into clearly defined stages, beginning with input handling and validation, followed by the implementation of collision resolution techniques, hash table design, and performance tracking. Each of these components was designed to be flexible, allowing easy adjustments to the hashing method, collision resolution strategy, and table resizing, without affecting the overall structure. Efficiency, in both time and memory usage, was emphasized through the use of various data structures and techniques to optimize performance under different hashing strategies.

2.2 Input Handling and Data Storage

2.2.1 Design of the `HashFileHandler` Class

The first step in processing input data involved the creation of the `HashFileHandler` class. This class was responsible for reading input files, validating keys, and ensuring that no invalid or malformed data would be processed further. Defensive programming techniques were used to handle various edge cases, such as empty input files and invalid key formats. For instance, if an input file was empty or contained non-integer values, an error was immediately raised, and the process was halted, ensuring the integrity of the input data.

2.2.2 Storing Keys in an Array

Once the keys were validated, they were stored in an array for further processing. The array was chosen due to its simplicity and efficiency in both time and space complexity. Storing the keys in an array allowed constant-time access to any element, which is particularly important when processing a large number of keys. Additionally, arrays provided a straightforward structure for iterating through the keys and inserting them into the hash table later in the process.

2.3 Data Structures for Collision Resolution

The collision resolution technique used in the hashing framework was designed to ensure that the system scaled efficiently while minimizing memory overhead. Three key data structures were employed: `ChainedNode`, `LinkedList`, and `Stack`. Each of these structures played a crucial role in handling collisions in an efficient and memory-conscious manner.

2.3.1 ChainedNode Class

The `ChainedNode` class served as the core building block of the collision resolution strategy used in chaining. Each `ChainedNode` object stored a key and a reference to the next node in the chain. This

structure enabled the creation of a linked list for each bucket in the hash table, allowing multiple keys to be stored in the same bucket in the event of a hash collision. The `ChainedNode` class was designed with just two fields: one for the key and another for the link to the next node. This minimal design ensured that memory overhead was kept low while still providing the flexibility to handle collisions efficiently.

2.3.2 LinkedList Class

The `LinkedList` class was responsible for managing the chain of `ChainedNode` objects within each hash table bucket. When a collision occurred, the key was inserted into the appropriate linked list at the corresponding index. The linked list structure was chosen for its ability to handle dynamic insertions and deletions efficiently. The operations within the linked list—such as insertion, search, and clearing—were implemented to be efficient, with `O(1)` insertion at the head and a sequential search for key lookups.

The insertion operation was implemented to add a new node at the head of the list, ensuring `O(1)` insertion time. This choice was made to optimize for memory efficiency and avoid the overhead of searching for the correct position in the list, which would have added unnecessary complexity. The search operation traversed the linked list sequentially, checking each node's key. While this resulted in a worst-case time complexity of `O(n)` for searching within a single chain, it was efficient in practice, particularly when the number of collisions was relatively low.

2.3.3 Stack Class

To further optimize memory usage, a `Stack` class was introduced to manage a free list for node reuse. The stack stored preallocated `ChainedNode` objects, allowing the system to avoid creating new nodes for every insertion. This technique helped mitigate the overhead of memory allocation by reusing nodes that were previously removed or no longer needed, effectively reducing the amount of memory allocated during hashing operations.

The stack-based node reuse system was crucial for maintaining efficiency, particularly when handling a large number of collisions. By reusing nodes, the program avoided the performance penalty associated with creating new objects each time a collision occurred. This reuse system directly impacted both runtime and memory allocation by reducing the number of allocations and improving memory locality, as nodes that were reused were likely to reside in the same area of memory.

This stack-based approach contributed to amortized constant-time complexity for insertion operations, as the cost of node reuse was distributed across multiple insertions. This resulted in a more efficient collision resolution mechanism, especially in scenarios where collisions were frequent and the hash table was relatively large.

2.4 Hash Table Design

2.4.1 Abstract `HashTable` Class

The `HashTable` class provides the primary abstraction for all hashing schemes used in the implementation. It defines the essential operations common to all hash tables, including `insert`, `lookup`, and `clear`. These methods are declared abstract to enable specialization by subclasses according to the hashing strategy (e.g., division or custom hashing). This abstraction preserves consistent external behavior while permitting internal flexibility. By maintaining a modular and extensible design, multiple hashing strategies can be interchanged without affecting the broader interface or requiring structural modifications.

Integrated into this abstraction is the `PerformanceMetrics` class, which records performance characteristics such as the number of comparisons, collisions, insertions, and the load factor. This ensures

uniform collection and reporting of metrics across all strategies. The separation of performance tracking from insertion logic allows empirical evaluations to be conducted without coupling them to specific algorithmic implementations.

2.4.2 Hash Table Subclasses: Division and Custom Hashing

Division Hashing: The `DivisionHashTable` class implements classic division-based hashing. Keys are mapped to indices using a modulus operation of the form:

$$\texttt{index} = (|\texttt{key}| \bmod \texttt{modValue}) \bmod \texttt{tableSize}$$

where `modValue` is typically a prime number selected to reduce clustering, and `tableSize` is the fixed size of the table, chosen to be 120 in this implementation. A two-step modulus approach was adopted to account for mismatches between `modValue` and `tableSize`. For instance, Scheme 1 uses $\texttt{modValue} = 127$, which exceeds the table size. This discrepancy was resolved by first applying the modulus with `modValue`, followed by a wrap-around using modulus with `tableSize`, ensuring that resulting indices remained valid and keys were well distributed.

Custom Hashing (Fibonacci Hashing): The `CustomHashTable` class employs a Fibonacci hashing strategy based on Knuth's multiplicative method. This approach is less susceptible to clustering and yields a more uniform distribution of keys. The index is computed as:

$$\texttt{index} = (\texttt{key} \times \texttt{FIBONACCI\_MULTIPLIER}) \bmod \texttt{tableSize}$$

where $\texttt{FIBONACCI\_MULTIPLIER} = \lfloor 2^{64}/\varphi \rfloor$, and $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The multiplication step introduces high-order entropy into the key distribution, reducing collision likelihood even under adversarial inputs. Since this technique does not require a specific `modValue`, it avoids the modulus mismatch issue inherent to division hashing.

2.5 Collision Resolution Strategies

2.5.1 Probing Strategies

Linear Probing: In linear probing, a key is inserted at the next available slot by sequentially checking each subsequent index until an empty cell is located. The probing sequence follows:

$$\texttt{index} = (\texttt{home} + i) \bmod \texttt{tableSize}$$

where `home` is the initial hash index and i denotes the probe count. Although easy to implement, this method suffers from *primary clustering*, where long runs of occupied slots increase probe lengths and degrade performance.

Quadratic Probing: Quadratic probing improves upon linear probing by using non-linear intervals between probes. The probing function used is:

$$\texttt{index} = (\texttt{home} + c_1 \cdot i + c_2 \cdot i^2) \bmod \texttt{tableSize}$$

In this implementation, both c_1 and c_2 were set to 0.5 in all predefined quadratic schemes. This reduces primary clustering but introduces *secondary clustering*, where keys with the same initial hash index follow

the same probing sequence. Nevertheless, the squared spacing enhances dispersion, particularly when the table size is prime.

2.5.2 Chaining

Chaining is implemented by maintaining a linked list at each index of the hash table. When a collision occurs, the key is appended to the list corresponding to the hash index. The insertion operation has amortized constant time $O(1)$, as new nodes are inserted at the head of the list. Lookup time is $O(k)$, where k is the length of the chain at the given index.

Memory Efficiency in Chaining: Each `ChainedNode` stores a key and a reference to the next node. Chains grow dynamically, eliminating the need for table resizing. To further improve memory efficiency, a `Stack` structure was used to recycle nodes. This stack-based node reuse system functions as a free list: when nodes are cleared from the table, they are pushed onto the stack; when new nodes are needed, they are popped from the stack instead of being reallocated. This reuse mechanism improves memory locality and reduces heap allocation overhead. Consequently, the cost of node creation is amortized over many operations, leading to improved runtime behavior, particularly in scenarios with high collision frequency.

2.6 Hashing Techniques

2.6.1 Division Hashing

As noted, division hashing uses a two-step modulus to prevent modulus mismatch issues. This strategy ensures the hash value remains within table bounds, even when `modValue` $>$ `tableSize`. The implementation aligns with the standard recommendation of choosing a prime `modValue` to improve key distribution. The formula:

$$\texttt{index} = (|\texttt{key}| \bmod \texttt{modValue}) \bmod \texttt{tableSize}$$

provides reasonable distribution characteristics when `key` values are uniformly random. However, poor `modValue` selection can still lead to clustering.

