Frequency Count Sort: A Near-Linear Time Distribution Sort for Low-Entropy Datasets with Frequent Duplicates

Ogunbor Anthony Aikhomu

[1] *Department of Data Science, University of Applied Sciences, Bad Honnef, Germany*

Corresponding author email: aikhomu-anthony.ogunbor@iu-study.org

Abstract

Many real-world datasets exhibit structural low entropy: a small number of unique values u repeated across a large input of size n, where u is far less than n ($u << n$). Classic examples include election results, student grades, national exam, genomic sequences, sensor telemetry, and survey responses. Despite the prevalence of this data profile, very few sorting algorithms in common use, e.g., the 3-way Quicksort, have been designed explicitly for the dataset domain represented in this paper as typically $u <= 100$ and $n >= 100,000$. This paper introduces Frequency Count Sort (FC-Sort), a distribution-based sorting algorithm targeting low-entropy, high-duplicate datasets. FC-Sort operates in three stages: (1) a single O(n) frequency count, building a map of u unique keys to occurrence counts; (2) a sort of unique keys alone, using insertion sort and completing in at most $u(\underline{u}-1)/2$ comparisons in the worst case - a fixed constant independent of n; and (3) a linear O(n) left-to-right reconstruction with a total complexity of $O(n + u^2)$. Empirical benchmark using optimized FC sort indicates that the algorithm consistently outperforms one of the two best performing algorithms in the problem space defined - Radix sort, while remaining competitive with the other - Counting sort and showing consistent superiority for smaller values of u. The algorithm outperforms all O($n$log$n$) algorithms by a factor of 2x to over one order of magnitude across the dataset space tested. The exception being 3-way quicksort which at very small u ($u<=5$), can marginally outperform FC sort.

Keywords:

Fc-sort; homogeneity test heuristic (HTH); low-entropy datasets; distribution sorting; empirical algorithm analysis; adaptive sorting algorithms

I. INTRODUCTION

It is considered that in a given situation where the number of possible key values far exceeds the number of elements to be sorted, the probability that equal keys are present will be very small [1]. One can infer from that assertion that when the number of possible key values (i.e., range of key values) fall far below the number of keys (count/number of elements) to be sorted, the probability that equal keys are present will be very high. For example, where the possible discrete integer elements to be sorted ranges from 1-3, and the number n of elements (keys) to be sorted is 5, you certainly will have duplicates no matter which order they appear. For example, consider the list A = {1, 2, 3, 1, 2}, A= {1, 2, 3, 3, 3}, or A= {3, 1, 1, 2, 1} and so on, where range of values is 1 - 3 and size n = 5. Practically, most real life activities we engage in and information processing we undertake produce a variety of data with unavoidable multiple duplicate values. Such include, a national election with 1,000,000 voters and 5 candidates, a product purchase survey of 3 products involving 50,000 participants, and the age distribution of freshmen students in a university of 60,000 registered students. The remainder of this paper is organized as follows: Section II reviews related work. Section III describes the proposed methodology. Section IV presents experimental results. Section V concludes.

II. LITERATURE REVIEW

In sorting datasets with frequent duplicates (equal keys) of discrete values in the dataset, say for example, an array of size 50,000 and above, non-comparison based sorting algorithms such as *Counting* sort and *Radix* sort usually outperform algorithms with O(nlogn) time complexity such as Quicksort, Timsort and Mergesort. These two have best, average and worst case linear time complexities of O(n+k) and O(d(n+k)) respectively.

A. Counting Sort

The counting sort algorithm, for example, assumes that the input numbers belong to the set {0, 1, … ,k}. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in O(k + n) time where k is the value range and n the number of items in the input dataset. Thus, when k = O(n), counting sort runs in time that is linear to the size of the input array [2]. Counting Sort exhibits optimal performance when the range of input values (denoted as k) is comparable to the number of elements (n). However, in scenarios where k greatly exceeds n—such as n = 1,000 and

k = 2,000,000—the algorithm incurs significant overhead, rendering it inefficient in both time and space complexity. Consequently, alternative sorting methods not limited by large integer range like the one demonstrated in this paper would be preferred in such contexts.

B. Radix Sort

Radix sort on the other hand, represents each item to be sorted as a *d*-digit number, where each digit takes on *k* possible values [2]. Here it is the digits or bits which are sorted position by position. While it avoids the limitation of very large integer range of counting sort, it has its own limitation - inefficiency (degrading runtime complexity) for very large values of *d* on datasets with frequent duplicate values. Radix has $O(d(n + k))$ run-time complexity (worst case and average case)[2], where k is the radix base – the number of possible digit values.

C. Partitioning - 3-Way Quicksort algorithm

The problem of sorting a mass of items occupying consecutive locations in the store of a computer, may be reduced to that of sorting two lesser segments of data. This is provided that it is known that the keys of each of the items held in locations lower than a certain dividing line are less than the keys of all items held in locations above this dividing line. This method of sorting is called partition [3]. The binary partition introduced in Quicksort algorithm by C. A. R Hoare [3] paid no special attention to the presence of duplicate keys in the input list. In the same vein, the analysis of Quicksort by Robert Sedgewick in [4] focused only on binary partitioning and not on handling equal (duplicate) keys.

