JCHAOSINDEX: MEASURING AND BENCHMARKING DISPERSION IN RANDOMIZED DATA

Jui Keskar

Metropolitan School, Frankfurt, Germany

ABSTRACT

Randomization of data is an ongoing need for various business reasons like design of clinical trials, or training an AI model, to name a few. To control the level of randomization, it is important to measure the level of randomness, i.e. unpredictability and dispersion, in the "randomized" data vis-à-vis the original data. While Permutation entropy measures unpredictability, there is no technique that measures dispersion of randomized data. To measure dispersion in randomized data, "Neighbour-displacement-delta" (NDD) based technique is proposed. JChaosIndex, measure of dispersion, considers displacement of each data element as well as relative displacements of the neighbours of each data element. Higher the JChaosIndex, more disersed is the randomized data. JChaosIndex measurement technique can be easily included in a programming language library or database methods or any algorithm. Importantly, this technique is domain-agnostic as it works purely on the indexes of the data record and not the actual data.

KEYWORDS

Measure of Randomness, Data Dispersion, JChaosIndex, Permutation Entropy, Neighbour Displacement Delta

1. INTRODUCTION

Randomness has been studied for long in the context of algorithms, probability, and complexity [1][5]. There also exists "Runs test" to test the randomness of samples [6]. It is based on statistical techniques [7]. There also is a need to measure randomness in the context of data processing and data storage. Such randomization of data can be business critical at times. This could be either for the reasons of privacy, or for removing time bias in the data (to avoid recency effect [8]) or just for fairness (say for the purpose of gaming [9]). Randomization is usually done in programs by using the random number generators [10].

However, it is important to be able to measure level of randomness in the randomized data (as compared to the original data). This will not only help in knowing if the randomized configuration is indeed the most optimal one, but will also help in designing a randomization technique that will provide the best configuration.

This paper describes in detail the technique for calculating JChaosIndex - a measure of dispersion generated by any data randomization process. The combination of permutation entropy and JChaosIndex can be useful in comparing various randomized configurations to pick the most suitable one.

2. Methods

The method involves measuring the movements in the randomized data vis-à-vis the original data.

2.1. Measuring Dispersion in Randomized Data Using Jchaosindex

In this technique, to be able to measure dispersion of the “randomized” data, one needs to measure the distance of the absolute new position of a particular data element (record) from its absolute original position before randomization. One also needs to check the relative position change of the neighbours of the data element vis-a`-vis neighbours’ old relative position (old relative position of neighbour minus new relative position of that neighbour).

This is because a record may have moved very far from its original position, but if its neighbours are still the same in the new configuration, then the dispersion level is low. Total dispersion of data elements and their neighbours also needs to be normalized by the size of data for ease of comparison across datasets of varying sizes.

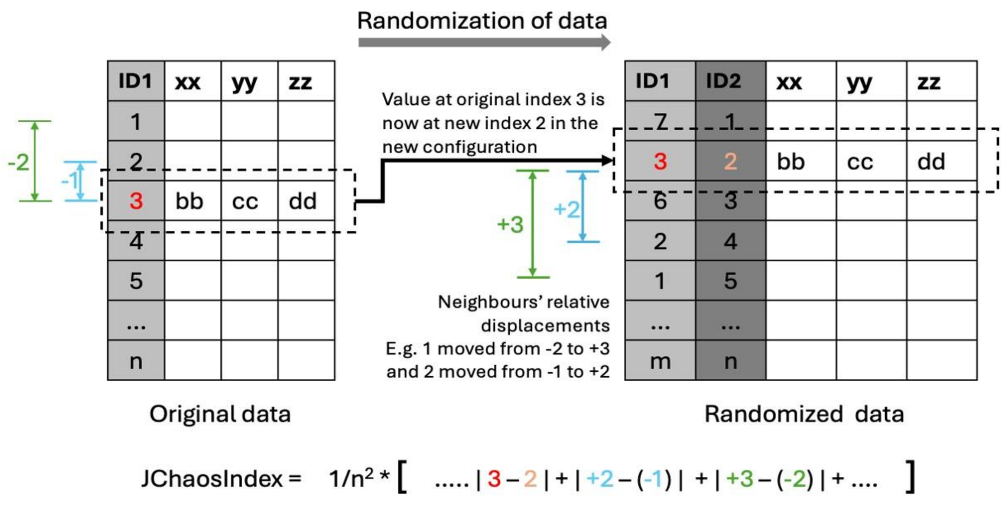

Randomization of data
ID1
xx
yy
zz
1
2
3
bb
cc
dd
4
5
...
n
-2
-1
Original data
Value at original index 3 is now at new index 2 in the new configuration
+3
+2
Neighbours’ relative displacements
E.g. 1 moved from -2 to +3 and 2 moved from -1 to +2
ID1
ID2
7
1
3
2
6
3
2
4
1
5
...
m
n
Randomized data
JChaosIndex = 1/n² * [..... | 3 – 2 | + | +2 – (-1) | + | +3 – (-2) | + ...]