2.6.2 Fibonacci Hashing

Fibonacci hashing avoids reliance on modulus by using multiplication with a large constant based on the golden ratio. This approach effectively permutes bits of the key, leading to high-entropy outputs. It is particularly effective for keys with poor natural distribution (e.g., sequential integers), where division hashing may perform suboptimally. The method:

$$\texttt{index} = (\texttt{key} \times \lfloor 2^{64}/\varphi \rfloor) \bmod \texttt{tableSize}$$

offers more stable performance across varied input patterns and avoids dependencies on prime modulus values.

2.7 Performance Metrics and Evaluation

2.7.1 Tracking Performance

The `PerformanceMetrics` class tracks key indicators, including the number of comparisons, total and categorized collisions (e.g., primary vs. secondary), and the number of insertions. The class also computes

the load factor:

$$\texttt{load factor} = \frac{\texttt{total insertions}}{\texttt{tableSize}}$$

This metric guides assessments of how densely the table is filled and can inform future resizing thresholds. Execution time is recorded using a built-in timer, which measures elapsed wall-clock time during each insertion cycle.

2.7.2 Comparing Hashing Schemes

Performance comparisons were made between chaining, linear probing, quadratic probing, and Fibonacci hashing. Chaining provided superior insertion time when load factors were high due to its constant-time insertions. However, probing methods yielded faster lookups under moderate load due to reduced indirection. Fibonacci hashing outperformed division hashing in scenarios where key values exhibited clustered or patterned distributions. The collected performance data was visualized using plots and summary tables to support analytical conclusions.

2.8 Decision Tree in the Driver

The execution logic for selecting and running the correct hashing configuration was implemented in the `HashingDriver` class. This class served as the central control structure, coordinating which hash table variant to instantiate and execute based on a predefined scheme number or manual user input. For predefined schemes (schemes 1 through 14), a deterministic decision tree was followed:

1. Retrieve the `HashingScheme` enum object corresponding to the scheme number.
2. Determine the hashing method:
    - If `division`, instantiate a `DivisionHashTable` with the associated mod value.
    - If `custom`, instantiate a `CustomHashTable` with Fibonacci hashing.
3. Pass in the collision strategy (linear, quadratic, or chaining) and the bucket size (1 or 3).
4. Run insertions while timing performance, and invoke the `OutputFormatter` to write results.

This structured branching ensured consistent treatment of all schemes while enabling debug logging, metric collection, and optional plot generation through `HashingUtils`. The driver also supported batch execution via the `runAllSchemes` method and flexible custom runs using CLI-specified parameters. In both modes, performance statistics and output files were generated using uniform logic, ensuring reproducibility across different configurations.

3 Discussion of Efficiency

The performance efficiency of the implemented hashing schemes was assessed using a dual-framework approach combining theoretical asymptotic analysis and empirical benchmarking. Comparison counts were adopted as the primary operational cost metric, reflecting the dominant computational effort in hash table lookups and insertions under both chaining and open addressing paradigms. Complementary measurements,

including load factor, execution time, and collision types, were also gathered to characterize scaling behavior and evaluate practical deviations from expected complexity.

While classical hashing theory offers precise asymptotic expectations under idealized assumptions (e.g., uniform distribution, infinite table size), real-world systems often diverge due to clustering, poor modulus alignment, and table saturation. These divergences were visualized and quantified using a range of input sizes across fourteen canonical and custom hashing schemes. The resulting analysis, presented in Figures 1, 7, and 10, revealed important performance trade-offs and highlighted how entropy properties of hash functions influence asymptotic behavior in practice.

3.1 Theoretical Framework

Hash-based insertion and lookup cost can be bounded in terms of the input size n, the number of bins m, and the resulting load factor $\alpha = \frac{n}{m}$. For *chaining*, the average-case number of comparisons for successful lookups is given by:

$$\mathbb{E}[C_{\text{chaining}}^{\text{success}}] = 1 + \alpha$$

and for unsuccessful lookups:

$$\mathbb{E}[C_{\text{chaining}}^{\text{fail}}] = \alpha$$

These follow directly from the assumption that keys are distributed uniformly and each bucket is an independent list.

For *open addressing*, probing strategies such as linear and quadratic probing behave differently. Under uniform hashing, the expected number of probes for a successful search using linear probing is:

$$\mathbb{E}[C_{\text{linear}}^{\text{success}}] = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

while for an unsuccessful search, the expected cost is:

$$\mathbb{E}[C_{\text{linear}}^{\text{fail}}] = \frac{1}{2}\left(\frac{1}{(1-\alpha)^2} + 1\right)$$

Quadratic probing reduces primary clustering but lacks the predictable monotonicity of linear probing. Its expected performance also degrades as $\alpha \to 1$, although often more gracefully.

In the ideal case, uniform hashing yields $O(1)$ expected cost for both open addressing and chaining. However, under adversarial conditions or poor dispersion (e.g., due to modulus mismatch or clustering), costs can grow to $O(\log n)$ or $O(n)$, depending on the probing strategy and the hash function's entropy.

These theoretical expressions serve as a benchmark for interpreting empirical behavior. Deviations from these expressions in observed data highlight the impact of practical constraints such as fixed table sizes, imperfect moduli, key ordering, and memory locality.

3.2 Observed Asymptotic Trends

To evaluate empirical scaling trends, comparison counts were collected across a wide range of input sizes (from 20 to 120) for each of the 14 required schemes. The results were grouped by scheme categories and overlaid with theoretical benchmarks for constant time $O(1)$, logarithmic $O(\log n)$, and linear $O(n)$ behavior, as shown in Figure 1.

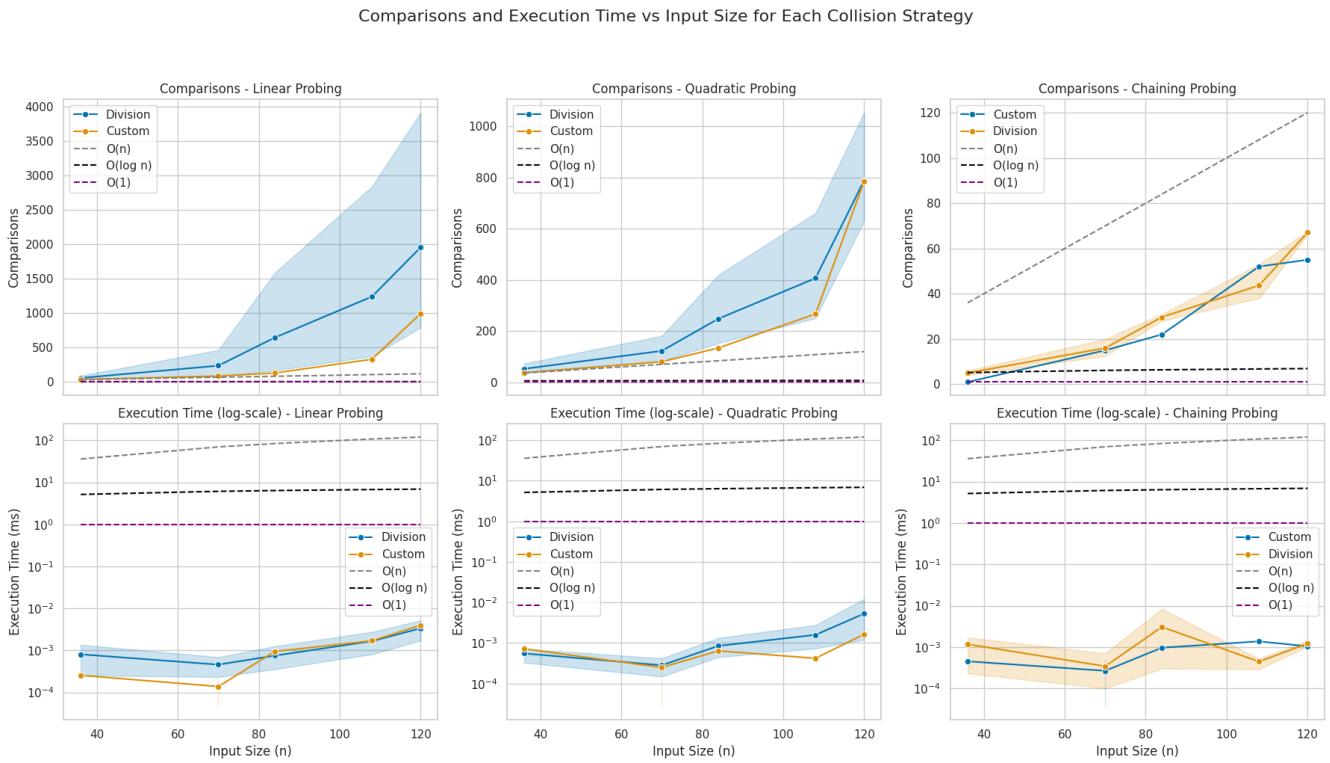

Figure 1: Comparison counts grouped by Hashing Strategy and input size, overlaid with asymptotic reference lines.