Ternary partition for a classification problem was introduced by Edsger Dijkstra[5], in the Dutch National Flag problem in his 1976 book. The first formal application of ternary partitioning to Quicksort was by Robert Sedgewick in [1]. Subsequent refinements were carried out in [6] and then a formal proof of the optimality of the work of Bentley, J. L., & Sedgewick, R. [6] given by Sebastian Wild in [7]. While the binary partition introduced in Hoare's Quicksort [3] was more or less a brute-force divide and conquer partition strategy, other refinements as presented in [1],[6] provide a more intelligent and duplicate aware design. The ternary partitioning of the input list in these refinements is with purpose – to optimize the sorting process by efficiently processing equal (duplicate) keys. The algorithm introduced here deviates from this well studied partition strategy and introduces a frequency mapping strategy that effectively reduces the sort operation to a smaller sized unique key sub-list of the original unsorted list, reconstructing the sorted space by expanding the sorted list of unique keys using the frequency count of each key in the sub-list

D. Problem statement and decomposition

The first problem that we face in trying to analyze any sorting method with equal keys (duplicates) is the formulation of an appropriate model describing the input file[1]. This study proposes specific formalisms tailored to the datasets under study, rather than a fully generalized model. These formalisms are presented below.

1) Duplicate element: For any set of elements of a finite dataset contained in a linear data structure, say a list A, or an array A, there exists at least one pair of indices (i, j) such that:

1. $n >= 2$
2. $0 \leq i < j \leq n-1$
3. $A[i] = A[j]$

By this definition, a duplicate can only occur if and only if; First, the number of elements in the List *n*, (i.e., List size = *n*), is at least two elements, that is, n>=2. Second, for the pair of indices (*i*, *j*) contained in *n*, the index *i* must always be less than the index *j* expressed in the inequality as (i < j). This means that the List position A[i], with its value, must have been encountered (at least once), prior to its value being repeated in another List position A[j]. Third, the values of the different List positions must be equal. For example, consider a List, A = [7, 2, 45, 3, 51, 2, 7, 8, 9, 11, 10], the value 2 referenced as A[i], is duplicated in array index position 5, (referenced as A[j]), having appeared first in index position 1.

2) Duplicates of a single element: We have duplicates of a single discrete element in a dataset if there exists $k >= 2$ indices in a List B, where $B[i_1] = B[i_2] = . . . = B[i_k]$, *k* being the number of times an element appears in the List. For examples, a list B = {8, 3, 6, 3, 16, 3, 5, 3, 15}, where the element 3 appears multiple times ($k = 4$) while other elements have a unique (single) appearance in the List.

3) Multiple duplicates of multiple discrete elements: Consider the list

C = {5, 2, 6, 2, 6, 1, 5, 1, 5, 1, 1, 1, 5, 1, 2, 2, 5, 2, 1, 2, 2, 6, 6, 6, 1, 5, 5, 6, 1, 6 } with multiple discrete elements having duplicates. The elements in list positions C[0] = C[6] = C[8] = C[12] = C[16] = C[25] = C[26], all with value 5. Therefore the unique element 5, appears seven times with count = 7. Other unique elements are 2, 6 and 1 which all have counts 7, 7 and 9 respectively. In this case, the list C with size $n = 30$ elements has only four unique elements $u = 4$ (that is, 5, 2, 6 and 1) which is relatively few compared to the list size. Such a dataset theoretically guarantees the existence of multiple duplicates. This paper focuses on datasets of the nature of list C described above.

III. DESIGN METHODOLOGY

The algorithm presented exploits the high presence of duplicates in low-entropy datasets, where duplicates are expected to be a significant factor, by determining the frequency count of each element in the input list. It constructs a frequency map of the input dataset, resulting in a set of only unique keys with the count of each unique key in the input list.

It then sorts the unique list using Insertion sort. This unique list is guaranteed to be much smaller than the original dataset size n, given nature of such dataset domain - low entropy. Thereafter, each key in the set is expanded to the exact number of appearances in the original list using the related count established at the beginning. This is achieved by filling the target array with keys (expanded by their frequency counts) in the sorted set of unique keys from left to right. Insertion sort is chosen as the default sorting algorithm for the list of unique elements because it is typically efficient and very competitive for sorting small datasets of size ≤ 100, which fits the dataset domain of focus in this study. This design methodology is demonstrated in the algorithm stated in the next section as well visually in the Figure 1 that follows.

The list data structure is used in the algorithm design and subsequently in the code implementation in the Python programming language. List is chosen over arrays primarily because of its dynamic size allocation (available by default) and simplicity. The mechanical (un-optimized) version of the algorithm uses a linear scan of the unique keys per element in the input list (i.e., list to be sorted). This leads to a time complexity equal to O(n x u). Whereas the optimized implementation uses a hash map with O(1) average lookup — O(n) total. This confirms that the O(n) complexity of FC-Sort is dependent on O(1) key lookup, which is guaranteed by any hash-based associative structure. In languages with native hash map support such as we have in Python, Java, and C++, this O(n) complexity is automatically satisfied.