Figure 1: Calculating the neighbour-displacement delta (NDD)

Figure 1 depicts the nuances involved in measuring dispersion. Few points to be noted about Fig. 1 are as follows: ID1 is the index of original data in the original record collection.

ID2 is the new index of data in the new randomized data collection. Also, one can see that ID1 is now randomized in the new randomized data collection. As an example, for the left (original) data collection, the blue and orange arrows highlight the original relative positions of a few neighbours (1 and 2) relative to a specific data element (3). Whereas, for the right (randomized) data collection, the green and blue arrows highlight the new ‘displaced’ positions of these neighbours (1 and 2) relative to new position (ID2=2) of the original data element (ID1=3). While implementing the measurements (JChaosIndex), one needs to temporarily create ID1, ID2 fields and have them pre-populated before and after randomization respectively.

Also, methods old() and new() are needed. Function old() will take ID2 value as an argument and return the corresponding value in the ID1 field of the specific data element [i.e. ID1 = old(ID2) e.g. 3 = old(2)] . Function new() will take ID1 value as an argument and return the corresponding value in the ID2 field [i.e. ID2 = new(ID1) e.g. 2 = new(3)] . Please note, both ID1 and ID2 have all unique values without repetition.

Comparing the values in column ID1 and ID2 can provide information about absolute position change of each record. Ex. Data in ID1=3 in the original data is now at position ID2=2 in the Randomized data. Hence, it has moved by 1 position [$|3-2|$]. One also needs to check the relative position of neighbours of each of the data element and check how their distance from the data element has changed in the new "randomized" data. E.g. In the Original data, one of the neighbours of ID1=3 was ID1=1. The relative position of 1 vis-a-vis 3 was -2 (i.e.1-3). For convenience of calculations, positions lesser than the given data element is considered negative and higher ones are considered positive. In the new configuration, when the data element (ID1=3) has moved to ID2=2 and the "neighbour" ID1=1 is now moved to ID2=5. Thus, a neighbour which was -2 records away from that data element is now 5-2 = 3 records away. The change of this neighbour's position relative to the data element (ID1=3) is $|3-(-2)| = 5$ record.

In this way, relative dispersion of each data elements and that of their neighbours can be calculated and positive value (mod) of all such relative displacements are added together and normalized (divided) using the square of total record count (n^2).

$$JChaosIndex = \frac{1}{n^2}\sum_{p=0}^{p=n-1}\left[\,|p-old(p)| + \sum_{i=-old(p)}^{i=1-old(p)}[|new(old(p)+i|]\,\right]$$

Normalization using 'n^2' provides a near-straight-line relationship between maximum JChaosIndex and number of records, hence is most suited for normalization.

2.2. Getting most Dispersed Configuration

Most dispersed configuration can be achieved if the dispersion levels of all possible permutations of data can be measured numerically. The configuration that corresponds to the max dispersion level can then be picked up for further processing. Figure 2 depicts this approach. Please note, calculating JChaosIndex requires both the new "randomized" configuration as well original configuration before randomizing the data.

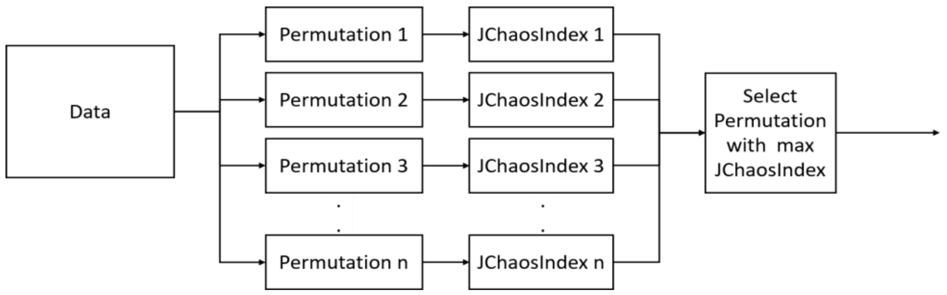


Figure 2: How to get most dispersed configuration

Following is one way to get to the most dispersed configuration:

....

```
$JChaosIndex_{max}$ = 0
BestDataConfigurationIndexArray = []
Iterate through all possible data configurations:
```

NewDataConfigurationIndexArray = [< *originalindexinthenewseq* >] Calculate Randomization Index for this configuration:

$$JChaosIndex = \frac{1}{n^2} \sum_{p=0}^{p=n-1} [\, |p - old(p)| + \sum_{i=\,-old(p)}^{i=1-old(p)} [|new(old(p) + i|] \,]$$

Compare the index with the earlier max index: If $JChaosIndex > JChaosIndex_{max}$:

$JChaosIndex_{max}$ = JChaosIndex

BestDataConfigurationIndexArray = NewDataConfigurationIndexArray

. ...