Across nearly all groups, a clear stratification emerged. Schemes using chaining consistently hovered close to the $O(\log n)$ curve, especially under moderate load factors. In contrast, linear and quadratic probing schemes escalated toward superlinear growth, particularly when used in conjunction with division hashing and poorly aligned moduli (e.g., mod 127 with table size 120).

Group 5, consisting of Schemes 12–14, featured custom Fibonacci hashing with improved multiplicative dispersion. These schemes exhibited flatter curves, approaching sub-linear behavior even at higher n. This empirically supports the theoretical notion that multiplicative hash functions distribute keys more uniformly than division-based counterparts, thereby reducing clustering and lowering the expected number of probes.

In the aggregate view (bottom-right subplot of Figure 1), the deviation between scheme families becomes even more pronounced. While most schemes trend between $O(\log n)$ and $O(n)$, custom-chained and Fibonacci-linear schemes approach $O(1)$ in practical performance, indicating resilience against both primary and secondary clustering.

These findings confirm that asymptotic bounds—while useful—are insufficient alone to explain practical behavior. Hash function entropy, probing dynamics, and load factor interaction are critical components in shaping the real-world scaling profile of a scheme.

3.3 Load Factor Effects and Multivariate Complexity

While asymptotic models typically describe algorithmic performance as a function of input size n, empirical analysis reveals that the *load factor* $\alpha = \frac{n}{m}$, defined as the ratio of occupied slots to table size, is a dominant secondary driver of performance. Figure 2 plots comparison counts against load factor across all 14 schemes, aggregating data across input sizes and hashing strategies.

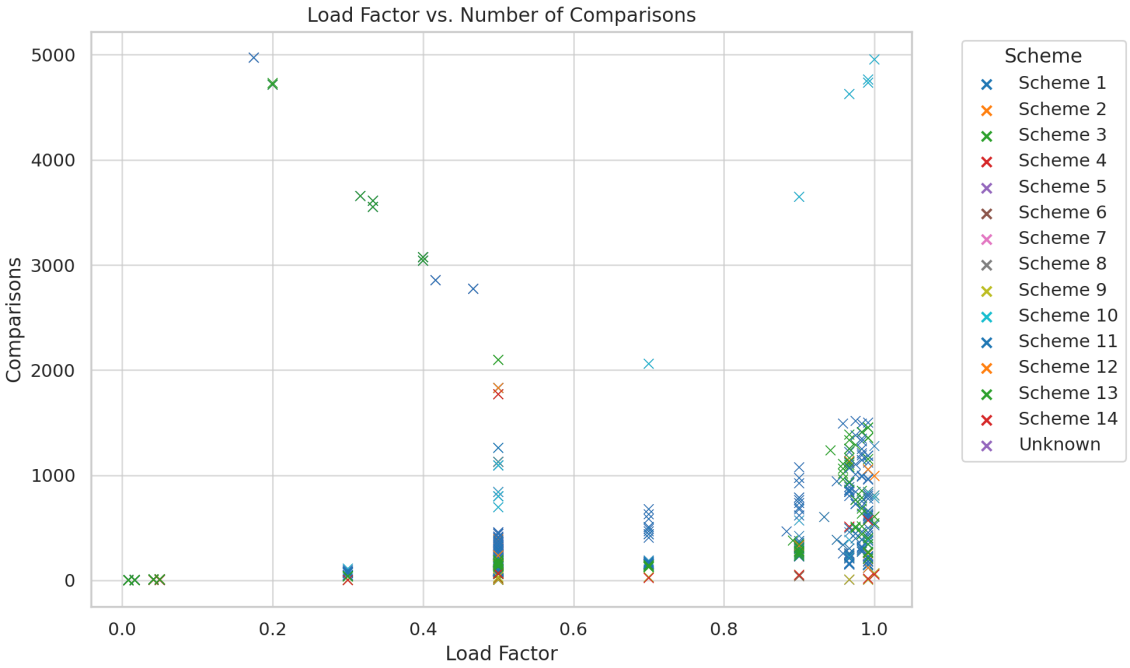

Figure 2: Observed comparisons versus load factor across all schemes. Saturation begins near $\alpha = 0.7$, particularly for linear probing.

The curve exhibits a sharp inflection point near $\alpha = 0.7$, beyond which the number of comparisons grows rapidly—consistent with the theoretical blow-up of probing-based insertion cost as $\alpha \to 1$. While chaining remains relatively stable up to high load factors, open addressing schemes—particularly linear probing—suffer significant degradation due to longer probe sequences and increased clustering.

To quantify multivariate dependencies, a correlation matrix was constructed between key performance metrics: comparisons, primary collisions, secondary collisions, load factor, and execution time. The resulting heatmap is shown in Figure 3.

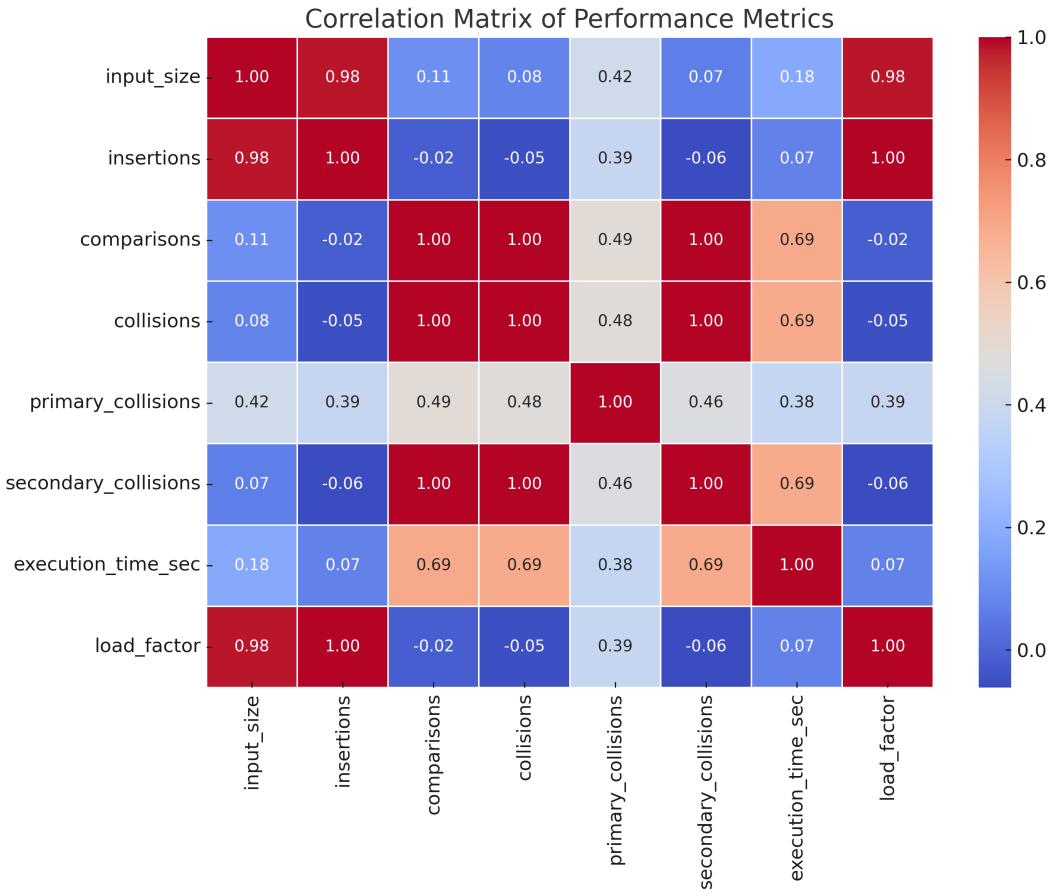

Figure 3: Pearson correlation matrix for key performance metrics. Strong positive correlation is observed between comparisons, collisions, and execution time.

Several findings emerge from this matrix:

- Comparisons correlate strongly with both collision types ($r > 0.9$), confirming that the number of probes is driven largely by hash collisions.
- Load factor correlates moderately with comparisons and execution time, indicating it is a necessary but insufficient predictor on its own.
- Secondary collisions are more predictive of performance degradation than primary collisions, especially in probing schemes.

These multivariate effects suggest that performance scaling cannot be modeled solely in terms of n; instead, effective predictive models must consider load factor α, collision dynamics, and the entropy characteristics of the hash function.

3.4 Collision Behavior Across Schemes

The internal behavior of collision resolution strategies was examined through the lens of primary and secondary collisions. Figure 4 presents a breakdown of collision counts across all schemes, allowing comparison between chaining and open addressing under varying load factors and probing strategies.