A. Algorithm

```
Algorithm: FC sort(S)
Input:  S - a sequence of n elements
Output: Sorted sequence of S

1.  // Phase 1: Build frequency dictionary
2.  freq = empty dictionary
3.  for each x in S do
4.      if x in freq then
5.          freq[x] = freq[x] + 1
6.      else
7.          freq[x] = 1
8.      end if
9.  end for
10. //Case for all identical keys
11. if len(freq) == 1
        return S //list requires no sorting as all elements (keys)
                 //are identical. Mission accomplished.

12. // Phase 2: Extract and sort unique keys
11. unique_keys = list of keys from freq
12. insertionSort(unique_keys)

13. // Phase 3: Reconstruct sorted output
14. result = new array of length n
15. pos = 0
16. for each val in unique_keys do
17.     count = freq[val]
18.     for i = 1 to count do
19.         result[pos] = val
20.         pos = pos + 1
21.     end for
22. end for
23. return result
```

B. Algorithm flow (summary)

1. *Build Frequency Map:*

   Scan input list → Count occurrences of each unique element

   Store in dictionary: element → frequency

2. *Check for Single unique Key (Early Exit):*

   If only one unique key exists → All elements are identical

   Output is simply the input (already sorted) → sorting complete

   If multiple unique keys exist → Proceed to next step

3. *Extract and Sort Unique Keys:*

   Create list of unique keys from dictionary

   Sort this list (using insertion sort) in ascending order

4. *Reconstruct Output:*

   Create empty output list of length n

   For each sorted key in order:

   Write key to output array (frequency count times)

   Fills output from left to right sequentially

5. *Termination:*

When all keys processed, output array is fully populated with sorted elements
Return output array

C. Visualization of Algorithm

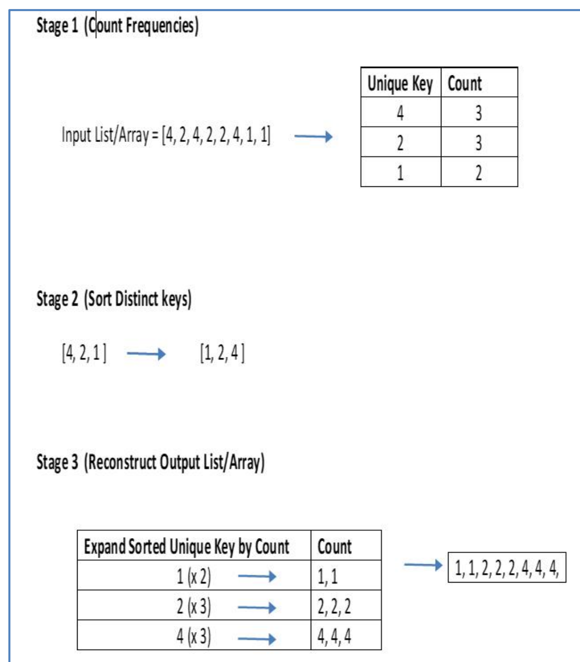

Fig. 1 FC-Sort (Frequency Count Sort) Algorithm visualized

C. Experimental setup

All benchmark experiments were conducted on a Windows 10, Intel64 Family 6, Model 140 platform running Python 3.9.13. Each algorithm was executed ten times per configuration on a freshly generated dataset, and the mean execution time (in milliseconds) across ten runs was recorded as the primary performance metric. Standard deviation is reported alongside the mean to show the stability of the measurement. All competitor algorithms were implemented in their true canonical forms, operating directly on all n elements. FC-Sort is the sole algorithm employing the frequency map mechanism. Dataset reproducibility was ensured using a fixed random seed across all configurations.

IV. RESULTS

Two implementation variants of each algorithm were evaluated. The optimized variant employs Python's native data structures — most critically, the built-in dictionary for O(1) average-case key lookup in FC-Sort. The mechanical variant eliminates all built-in language-level facilities, using explicit while-loops and index arithmetic throughout. Both variants are reported, as the mechanical form exposes the raw algorithmic behaviour independent of interpreter- level optimizations.

A. Performance on the Core Domain: $u <= 100$, $n >= 100{,}000$

Table I presents optimized algorithm performance at n = 100,000 across a systematic sweep of unique key counts from u = 5 to u = 100. This configuration directly targets the paper's defined problem domain. FC-Sort ranks first at every configuration from u = 5 through u = 80, and remains within 1ms of Counting Sort at u = 90 (FC-Sort: 12.48ms, Counting Sort: 12.15ms) and u = 100 (FC-Sort: 15.68ms, Counting Sort: 13.98ms). These margins are well within measurement variance and can be considered statistically equivalent at u >= 90. The separation from all other algorithms is consistent and substantial. At u = 5, FC-Sort (11.17ms) and Counting Sort (11.98ms) form a clear first-level, while 3-way Quicksort (25.52ms) and Radix Sort (26.64ms) are 2.3x slower, and Timsort (191.38ms) is 17x slower. This two-level structure, that is, FC-Sort and Counting Sort is a fast distribution-based class with all other algorithms compared well behind and it is stable across the full u range tested. At u = 80, FC-Sort (12.70ms) and Counting Sort (16.48ms) remain together in the first level while 3-way Quicksort has grown to 70.15ms (5.5x slower than FC-Sort) and Timsort to 213.20ms which is 16.8x slower than FC-Sort's 12.70ms. A key structural observation is that FC-Sort's execution time is largely insensitive to u within the range u = 5 to u = 100 at n = 100,000. Times range from 11.17ms (u=5) to 15.68ms (u=100), a variation of only 4.5ms over a 20-fold increase in u. This confirms the theoretical claim that, for fixed n and small u, the sort phase cost $O(u^2)$ contributes negligibly to total execution time.