Annexure 8.1 contains a sample Python program for calculating the most dispersed configuration of an 8-element-long array.

Following are the details of Program (Annexure 8.1):
Original List is [11, 12, 13, 14, 15, 16, 17, 18]
Most dispersed configuration is [18, 17, 16, 15, 14, 13, 12, 11]
Total displacement = 368
Max JChaosIndex = 5.75

This however can be expensive on CPU due to the sheer number of permutations possible. E.g. for just 8 data points, the possible permutations are 40,320 (i.e. 8!). Calculating the level of displacement for each permutation can be time consuming. Also, the above configuration with maximum JChaosIndex is predictable making it undesirable. Hence, a more pragmatic technique is to use random number generators to come up with a few random configurations and to measure the permutation entropy and dispersion level of each of the configurations and pick the one with optimal randomness level. Another approach is to check permutation entropy of each permutation, pick the ones with maximum permutation entropy and then get the most dispersed ones from them.

2.2.1. Challenges with Perfectly Dispersed Configuration

A most dispersed configuration selected using the above technique will have the highest JChaosIndex. For this selected configuration, each record and its neighbours have the maximum displacement vis-`a-vis their original position. But the configuration with maximum dispersion is always the configuration which is the exact reverse sequence of the original sequence of records. This makes it too predictable. One can always predict the original sequence from the randomized sequence if one knows that the new configuration has the highest JChaosIndex! Hence, the key is to get the most unpredictable and dispersed configuration (rather than the most displaced one only) through the judicious use of permutation entropy and JChaosIndex.

2.2.2. How is Permutation Entropy Useful?

Permutation entropy is a measure of the complexity of a time series based on the ordinal patterns (permutations) of its values over time [4]. It is particularly suited for analysing time series data, where the order of data points is crucial. However, it can still use permutation entropy by treating the data series as a time series with an implicit order. [11].

For a given embedding dimension (m) and time delay (τ = 1)

$Normalized\ Permutation\ Entropy\ (H) = -\frac{[\sum P(\pi)\log(P(\pi))]}{\log(m!)}$ (Added over all possible permutation patterns of length m) [4].$P(\pi)$ in the equation is the probability of a permutation pattern π.

2.2.3. How to Get the Most Random Configuration?

For most random configuration, the randomization process needs to be conducted in two steps. Firstly, identify all possible unpredictable configurations using permutation entropy and then look for most displaced ones among them. Figure 3 depicts the technique to get most random configuration.

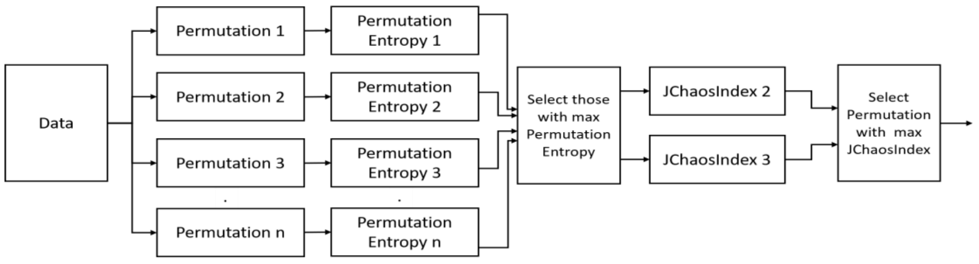


Figure 3: How to get most random configuration

Following is one way to get to the most optimal configuration:

```
..........................................................................................................................................
    JChaosIndexopt = 0 //Optimal displacement
    OptDataConfigurationIndexArray = [] //Optimal configuration
    Embedding Dimension = m //Embedding dimension
    period = $\tau$ //period of data series

    Calculate maximum permutation entropy (m p e) for given data size, embedding dimension
and period
    Iterate through all possible data configurations:

        For each NewDataConfigurationIndexArray:

        Calculate Permutation Entropy (p e) for each configuration:
        If m p e == p e:

            Calculate Randomization Index for this configuration:

$$JChaosIndex = \frac{1}{n^2}\sum_{p=0}^{p=n-1}\left[\,|p - old(p)| + \sum_{i=-old(p)}^{i=1-old(p)}[|new(old(p)+i|]\ \right]$$

          If JChaosIndex >JChaosIndexopt:
                         JChaosIndexopt = JChaosIndex
                         OptDataConfigurationIndexArray = NewDataConfigurationIndexArray
. .........................................................................................................................................
```