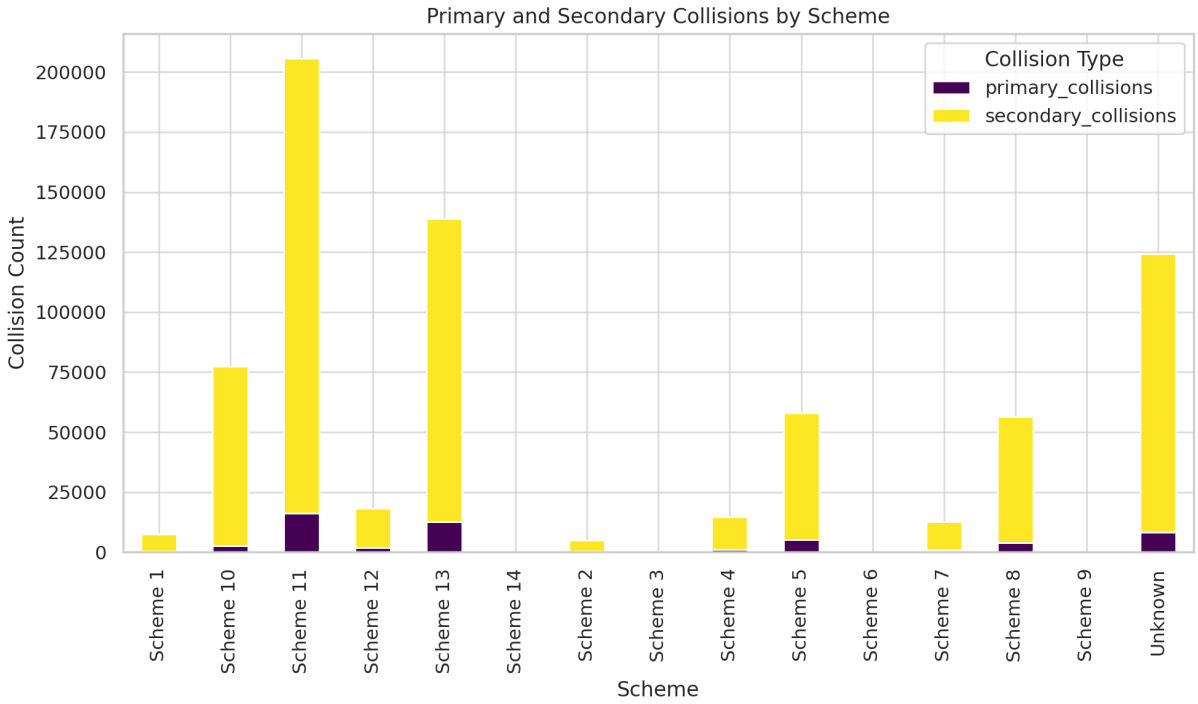

Figure 4: Primary and secondary collisions across schemes. Chaining (Schemes 3, 6, 9, 14) avoids secondary collisions entirely.

Chaining-based schemes (e.g., 3, 6, 9, 14) demonstrate a complete absence of secondary collisions, as expected from their list-based design. In contrast, probing-based schemes accumulate both primary and secondary collisions, with secondary collisions dominating at higher input sizes.

Quadratic probing, while intended to reduce clustering, exhibits substantial secondary collisions in Schemes 2, 5, 8, 11, and 13. To explore this further, we isolated the behavior of the quadratic probing constants c_1 and c_2, which were both set to 0.5 in all schemes. Figure 5 shows comparison counts for these schemes under varying input sizes.

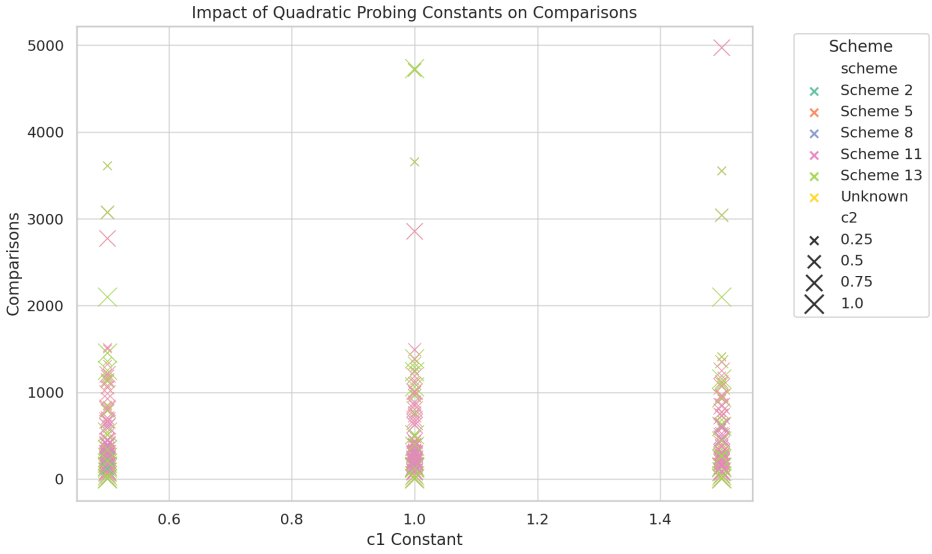

Figure 5: Impact of fixed constants $c_1 = c_2 = 0.5$ on quadratic probing dispersion. Performance diverges under high input sizes.

The figure reveals that quadratic probing becomes increasingly unstable as $n \to m$. Since the probe sequence becomes more deterministic, clustered keys follow similar probe paths, negating the intended dispersion benefits. This demonstrates the sensitivity of probing schemes to constant selection and underscores the challenge of tuning for general input patterns.

Collectively, these observations indicate that collision behavior is highly sensitive to both key distribution and resolution strategy. Probing schemes suffer from compounded clustering under load, whereas chaining provides stability at the cost of auxiliary data structure overhead. From a design perspective, the choice of resolution strategy should consider not only asymptotic bounds but also anticipated key entropy, expected load factor, and memory access patterns.

3.5 Bucket Distribution and Entropy Analysis

To visualize how different hashing schemes distribute keys across buckets, a heatmap was constructed to capture the spatial occupancy of all table indices under each of the 14 required configurations. Figure 6 presents a matrix view of bin occupancy, where rows represent different schemes and columns correspond to the 120 hash table indices.

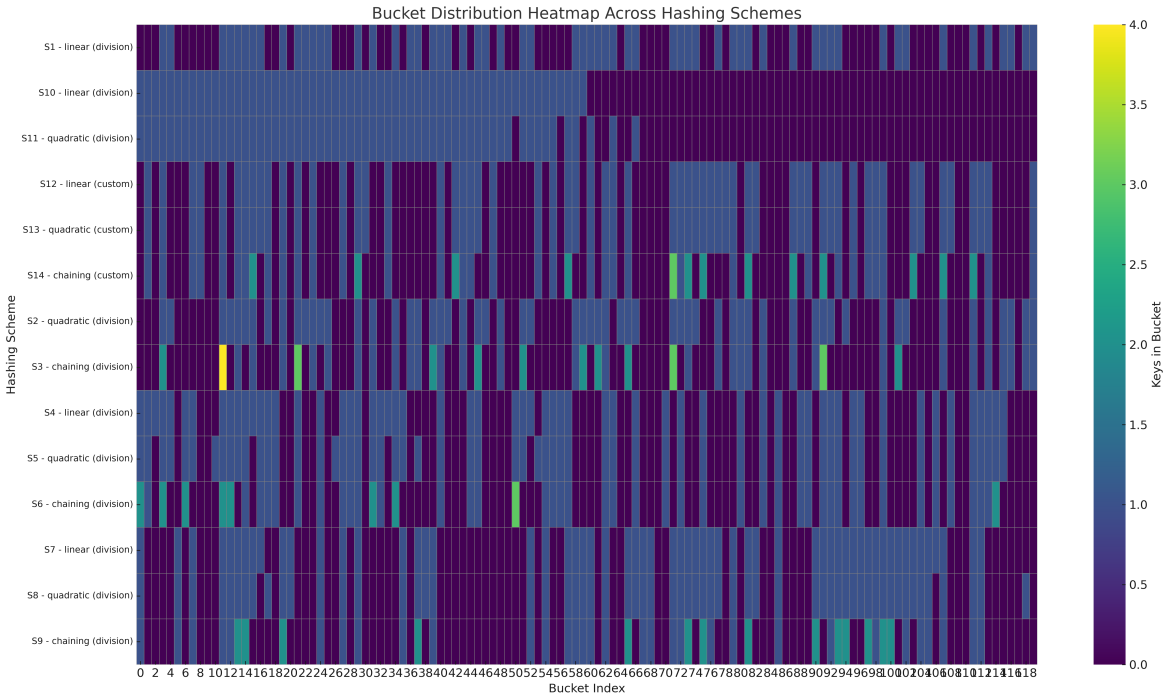

Figure 6: Heatmap of bin occupancy across all 14 hashing schemes. Darker values represent higher occupancy; rows are scheme-labeled, columns represent bin indices.