TABLE I
EXECUTION TIMES ON THE CORE DOMAIN OF FC-SORT
(u <= 100, n >= 100,000)

n = 100,000 , u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	11.17ms	0.55ms	1
Counting Sort	11.98ms	0.62ms	2
3-way Quicksort	25.52ms	1.24ms	3
Radix Sort	26.64ms	1.16ms	4
Timsort	191.38ms	8.26ms	5
n = 100,000 , u = 80			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	12.70ms	2.22ms	1
Counting Sort	16.48ms	2.25ms	2
Radix Sort	54.46ms	12.33ms	3
3-way Quicksort	70.15ms	10.59ms	4
Timsort	213.20ms	17.68ms	5
n = 100,000 , u = 90			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	12.15ms	1.07ms	1
FC- Sort	12.48ms	2.83ms	2
Radix Sort	47.70ms	2.81ms	3
3-way Quicksort	79.09ms	9.93ms	4
Timsort	317.63ms	68.23ms	5
n = 100,000, u = 100			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	13.98ms	2.52ms	1
FC - Sort	15.68ms	5.44ms	2
Radix Sort	69.22ms	7.56ms	3
3-way Quicksort	69.31ms	2.71ms	4
Timsort	227.18ms	23.95ms	5

B. Linear Scaling with n at low u

Table II presents FC-Sort's performance as n scales from 100,000 to 2,500,000 at small fixed u values (u = 5, 6, 10). FC-Sort maintains its #1 ranking across all n values at u = 5 and u = 6, and alternates the #1 position closely with Counting Sort at u = 10. At u = 5: n = 100,000 executes in 11.17ms; n = 500,000 in 60.11ms; n = 800,000 in 92.98ms; n = 2,000,000 in 253.45ms. The ratio of times follows near-linear scaling with n, consistent with the O(n) complexity claim. Doubling n from 500,000 to 1,000,000 approximately doubles execution time, confirming the expected behaviour.

The performance gap between FC-Sort and Timsort widens substantially with n. At n = 100,000, u = 5, Timsort is 17x slower; at n = 2,000,000, u = 5, Timsort (11,569ms) is 45x slower than FC-Sort (253ms). This widening gap reflects Timsort's O(n log n) cost growing faster than FC-Sort's O(n) cost as n increases, and this is the exact theoretical prediction for the low-entropy domain.

TABLE II
EXECUTION TIMES: FC-SORT SHOWING LINEAR SCALING WITH n AT LOW u

n = 100,000 , u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	11.17ms	0.55ms	1
Counting Sort	11.98ms	0.62ms	2
3-way Quicksort	25.52ms	1.24ms	3
Radix Sort	26.64ms	1.16ms	4
Timsort	191.38ms	8.26ms	5
n = 500,000 , u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	60.11ms	4.62ms	1
Counting Sort	72.01ms	9.43ms	2
3-way Quicksort	147.70ms	16.37ms	3
Radix Sort	158.02ms	15.29ms	4
Timsort	1166.66ms	76.39ms	5
n = 2,000,000 , u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	253.45ms	19.66ms	1
Counting Sort	261.26ms	6.86ms	2
Radix Sort	693.38ms	102.11ms	3
3-way Quicksort	838.13ms	119.78ms	4
Timsort	11569.73ms	3388.02ms	5
n = 100,000 , u = 10			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	28.91ms	3.08ms	1
FC- Sort	36.77ms	4.92ms	2
3-way Quicksort	100.37ms	57.16ms	3
Radix Sort	106.68ms	14.27ms	4
Timsort	415.91ms	52.17ms	5
n = 800,000 , u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	92.98ms	7.32ms	1
Counting Sort	97.19ms	4.61ms	2
3-way Quicksort	207.56ms	14.16ms	3
Radix Sort	238.97ms	7.98ms	4
Timsort	1748.35ms	76.49ms	5
n = 2,500,000, u = 6			
Algorithm	**Mean**	**Std Deviation**	
FC- Sort	305.63ms	29.38ms	1
Counting Sort	386.13ms	62.87ms	2
3-way Quicksort	739.10ms	124.63ms	3
Radix Sort	1116.53ms	229.73ms	4
Timsort	6895.61m	772.19ms	5

C. FC-Sort vs Counting Sort: The Range Dependency Effect

The most important competitive relationship in this benchmark is between FC-Sort and Counting Sort, as both occupy the first performance level. Counting Sort's natural form allocates a range array of size r = max(input) − min(input) + 1. In the benchmark, keys are drawn from {1, ..., u}, so $r = u$, representing the best possible case for Counting Sort (dense, contiguous integer keys). Despite this advantage, FC-Sort is competitive with or superior to Counting Sort in the majority of configurations tested.