Annexure 8.2 contains a sample Python program for calculating permutation entropy of any data series. Annexure 8.3 contains sample program for calculating most unpredictable and dispersed configuration of an 8-element-long array using permutation entropy and Neighbour-displacement-delta (NDD). Optimal configuration selected is the one with maximum permutation entropy score of 2.585 and optimal JChaosIndex of 5.3125 within the enforced constraints regarding unpredictability.

For an 8-element-long array, out of all the possible 40,320 permutations there are about 2396 permutations that have highest permutation entropy of 2.585. JChaosIndex helps in identifying the most dispersed configurations from these most unpredictable ones.

Following are the details of the two most unpredictable and most dispersed configurations:

1)
Original List is [11, 12, 13, 14, 15, 16, 17, 18]
Constraints = Maximum permutation entropy AND optimal dispersion of data elements
Best Randomization is [17, 15, 18, 16, 12, 13, 14, 11]
Total displacement = 340
Optimal JChaosIndex = 5.3125

2)
Original List is [11, 12, 13, 14, 15, 16, 17, 18]
Constraints = Maximum permutation entropy AND optimal dispersion of data elements
Best Randomization is [18, 15, 16, 17, 13, 11, 14, 12]
Total displacement = 340
Optimal JChaosIndex = 5.3125

While these are not the one with most displacement, they will surely provide the most optimal random configurations while making them harder to predict than the most dispersed one. Figure 4 highlights graphical views of these configurations.

3. RESULTS

Analysis involved measuring the overall dispersion level for each of the combination – optimal or otherwise.

3.1. Analysing Random Configurations Based Various Constraints

Analysing the most dispersed (maximum JChaosIndex) array (as shown in Fig. 4) reveals the obvious. The data at extreme position is dispersed the most, while the data towards the centre is dispersed by least position. The challenge with this distribution is that it is not uniform and it is too easily predictable.

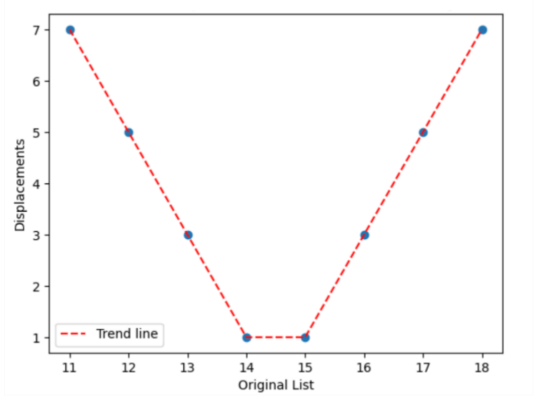


Figure 4: Positional displacement for most dispersed configuration

Thus, it is important to get more unpredictable randomization across all the data points. This can be done by ensuring permutation entropy to be maximum and then looking for the most dispersed configurations. Fig 5 [(a) and (b)]includes graphical view of the two most optimal configurations for 8-element-long array. Fig. 6 [(a) and (b)] includes two second-most optimal configurations achieved for 8-element-long array. For these second-most optimal configurations, the unpredictability (PE) is still the highest while dispersion (measured using NDD JChaosIndex) is at 5.25 .

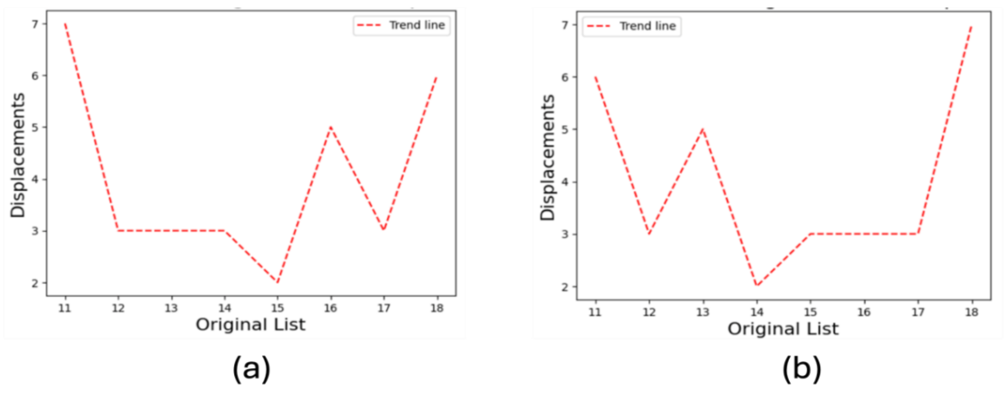


Figure 5: Displacement of positions with max PE and most optimal NDD cases (5.3125)