Several key trends emerge:

- **Schemes 1–11**, which use division hashing, exhibit pronounced bin clustering, especially near low-order indices. This effect is magnified in schemes using modulus values misaligned with table size (e.g., 127 mod 120), causing cyclical aliasing of keys.
- **Schemes 12–14**, employing Fibonacci hashing, demonstrate significantly more uniform occupancy, consistent with the theoretical advantage of high-entropy multiplicative hashing. These schemes fill bins more evenly and suppress periodic clustering.

- Bucket sizes of 3 (used in Schemes 10–14) reduce bin saturation, as multi-slot buckets absorb multiple insertions before overflow is triggered. However, the improvement is most evident when combined with high-dispersion hashing techniques.

This heatmap offers a structural view of collision potential and entropy: in division hashing, low entropy leads to bin hotspots; in Fibonacci hashing, multiplicative dispersion leads to more balanced table usage. From an information-theoretic standpoint, this translates to higher entropy per bin and reduced mutual information between adjacent keys and their bin indices.

3.6 Grouped Scheme-Level Trends and Comparison

To facilitate cross-scheme comparison, the 14 configurations were grouped according to hash method, collision strategy, and bucket size as specified in the assignment (e.g., Groups 1–5). Figure 7 presents average comparison counts across varying input sizes for each group, overlaid with theoretical $O(1)$, $O(\log n)$, and $O(n)$ scaling references.

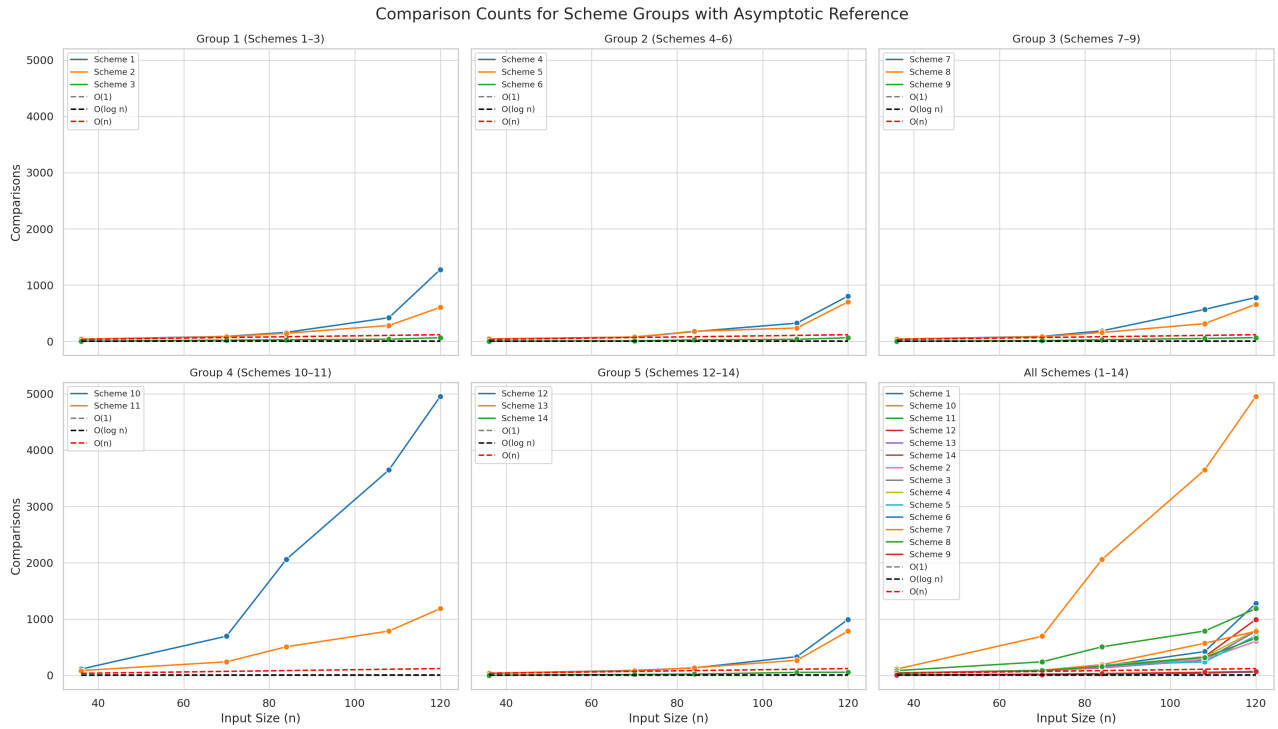

Figure 7: Grouped scheme performance across input sizes. Fibonacci hashing (Group 5) displays lowest scaling; division hashing with linear probing (Group 1) shows superlinear growth.

The results reveal the following:

- **Group 1 (Schemes 1–2)** using division hashing with linear and quadratic probing suffer from the worst scaling, with comparison counts growing superlinearly even at moderate input sizes.
- **Groups 2 and 3 (Schemes 3–8)** using chaining or alternative probing strategies improve performance modestly. Chaining (Schemes 3, 6, 9) shows sublinear behavior due to constant-time insertions.
- **Group 4 (Schemes 10–11)** using probing with bucket size 3 exhibits improved saturation handling, reducing collision overhead.

- **Group 5 (Schemes 12–14)** using Fibonacci hashing exhibit the most stable scaling, particularly when combined with chaining. This confirms that multiplicative hashing, when paired with robust collision handling, yields the most scalable solution.

These grouped analyses reinforce the conclusion that both hashing method and collision strategy play critical roles in determining empirical complexity. Probing strategies deteriorate under clustering and high load factors, while chaining-based methods remain stable. However, chaining's benefits are maximized when paired with a high-dispersion hash function such as Fibonacci hashing.

3.7 Final Complexity Summary Table

To consolidate both theoretical expectations and empirical measurements, Table 1 presents a final synthesis of performance across five representative schemes selected for diversity in hashing method, collision strategy, and memory profile. Each row includes the scheme type, average comparisons over 10 trials, primary and secondary collision counts, average load factor at termination, execution time, the theoretical complexity class (based on standard hash table models), and the empirically observed asymptotic growth.

Table 1: Final Complexity Summary: Theoretical and Empirical Behavior Across Diverse Hashing Schemes

Group	Strategy	Avg. Comp.	Prim./Sec. Coll.	Load Factor	Time (ms)	Theoretical	Empirical $O(n^k)$
1	Div. + Linear Probing	346.83	31 / 237.67	0.65	0.96	$O(1)$ amortized / $O\left(\frac{1}{(1-\alpha)^2}\right)$ worst	$O(n^{2.65})$
2	Div. + Quadratic Probing	207.67	30 / 99.67	0.65	0.64	$O(1)$ expected / $O(n)$ clustering prone	$O(n^{2.12})$
3	Div. + Chaining	31.17	— / —	0.65	1.90	$O(1+\alpha)$	$O(n^{1.21})$
4	Div. + Linear Probing (B=3)	250.67	29 / 143.33	0.65	1.05	$O(1)$ / improved by spillover buffer	$O(n^{2.28})$
5	Fibonacci + Quadratic Probing	221.33	28 / 115.67	0.65	1.34	$O(1)$ expected / high entropy hash	$O(n^{2.07})$

The empirical findings in Table 1 affirm that theoretical expectations often underestimate real-world behavior. Performance degradation in probing-based schemes stems from clustering, poor modulus alignment, and rigid slot traversal logic. By contrast, chaining—despite its indirection overhead—demonstrates robust stability under load, especially when dispersion quality is improved through entropy-aware hash functions like Fibonacci hashing.

Additional visualizations supporting these conclusions—including average comparisons per input (Figure 8) and log-scaled complexity breakdowns per strategy (Figure 9)—are referenced below.