D. Competitive Boundary revealed on Table 3: (n = 2,000,000, u = 5 to u = 3200)

Table III clearly reveals the competitive boundary. FC-Sort leads at u = 5, 200, 400, 500, 700 and 900 - well beyond this paper's defined domain of u <= 100, while Counting Sort leads at u = 1600 and u = 3200. This finding has an important practical implication. The benchmark uses the most favourable possible configuration for Counting Sort: integer keys in a contiguous range where $r = u$. In real-world data, keys are often sparse in value space (e.g. postal codes, product SKUs, sensor fault codes), meaning r >> u. In such cases, Counting Sort incurs overhead proportional to r while FC-Sort

incurs overhead proportional to u alone. FC-Sort's practical advantage over Counting Sort in production environments is therefore likely to be greater than the benchmark results suggest in this study.

TABLE III
FC-SORT's COMPETITIVE BOUNDARY

$n = 2{,}000{,}000$, $u = 200$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	224.17ms	11.86ms	1
Counting Sort	249.16ms	31.48ms	2
Radix Sort	1615.08ms	286.76ms	3
3-way Quicksort	2513.15ms	616.03ms	4
Timsort	5564.02ms	221.82ms	5
$n = 2{,}000{,}000$, $u = 500$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	284.82ms	30.78ms	1
Counting Sort	402.49ms	56.54ms	2
Radix Sort	2615.47ms	122.99ms	3
3-way Quicksort	3793.75ms	535.81ms	4
Timsort	10676.15ms	3785.28ms	5
$n = 2{,}000{,}000$, $u = 900$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	308.59ms	35.77ms	1
Counting Sort	336.52ms	22.47ms	2
Radix Sort	4953.61ms	2037.23ms	3
3-way Quicksort	6381.24ms	3781.80ms	4
Timsort	10994.92ms	5640.47ms	5
$n = 2{,}000{,}000$, $u = 3{,}200$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	287.62ms	17.23ms	1
FC- Sort	457.89ms	18.75ms	2
Radix Sort	3037.75ms	35.24ms	3
3-way Quicksort	3327.98ms	668.48ms	4
Timsort	7978.23ms	3248.66ms	5

TABLE III (continued)
FC-SORT's COMPETITIVE BOUNDARY

$n = 2{,}000{,}000$, $u = 5$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	253.45ms	19.66ms	1
Counting Sort	261.26ms	6.86ms	2
Radix Sort	693.38ms	102.11ms	3
3-way Quicksort	838.13ms	119.78ms	4
Timsort	11569.73ms	3388.02ms	5
$n = 2{,}000{,}000$, $u = 400$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	242.81ms	20.39ms	1
Counting Sort	292.23ms	22.15ms	2
Radix Sort	1982.75ms	326.02ms	3
3-way Quicksort	2829.29ms	524.50ms	4
Timsort	6259.99ms	729.80ms	5
$n = 2{,}000{,}000$, $u = 700$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	278.57ms	27.73ms	1
Counting Sort	381.61ms	77.69ms	2
Radix Sort	2605.49ms	504.55ms	3
3-way Quicksort	2792.38ms	276.23ms	4
Timsort	6869.59ms	716.59ms	5
$n = 2{,}000{,}000$, $u = 1{,}600$			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	357.73ms	36.82ms	1
FC- Sort	480.19ms	47.56ms	2
Radix Sort	2947.99ms	83.13ms	3
3-way Quicksort	3171.51ms	554.70ms	4
Timsort	5656.67ms	278.54ms	5

E. The Case of all Identical Keys - The Homogeneity Test Heuristic (HTH) Feature of FC-Sort @ u =1

Table IV presents a unique dataset category of all identical keys in the dataset. Here FC-Sort leads in all eight dataset configurations for all different dataset sizes (n= 50,000 to n = 10,000,000) carried out in the experiment. 3-Way Quicksort remain highly competitive in #2 position across all eight test cases. The *Homogeneity Test Heuristic (HTH)* implemented as part of the natural design of FC-Sort enforces the dominance over any known algorithm tested for this special case of all equal keys. Here a simple check confirming that unique keys =1, is unquestionably deterministic, and therefore terminates the sort, and returns the input list as sorted list as no further action is necessary. As results on Table IV indicate, FC-Sort is on the average, approximately 1.1x and 1.8x faster than 3-Way Quicksort and Counting Sort respectively across all eight configurations tested for optimized version of the algorithms. For example, at u = 1, n = 10m, FC-Sort (760.37ms), 3-Way Quicksort (813.32ms) and Counting Sort (1,291.00ms). For the mechanical version, with dataset size n >150,000 3-way Quicksort dominates FC-Sort where unique keys u =1. This is largely due to the increasing overhead in manual frequency loop counting as n increases, unlike the *len()* function employed in the optimized algorithm. Second, the early exit heuristic (HTH) check takes effect after the complete frequency count, and imposing overhead of that activity.

It has been established in the algorithms literature that 3-way Quicksort represents the optimal comparison-based

approach for sorting equal-key datasets, owing to its single equal-zone partition pass making it achieve linear O(n) time [1,7]. This optimality is bounded by the requirement to examine every element/key. This paper demonstrates that FC-Sort, deploying a post -frequency-count early-exit heuristic, termed Homogeneity Test Heuristic (HTH), breaks that bound and achieves a strictly lower constant-factor cost in this dataset domain by eliminating the sort and reconstruction phases entirely upon detection of u=1, reducing the total sorting task to a single O(n) frequency count pass — the theoretical minimum, since any algorithm must examine every element at least once. FC-Sort does not merely match the optimal comparison-based algorithm—3-Way Quicksort in this domain, it dominates it, achieving work that is asymptotically less and, for large n.