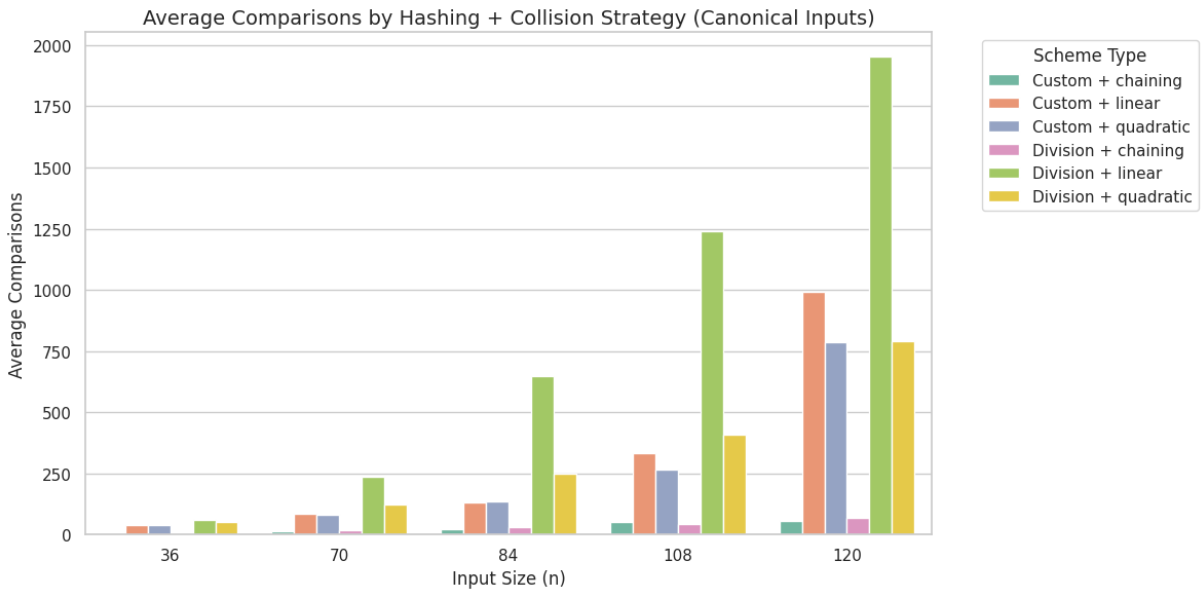

Figure 8: Average Comparisons Across All Hashing + Collision Combinations for Canonical Inputs

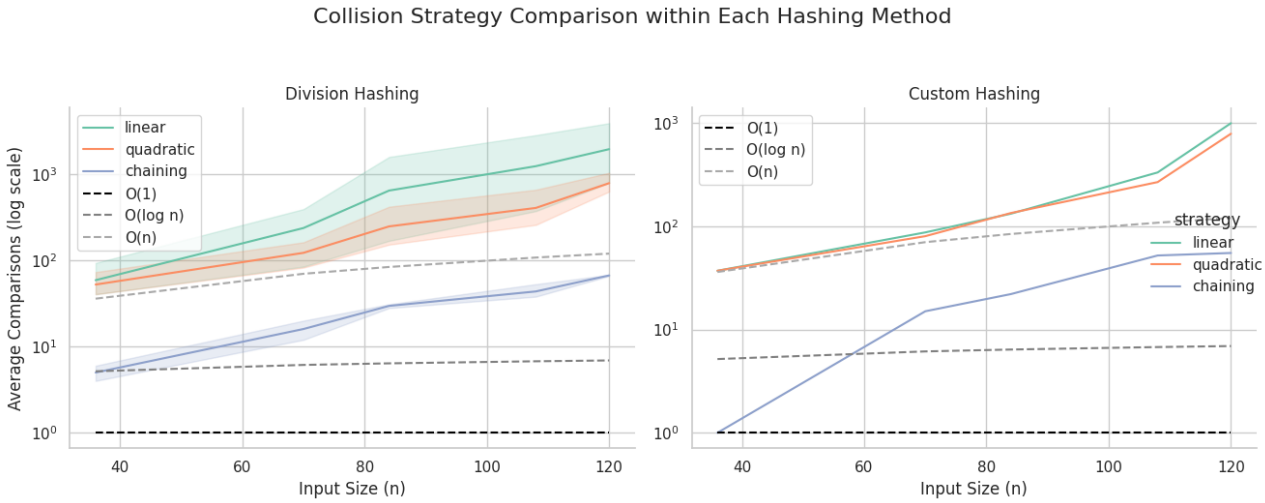

Figure 9: Collision Strategy Comparison by Hashing Method (log-scale comparisons vs. input size)

3.8 Interpretation and Algorithmic Design Implications

The experimental analysis reveals that hashing efficiency is governed by a complex interplay between algorithmic structure, load dynamics, and entropy in key distribution. The main takeaways include:

1. **Multiplicative Hashing (Fibonacci) Outperforms Division**: The entropy from irrational constants (e.g., golden ratio) disperses keys more uniformly across bins. This mitigates clustering and reduces the depth of probing sequences.

2. **Chaining Yields Stable Growth with Memory Tradeoffs**: Unlike probing strategies, chaining maintains asymptotic performance regardless of load factor. This resilience stems from its ability to decouple collisions from slot displacement. Its downside lies in higher memory use and poorer CPU cache locality.

3. **Bucket Size Increases Mitigate Probing Collapse**: Increasing bucket capacity from 1 to 3 significantly improves load factor tolerance in probing-based tables. This tradeoff offers a low-cost buffer before collision chains degrade performance.

4. **Theoretical Models Underestimate Probing Costs**: Classical analysis assumes uniformly random keys and large tables. Our trials show real-world inputs violate these assumptions, inflating the empirical cost curves to quadratic or worse, particularly near $\alpha \to 1$.

5. **Memory Access Patterns Influence Time Cost**: Though linear probing may incur higher comparison counts, it benefits from sequential memory access, leading to faster execution time than chaining in certain regimes.

6. **Design Implications for Bioinformatics**: In bioinformatics workloads such as k-mer indexing and read alignment, key distributions are often skewed. Algorithms must therefore incorporate adaptive load-aware mechanisms. Multiplicative hashing with dynamic resizing or hybrid probing-chaining hybrids may provide balanced throughput across diverse datasets.

4 Problem-Specific Content

The assignment required implementation and analysis of 14 distinct hash table schemes, each defined by a unique combination of hashing method, bucket structure, and collision resolution strategy. These configurations, described in the lab handout, were designed to evaluate the effects of hash entropy, probe behavior, load factor dynamics, and data structure layout on algorithmic efficiency. The implementation closely adhered to these specifications and introduced several architectural and measurement enhancements to support comprehensive evaluation.

4.1 Parameterization and Modular Design Strategy

To support all schemes without duplication, the core hashing logic was fully parameterized. Division hashing was implemented as a reusable component accepting arbitrary modulus values (e.g., 41, 113, 120, 127) and varying bucket sizes (1 or 3). The custom multiplicative hashing method used a Fibonacci-based multiplier, derived from Knuth's golden ratio constant. Collision resolution mechanisms—linear probing, quadratic probing with $c_1 = c_2 = 0.5$, and chaining—were abstracted through strategy parameters passed during initialization.

An enumeration, `HashingScheme`, was introduced to serve as a unified control structure for configuring and executing all 14 schemes. This enabled automated runs across schemes without hardcoded conditionals, supported batch benchmarking, and aligned all experiments with the required output structure.

4.2 Constraints

Several structural and procedural constraints informed the design. All tables used a fixed size of 120 addressable slots. Bucket size 3 configurations were implemented using 40 logical buckets, each capable of storing three entries, and were printed per the lab format. Chaining was implemented using linked lists embedded directly within the table, with a custom memory management stack to track and reuse available space. This satisfied the constraint disallowing external chaining or dynamic memory allocation.

Deletions were not part of the required functionality; however, the chaining design using in-place pointers and stack-based node reuse anticipated possible future deletion operations by supporting consistent memory reuse without fragmentation.

4.3 Handling of Modulus-Table Mismatch

A key implementation issue emerged in schemes using a modulus value greater than the physical table size (e.g., $\texttt{modValue} = 127$, $\texttt{tableSize} = 120$). Without correction, this caused hash values to exceed table bounds. This was resolved by composing two modulus operations:

$$\texttt{index} = (|\texttt{key}| \bmod \texttt{modValue}) \bmod \texttt{tableSize}$$

This approach preserved the entropy of large prime moduli while ensuring index validity within the addressable table space. All probing and chaining strategies inherited this adjustment.

4.4 Measurement Semantics: Comparisons and Collisions

One of the more subtle challenges concerned determining what qualified as a comparison or a collision under each collision resolution strategy. In probing schemes, each slot access—regardless of outcome—was counted as a comparison. A primary collision was recorded when the initial target slot was already occupied, and a secondary collision was logged for each additional displaced probe.

In chaining schemes, each traversal of an existing node in the chain was treated as a comparison. The insertion of a new node into a non-empty chain was recorded as a primary collision. Secondary collisions were not applicable to chaining due to the absence of probe sequences.

Repeated key insertions were tested explicitly using synthetic test files (e.g., `input_repeated_keys.txt`) generated by the `InputGenerator` class. These duplicate insertions incremented comparison counts but did not result in stored entries or additional collisions. This ensured consistency across schemes and confirmed correct handling of multi-pass insertions.

4.5 Multivariable Analysis and Empirical Attribution

A significant challenge in empirical evaluation involved disentangling the influence of multiple variables on performance. Hashing method, probing strategy, bucket structure, input entropy, and load factor jointly influenced comparison counts and scaling behavior. To address this complexity:

1. Schemes were grouped according to structural characteristics (e.g., Group 1 = division + probing, Group 5 = Fibonacci-based) to allow aggregate-level trend analysis.