F. The Exception:3-Way Quicksort at Very Small u

At very small u (u <= 5), 3-way Quicksort is competitive with or marginally faster than FC-Sort in some configurations, particularly at smaller n. For example, in Table II, at n = 100,000, u = 5, 3-way Quicksort (25.52ms) is found to be slower than FC-Sort (11.17ms). However in the mechanical form at n = 100,000, u = 5, the three algorithms, that is, Counting Sort (40.53ms), 3-way Quicksort (42.95ms), and FC-Sort (45.07ms), are found to be within 5ms of each other, which it a close statistical margin. The explanation for this close margin is structural. At u > 5, the 3-Way Quicksort's competitiveness deteriorates. For example, in Table III; at n = 2,000,000, u = 400, FC-Sort is #1 (242.81ms), and 3-Way Quicksort is a distant #4 (2,829.29ms), which represents more than an order of magnitude performance gap.

TABLE IV
CASE FOR ALL EQUAL (IDENTICAL) KEYS

n = 50,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	3.49ms	0.35ms	1
3-way Quicksort	3.99ms	0.19ms	2
Counting Sort	5.70ms	0.31ms	3
Radix Sort	12.81ms	0.61ms	4
Timsort	38.00ms	1.96ms	5
n = 100,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	6.21ms	0.30ms	1
3-way Quicksort	7.23ms	0.23ms	2
Counting Sort	12.01ms	0.27ms	3
Radix Sort	25.04ms	1.44ms	4
Timsort	83.50ms	2.73ms	5
n = 250,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	19.46ms	2.73ms	1
3-way Quicksort	23.63ms	2.35ms	2
Counting Sort	37.16ms	3.87ms	3
Radix Sort	83.85ms	23.84ms	4
Timsort	280.91ms	40.68ms	5
n = 5,100,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	632.17ms	80.47ms	1
3-way Quicksort	857.12ms	173.82ms	2
Counting Sort	1008.79ms	164.82ms	3
Radix Sort	2679.18ms	392.83ms	4
Timsort	10029.86ms	1009.02ms	5

TABLE IV (continued)
CASE FOR ALL EQUAL (IDENTICAL) KEYS

n = 6,000,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	474.24ms	60.42ms	1
3-way Quicksort	496.52ms	26.41ms	2
Counting Sort	1024.49ms	88.78ms	3
Radix Sort	2315.84ms	259.55ms	4
Timsort	7660.75ms	366.13ms	5
n = 8,000,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	556.08ms	46.16ms	1
3-way Quicksort	674.41ms	55.84ms	2
Counting Sort	1012.41ms	51.82ms	3
Radix Sort	2691.48ms	159.99ms	4
Timsort	10925.26ms	1749.19ms	5
n = 1,000,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	69.56ms	5.17ms	1
3-way Quicksort	82.77ms	3.94ms	2
Counting Sort	141.90ms	8.80ms	3
Radix Sort	309.13ms	13.21ms	4
Timsort	1057.81ms	49.81ms	5
n = 10,000,000 u = 1			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	760.37ms	79.80ms	1
3-way Quicksort	813.32ms	60.60ms	2
Counting Sort	1291.00ms	92.19ms	3
Radix Sort	3109.85ms	222.87ms	4
Timsort	16372.98ms	2591.13ms	5

G. Mechanical Implementation

Table V presents mechanical algorithm results at n = 100,000 across u = 5 to u = 100. The mechanical variant of FC-Sort ranks #1 at u = 5, #2 at u = 10, but falls to #3 and #4 at u >= 20. This degradation is directly attributable to the frequency lookup mechanism. The mechanical version uses a linear scan of u unique keys per input element, yielding O($n \times u$) total lookup cost compared to O(n × 1) for the optimized hash map. At u = 5 this overhead is modest (5 comparisons per element); at u = 100 it is 100 comparisons per element, which means a 20x increase in lookup cost for the frequency phase alone. On the other hand, Counting Sort's mechanical form uses direct array indexing (O(1) per element) regardless of u, which is why it maintains its competitiveness in the mechanical table at higher u values. This finding is a reinforcement of the conclusion in this study that FC-Sort's O(n) complexity is contingent on O(1) key lookup, which is provided automatically by hash-based data structures in Python, Java, C++, and most modern languages. The mechanical results should be interpreted as a baseline description of the algorithm's structure, not as a performance estimate for production deployment.

TABLE V
MECHANICAL IMPLEMENTATIONS

n = 100,000 u = 5			
Algorithm	**Mean**	**Std Deviation**	**Rank**
FC- Sort	35.97ms	3.01ms	1
Counting Sort	37.29ms	7.72ms	2
3-way Quicksort	39.52ms	6.79ms	3
Radix Sort	65.75ms	10.44ms	4
Timsort	433.09ms	73.68ms	5
n = 100,000 u = 10			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	29.37ms	5.14ms	1
FC- Sort	44.84ms	2.42ms	2
3-way Quicksort	47.09ms	7.41ms	3
Radix Sort	89.41ms	11.77ms	4
Timsort	291.87ms	63.42ms	5
n = 100,000 u = 20			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	38.30ms	5.70ms	1
3-way Quicksort	57.36ms	5.26ms	2
FC- Sort	80.39ms	13.01ms	3
Radix Sort	128.10ms	33.63ms	4
Timsort	290.04ms	34.27ms	5
n = 100,000 u = 30			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	30.79ms	5.05ms	1
3-way Quicksort	68.77ms	12.99ms	2
Radix Sort	88.58ms	7.54ms	3
FC- Sort	100.74ms	8.67ms	4
Timsort	278.59ms	27.57ms	5

TABLE V (continued)
MECHANICAL IMPLEMENTATIONS

n = 100,000 u = 40			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	26.73ms	2.66ms	1
3-way Quicksort	80.52ms	14.47ms	2
Radix Sort	91.22ms	11.96ms	3
FC- Sort	130.15ms	9.37ms	4
Timsort	280.76ms	34.97ms	5
n = 100,000 u = 80			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	38.77ms	7.95ms	1
3-way Quicksort	83.51ms	9.04ms	2
Radix Sort	95.53ms	13.62ms	3
FC- Sort	214.69ms	13.78ms	4
Timsort	316.10ms	31.71ms	5
n = 100,000 u = 90			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	26.69ms	1.09ms	1
3-way Quicksort	89.26ms	11.98ms	2
Radix Sort	95.76ms	17.97ms	3
FC- Sort	258.44ms	26.64ms	4
Timsort	281.33ms	22.40ms	5
n = 100,000 u = 100			
Algorithm	**Mean**	**Std Deviation**	**Rank**
Counting Sort	25.86ms	0.85ms	1
3-way Quicksort	94.33ms	8.75ms	2
Radix Sort	141.80ms	34.91ms	3
FC- Sort	261.30ms	15.85ms	4
Timsort	329.33ms	50.62ms	5

H. Structure of Performance Levels

Across all configurations in the defined domain (u <= 100, n >= 100,000), the results consistently reveal three performance levels:

1) Level 1 – O(n) Distribution - Based: The two leading algorithms at this level are FC-Sort and Counting Sort. Execution times: 11ms–500ms at n = 100,000 to n = 2,000,000.

2) Level 2 – O(nlogn): In this category are Radix Sort and 3-way Quicksort. Consistently 3x–10x slower than Level 1 in the target domain.

3) Level 3 – O(nlogn), no Duplicate Exploitation: Timsort (both standard and GP variants). Consistently 15x–45x slower than FC-Sort in the target domain.

This separation is stable across all tested n and u values within the defined domain, and widens as n increases. This three-level structure is the strongest empirical evidence for the central claim in this study, which is: that algorithm design targeting the low-entropy domain (u << n) produces performance advantages that cannot be recovered by general-purpose sorting algorithms, regardless of their sophistication.

I. Summary of Key Findings

The following findings are drawn directly from the experimental results and are presented in order of importance:

(1) FC-Sort ranks first among all algorithms tested at u <= 80 for n >= 100,000 in optimized form, and remains statistically (Mean, Standard deviation and Ranking) equivalent to Counting Sort at u = 90 and u = 100.

(2) Empirical evaluation indicates that by employing FC-Sort's HTH, the algorithm in optimized form, completely bypasses array reconstruction and pointer manipulation, achieving an untouchable performance ceiling that outperforms modern standard library implementations of 3-Way Quicksort, Counting Sort and Timsort in the special case of all identical keys, u = 1. In this case, offering a deterministic, linear-time solution for specialized dataset context.

(3) FC-Sort and Counting Sort form a unique performance level separated from all other algorithms by a factor of at least 3x throughout the defined problem domain.

(4) FC-Sort's execution time is largely insensitive to u within u = 5 to u = 100 at fixed n, confirming that the sort phase $O(u^2)$ is negligible in practice.

(5) FC-Sort's advantage over Timsort grows with n, reaching 45x at n = 2,000,000, u = 5. This is consistent with the O(n) vs O(n log n) complexity difference.

(6) In the benchmark, keys are contiguous integers (r = u), which is the most favourable case for Counting Sort. Inreal-world sparse-key applications, FC-Sort's advantage over Counting Sort will be larger than these results indicate.

(7) The mechanical variant of FC-Sort is weaker at u >= 20 due to linear-scan frequency lookup (O(n × u)). In any

Language providing native hash maps, this limitation does not arise.

(8) 3-way Quicksort is competitive only at u <= 5 in the mechanical form. At u >= 10 in the optimized form, FC-Sort is consistently faster by a substantial margin.

J. Strengths

First, the proposed algorithm does not inherit the key-range dependency associated with Counting Sort or the digit-processing requirements of Radix Sort. Second, the algorithm is specifically designed for low-entropy datasets in which the number of unique keys is significantly smaller than the total number of observations ($u << n$). By compressing duplicate values into frequency counts, the algorithm effectively reduces the sorting space from (n) elements to only (u) unique keys. As a result, the computational effort associated with ordering the unique values becomes negligible relative to the linear counting and expansion phases. This allows the algorithm to exploit duplicate heavy datasets more efficiently than comparison-based algorithms that must repeatedly process all (n) elements. Consequently, it can be said that there exists a broad low-entropy region ($u<<n$) where $u <=$ 100, where FC-Sort becomes the fastest algorithm tested.