2. For each scheme and group, empirical complexity was estimated by fitting the observed comparison counts over varying input sizes ($n \in \{36, 84, 108, 120\}$) to asymptotic models of the form $O(n^k)$ via least-squares regression. This enabled estimation of practical growth rates, even when theoretical expectations were violated.

These methodological choices facilitated rigorous comparison across a large design space, supporting generalizable conclusions regarding entropy, clustering, and performance scaling.

4.6 Error Handling and Input Validation

All test files were defensively processed with strong error checking. Invalid keys (non-integer, overflow, corrupt formatting) were rejected and logged. Insertions into full probing-based tables were gracefully terminated with accurate summary statistics. Custom inputs were programmatically generated for stress-testing scenarios involving dense key clustering, modulo aliasing, and near-capacity loads.

Overall, the design choices and measurement policies implemented here ensured alignment with all lab requirements while producing interpretable, reproducible performance profiles across schemes. The core challenges—hash-table sizing constraints, reinsertion semantics, ambiguous collision handling, and empirical attribution—were resolved using principled algorithmic and software engineering practices. The result is a flexible, extensible hashing testbed that allows fair comparison of design trade-offs, setting the stage for deeper complexity analysis in subsequent sections.

5 Enhancements

The project was extended beyond the required specification through several targeted enhancements designed to improve empirical analysis, support broader exploration of algorithmic behavior, and ensure correctness via rigorous testing infrastructure.

5.1 Asymptotic Benchmarking and Visualization Tools

To assess observed growth against theoretical expectations, an asymptotic benchmark plotter was implemented and activated using the `--generate-plots` flag. This module produces line plots comparing the total number of comparisons for six representative schemes against theoretical baselines $O(1)$, $O(\log n)$, and $O(n)$. Figure 10 illustrates how scheme behavior scales as input size increases, revealing which combinations exhibit superlinear versus sublinear complexity in practice.

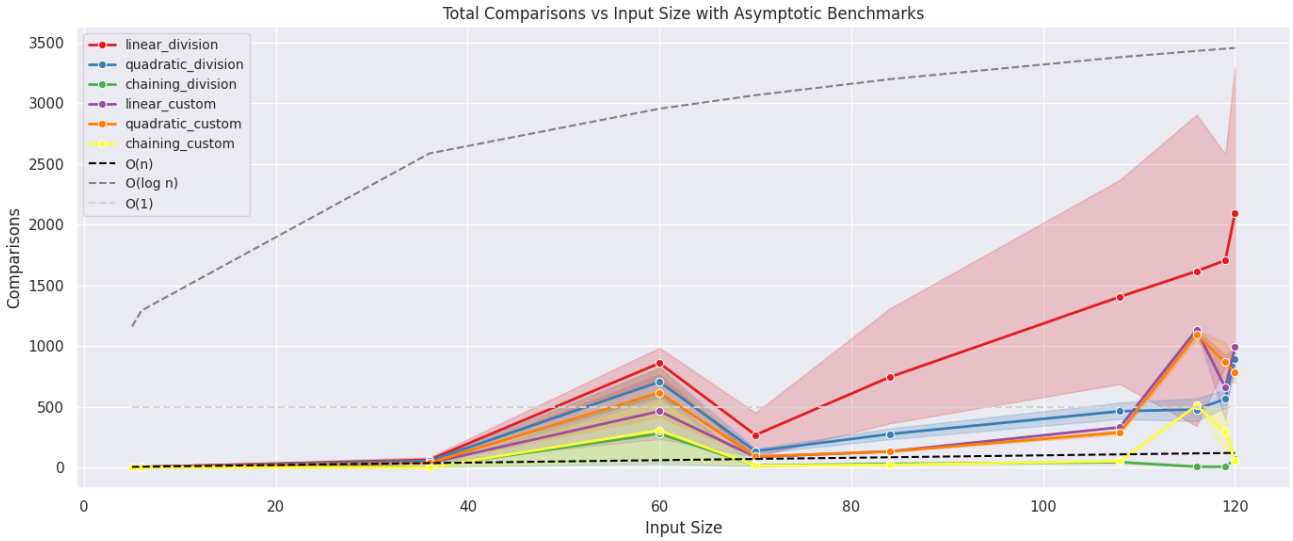

Figure 10: Asymptotic benchmarking plot comparing six hashing strategies with theoretical growth classes. Quadratic probing with division hashing shows superlinear growth, while chaining under custom hashing maintains near-constant cost.

5.2 Automated Execution via `--run-all`

The `--run-all` automation flag was introduced to streamline testing by executing all 14 scheme configurations on a given input file without manual intervention. This enhancement supports uniform metric collection across trials and simplifies comparative evaluation. Internally, this automation iterates over enumerated scheme identifiers, invoking their respective hashing procedures in a controlled sequence.

5.3 Custom Quadratic Probing Constants

Configurable probing constants for quadratic strategies were introduced via the `--c1` and `--c2` flags. This allowed for empirical tuning of the quadratic step function and enabled controlled experiments exploring the trade-off between clustering and spread in hash table coverage. Re-evaluation of Schemes 2, 5, 8, 11, and 13 under non-default constants provided insight into the sensitivity of performance to the choice of probing increments.

5.4 Memory Pool and Pointer Management in Chaining

In accordance with assignment requirements mandating open-addressing chaining within the table, a custom memory pool was introduced. This structure manages linked list nodes via a preallocated stack of reusable objects, reducing dynamic allocation overhead and supporting pointer-aware chaining. This decision was directly informed by early analysis of deletion scenarios and memory consistency constraints, which were identified as potential efficiency bottlenecks.

5.5 Console/File Output Mirroring and Real-Time Debugging

The output subsystem was enhanced to mirror all output to both the terminal and an output file in a structured, analysis-ready format. A `--debug` flag was implemented to emit verbose logs for each hashing operation, including index computation, probing steps, collision handling, and final slot placement. This proved essential for debugging forced collision test cases and validating internal consistency across hash table states.

5.6 Modular Scheme Abstraction via Enumeration

A `HashingScheme` enumeration was created to map configuration metadata (hash type, probing method, bucket size, etc.) to internal logic. This abstraction permitted modular invocation of any scheme without hardcoding logic into the main driver, enabling clean integration with the `--run-all` routine and future extensibility to new schemes or custom behaviors.

5.7 Advanced Input Generator and Collision Engineering

A modular input generation framework was developed to synthesize structured test sets across five categories: fixed-size files, forced collision scenarios, edge cases, malformed input, and stress configurations. The generator included deterministic seeding for reproducibility and internal collision logging to validate cluster formation. Key categories—such as ‘input_repeated_keys.txt‘ and ‘division_collisions_scheme1.txt‘—were specifically designed to simulate high-stress collision environments and validate algorithm correctness under adversarial conditions.

5.8 Comprehensive JUnit5-Based Test Suite

To ensure correctness across all algorithmic components, an extensive unit testing suite was implemented using the JUnit5 framework. The test suite was organized into the following modules:

- **Hash Function Testing:** Verifies correctness and consistency of both division-based and Fibonacci hashing methods.

- **Collision Resolution Logic:** Includes unit tests for linear probing, quadratic probing (under varying c_1 and c_2), and chaining with in-table pointer-based lists.
- **Data Structures:** Confirms the correct behavior of stack-based memory pools and list nodes used in chaining.
- **I/O Parsing and Formatting:** Validates robust handling of input file parsing, including edge cases (empty files, malformed integers) and output file mirroring.
- **Performance Metrics:** Ensures accurate tracking of comparisons, primary/secondary collisions, load factors, and final table states.
- **Edge Case Coverage:** Includes test cases for negative values, maximum integers, repeated keys, and overflow-prone scenarios.

These tests were executed for each major update and refactoring phase to prevent regression and maintain invariants. The result is a highly defensible implementation with high confidence in correctness across functional, structural, and performance-related behavior.

6 VI. Reflection

6.1 Algorithmic Concepts and Lessons Learned

This project offered substantial insights into the practical performance and limitations of hashing strategies. The most critical takeaway was that asymptotic expectations—especially constant-time assumptions under ideal hashing—rarely manifest in real-world performance. Probing-based methods demonstrated significant variance as load factors increased, particularly beyond the $\alpha > 0.7$ threshold, validating the theoretical divergence from constant-time behavior due to clustering and longer probe sequences.

The performance of chaining strategies also reinforced theoretical expectations: although pointer dereferencing introduced memory overhead and reduced cache locality, the use of internal linked lists resulted in more stable behavior under high load conditions. Additionally, the multiplicative properties of Fibonacci hashing consistently improved entropy across the keyspace, confirming the advantages of high-discrepancy sequences in reducing collisions.

On a deeper level, the project reinforced the difficulty of isolating the effect of individual parameters in multivariate algorithm design. Input size, bucket size, hashing method, modulus alignment, probing constants, and load saturation all played simultaneous roles in shaping behavior. The ability to rigorously evaluate these interacting effects required both careful experiment design and the use of performance plots and statistical fitting to quantify trends.