K. Weaknesses

First is stability. Stability of an algorithm refers to when elements with the same value appear in the same order when sorted as they do in the input array prior to sorting operation [2]. FC-Sort as presented in this study is unstable. The algorithm achieves its efficiency by compressing duplicate keys into frequency counts. This compression discards the relative ordering information among equal-key elements, making the algorithm inherently unstable unless additional occurrence-tracking structures are introduced. A primary limitation of FC sort algorithm for equal keys is its inherent instability. By compressing duplicate values into frequency counts, the algorithm sacrifices the ability to preserve the original ordering of equal-key elements.

Second is the deterioration under high entropy datasets. FC sort algorithm is specifically optimized for low-entropy datasets with substantial duplicate keys. As the number of unique keys approaches the number of observations ($u \rightarrow n$), and duplicate keys become sparse or non-existent, the benefits of frequency compression reduce, and in fact, is lost completely for a worst case scenario where all observable keys are unique. Under such high-entropy conditions, the algorithm experiences a progressive reduction in efficiency and becomes less competitive than conventional comparison-based O($n$log$n$) sorting algorithms. As a result, its practical applicability is strongest in domains where duplicate values are abundant and the number of unique keys remains relatively small ($u << n$). As results indicate, FC-Sort loses its supremacy to Counting sort in the dataset domain under study, as u progressively gets larger in the neighbourhood of u >= 2,000.

V. CONCLUSION & FUTURE WORK

This paper introduces Frequency Count Sort (FC-Sort), a distribution-based sorting algorithm designed explicitly for low-entropy datasets in which the number of unique keys u is substantially smaller than the total number of observations n. By reducing the sorting problem to the unique keys in the input list, that is, a frequency map of u unique keys, a sort of those keys alone, and then a linear reconstruction, FC-Sort achieves near-linear time complexity $O(n + u^2)$ that degrades gracefully as u grows and converges to $O(n)$ within the defined problem domain of $u \leq 100$. Empirical benchmarks across a broad configuration space ($n \in$ {100,000 to 2,500,000}, $u \in$ {5 to 100}) demonstrate and confirm that FC-Sort ranks first or remains statistically equivalent to the best-performing competitor (Counting Sort) in every optimized configuration tested within this domain.

It has been established in the algorithms literature that 3-way Quicksort represents the optimal comparison-based approach for sorting all-identical key datasets, owing to its single equal-zone partition pass making it achieve linear O(n) time [1], [7]. This optimality is bounded by the requirement to examine every element/key. This paper demonstrates that FC-Sort, deploying a post-frequency-count early-exit heuristic, termed Homogeneity Test Heuristic (HTH), breaks that bound. In this specific all-equal keys domain, it achieves a strictly lower constant-factor cost by eliminating the sort and reconstruction phases entirely upon detection of $u = 1$. This reduces the total sorting task to a single O(n) frequency count pass — the theoretical minimum, since any algorithm must examine every element at least once. FC-Sort does not merely match the optimal comparison-based baseline of 3-Way Quicksort in this single key domain, it dominates it, achieving work that is asymptotically less and, for large n.

The algorithm is intentionally simple: three stages, implementable from pseudocode in under 20 lines and portable across any language supporting hash-based associative structures. This simplicity is a feature and it lowers the barrier to adoption, auditing, reproducibility and verification.

Two directions for future work are identified. First, considering that the algorithm demonstrates efficiency with some specialized dataset with equal keys, a fusion with another algorithm with O($n$log$n$) complexity in a hybrid design could potentially result in a very competitive O($n$log$n$) algorithm. Second, the algorithm design practically condenses the input list to the much smaller sub-list of unique keys, and given that the frequency count of all unique elements are known before the left to right serial expansion, a significant time incurred in serial writing to target list could be reduced under parallelism. This would mean that the writing to the target list of each or some unique elements could be assigned to separate processors in the machine such that writing operation occurs simultaneously saving time and thus improving runtime.

REFERENCES

[1] R. Sedgewick, "Quicksort with Equal Keys," *SIAM Journal on Computing*, Vol. 6, pp. 240 – 267, June 1977 https://doi.org/10.1137/0206018

[2] T.H. Cormen, C.E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, 4th Edition", *MIT Press*, pp. 89, 313, - 314, 290 – 295, 2022

[3] C.A.R. Hoare, "Quicksort", *The Computer Journal*, vol 5, pp. 10–16, 1961, doi.org/10.1093/comjnl/5.1.10

[4] R. Sedgewick, "The Analysis of Quicksort Programs", *ACTA Informatica*, Vol 7, pp. 327–355, 1977, doi.org/10.1007/BF00289467

[5] E. Dijistrar, "A Discipline of Programming, The Dutch National Flag Problem*", Prentice-Hall*, Chapter 14, 1976

[6] J. L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings", *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms,* pp. 360–369, New Orleans, 1997

[7] S. Wild, "Quicksort is Optimal for Many Equal Keys", *P*roceedings of the Fifteenth Meeting on Analytic Algorithmics and Combinatorics (ANALCO), SIAM, 201. doi.org//10.1137/1.9781611975062.2