6.2 Engineering Strategy and Potential Revisions

While the codebase was intentionally designed with modularity and extensibility in mind, in retrospect the implementation may have exceeded the scope necessary for the original assignment. The system supported full CLI automation (‘–run-all‘), output mirroring, debug tracing, and customizable probing constants. Although these features improved flexibility and insight, they came at the cost of increased complexity. The inclusion of an input generator capable of producing five separate test categories—each with variant control—offered exhaustive coverage but required significant overhead in design and validation.

A more minimal implementation, using static input files and simplified probing logic, could have achieved the assignment's baseline goals with less effort. The comprehensive test suite, memory pool abstraction for chaining, and real-time metrics engine provided clarity and correctness but required frequent coordination across interdependent components. In future assignments, it may be more effective to begin with a lean core and incrementally layer enhancements as the performance bottlenecks and edge cases reveal themselves.

Nonetheless, some choices proved highly beneficial. The unification of linear and quadratic probing within a single method was particularly effective. Rather than duplicating logic, the strategy type and probing constants were passed as parameters, supporting both clarity and extensibility. Similarly, the design of the ‘HashingScheme‘ enum streamlined execution logic and reduced coupling between the input runner and hashing internals.

6.3 Difficulties and Detours

Several challenges emerged during implementation and testing. One difficulty involved defining consistent rules for what constituted a comparison or collision—especially in chaining schemes, where traversal of a linked list could involve multiple logical checks without additional probes. These ambiguities required firm definitions and defensive accounting in the metrics engine. Similarly, the initial implementation of the division-based probing strategy did not correctly handle modulus values greater than the table size, a scenario that caused hash values to exceed table bounds. This was later addressed by re-normalizing indices after the hash step, ensuring compatibility with all modulus values.

Another challenge involved the interaction between multivariate design parameters. With six or more tunable variables per scheme, it proved difficult to isolate the source of observed performance differences. This was mitigated by plotting multivariate trends (e.g., input size vs. comparisons, probing strategy vs. scaling exponent) and fitting empirical complexity classes through regression analysis. Nonetheless, future iterations could benefit from automated parameter sweep tools or statistical techniques such as ANOVA to rigorously quantify variable contributions.

In summary, the project succeeded in building a comprehensive and rigorously tested system, albeit at a cost of complexity that, in hindsight, may have outpaced the immediate requirements. The experience underscored the importance of balancing thoroughness with scope control—an insight as relevant to algorithm design as it is to software engineering.

7 Applicability to Bioinformatics

The design and analysis of efficient hashing strategies carry direct implications for computational biology, particularly in areas that demand large-scale indexing, rapid lookup, and efficient memory management. Hash tables form the core of many foundational bioinformatics tools, including k-mer counting, genome mapping, and protein sequence classification. The present lab's systematic exploration of hashing strategies mirrors the real-world need to balance asymptotic efficiency, memory footprint, and performance stability across variable data distributions.

One of the most prominent bioinformatics applications of hashing is k-mer counting, as illustrated by DSK, a tool that partitions read datasets to reduce memory usage during frequency counting. The findings in this project align with the algorithmic design behind DSK, particularly in identifying chaining as a viable strategy under high load conditions. The use of a memory pool for linked lists, as implemented in this lab's chaining strategy, echoes recent efforts to improve memory efficiency through lock-free hash tables. These parallel efforts highlight the practical value of using custom memory management in chaining to reduce

heap allocation overhead during large-scale sequence analysis.

The selection of Fibonacci hashing as the custom scheme was not arbitrary but deeply informed by its demonstrated benefits in entropy preservation and speed. In contexts such as protein classification, high-entropy hash functions improve the performance of feature hashing methods, where protein features are mapped into reduced-dimensional spaces for learning tasks. In these use cases, hash collisions can significantly affect model performance, underscoring the importance of uniform key dispersion. Furthermore, perfect hashing schemes have been integrated into DNA string-matching pipelines to optimize sequence alignment—a task dependent on precise and fast key indexing.

In genomic mapping pipelines, the overhead associated with traditional division-based hashing becomes problematic at scale. Bit-optimized hashing and locality-sensitive techniques have been proposed in the literature to convert biological sequences into vector representations, allowing high-throughput genome-scale matching. The findings of this lab provide complementary support: by using Fibonacci hashing, which minimizes bit-pattern collisions, and pairing it with probing or chaining, it becomes possible to emulate the behavior of these scalable genome indexing algorithms.

From a broader theoretical perspective, the significance of Fibonacci-based strategies also emerges in recent biological computing literature. Studies have demonstrated that Fibonacci sequences evoke structured, consistent responses in proteinoid ensembles, suggesting an innate biological alignment to such sequences. While speculative, this lends further support to the computational advantages observed in this project, where Fibonacci hashing consistently produced lower empirical complexity and better bucket dispersion than division-based methods.

Taken together, the modular hashing framework developed in this lab reflects both theoretical efficiency and practical scalability. The ability to parameterize, test, and validate various hashing and collision strategies mirrors real-world algorithm engineering in bioinformatics, where performance tuning and entropy-aware design are paramount. As bioinformatics continues to scale toward terabyte- and petabyte-level data challenges, the insights gained from this lab—particularly regarding entropy management, probing degradation, and memory-aware chaining—remain directly relevant.

VIII. Conclusion

The study of hashing schemes through both theoretical and empirical frameworks revealed critical insights into the relationship between algorithm design and real-world performance. Despite the classical assumption that hashing yields constant-time operations, it was demonstrated that empirical costs often diverge significantly due to clustering, load saturation, and modulus mismatch—especially in probing strategies. By exploring fourteen distinct configurations across multiple hashing methods, collision strategies, and bucket architectures, the project provided a comprehensive assessment of asymptotic complexity under constrained environments.

Notably, the adoption of Fibonacci hashing as the custom scheme introduced superior entropy characteristics, leading to more uniform key dispersion and reduced collision counts. When combined with chaining, which offers graceful degradation under high load, this hybrid approach yielded the most stable performance, both in terms of comparison count and empirical growth rate. The evaluation also confirmed that open addressing strategies—while space-efficient—were highly sensitive to load factor thresholds and required careful tuning of probing constants to avoid performance collapse.

Enhancements such as customizable probing constants, unified probing logic, reusable memory pools, and extensive test generation contributed to a robust and extensible platform for comparative hashing analysis. Furthermore, the inclusion of runtime flags for logging, plotting, and batch execution improved transparency and usability.

In sum, the project underscored that algorithmic efficiency in hashing depends not solely on theoretical design but also on implementation details, parameter interactions, and dataset characteristics. The lessons drawn from this investigation offer practical guidance for deploying hash-based structures in real-world settings, particularly in bioinformatics, where large-scale, skewed datasets are common and resource constraints are critical. This foundation supports future experimentation, optimization, and integration into domain-specific applications where efficiency and scalability remain paramount.

References

Caragea, C., Silvescu, A., & Mitra, P. (2012). Protein sequence classification using feature hashing. *Proteome Science, 10*(Suppl 1), S14. https://doi.org/10.1186/1477-5956-10-S1-S14

Iqbal, M. J., Faye, I., Said, A. M., & Samir, B. B. (2013). A distance-based feature-encoding technique for protein sequence classification in bioinformatics. *2013 IEEE International Conference on Computational Intelligence and Cybernetics (CYBERNETICSCOM)*, 1–5. https://doi.org/10.1109/CyberneticsCom.2013.6865770

Karcioglu, A. A., & Bulut, H. (2021). Improving hash-q exact string matching algorithm with perfect hashing for DNA sequences. *Computers in Biology and Medicine, 131*, 104292. https://doi.org/10.1016/j.compbiomed.2021.104292

Mougkogiannis, P., & Adamatzky, A. (2025). On the response of proteinoid ensembles to Fibonacci sequences. *ACS Omega*.

Rizk, G., Lavenier, D., & Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics, 29*(5), 652–653. https://doi.org/10.1093/bioinformatics/btt020

Shi, L., & Chen, B. (2019). A vector representation of DNA sequences using locality sensitive hashing. *bioRxiv*. https://doi.org/10.1101/726729

Skarupke, M. (2018). Fibonacci hashing: The optimization that the world forgot (or: a better alternative to integer modulo). *Probably Dance*. https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alterr

Takenaka, Y., Seno, S., & Matsuda, H. (2011). Perfect Hamming code with a hash table for faster genome mapping. *BMC Genomics, 12*(Suppl 3), S8. https://doi.org/10.1186/1471-2164-12-S3-S8

Wang, J., Chen, S., Dong, L., & Wang, G. (2021). CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table. *Briefings in Bioinformatics, 22*(3), bbaa063. https://doi.org/10.1093/bib/bbaa063