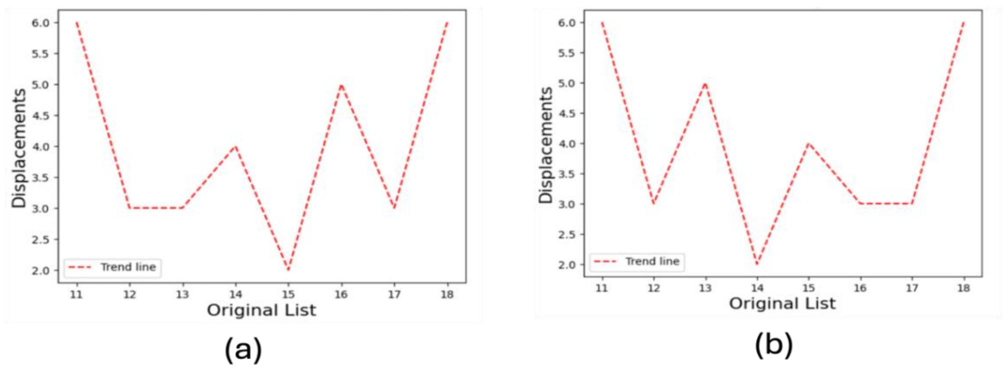
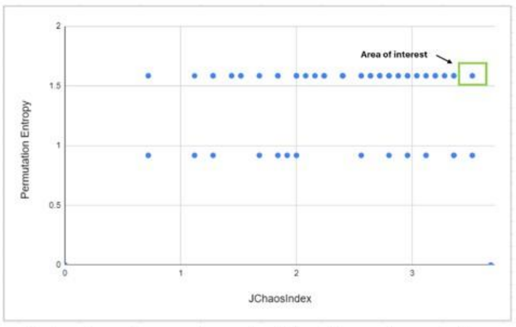
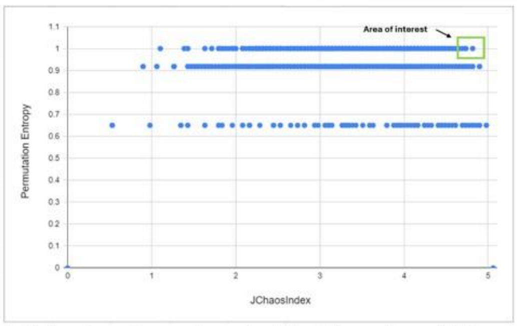


Figure 6: Displacement of positions with max PE and 2nd optimal NDD cases (5.25)

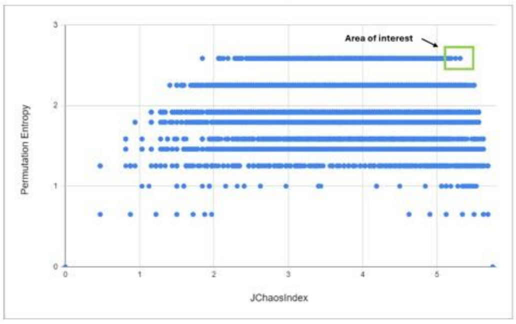
Figure 7 depicts the distribution of Permutation Entropy and JChaosIndex for various data records sizes for all possible permutations.



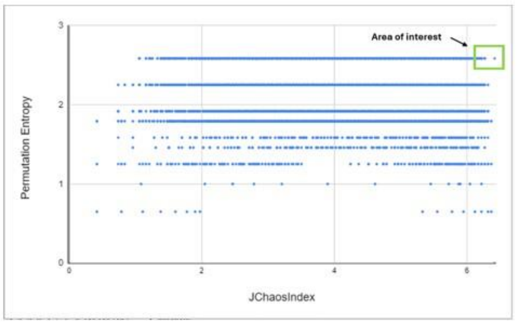
a. Data size = 5 records, embedding dimension = 1, Time delay = 1 (permutations = 120)



b. Data size = 7 records, embedding dimension = 2, Time delay = 1 (permutations = 5,040)



c. Data size = 8 records, embedding dimension = 3, Time delay = 1 (permutations = 40,320)



d. Data size = 9 records, embedding dimension = 3, Time delay = 1 (permutations = 362,880)

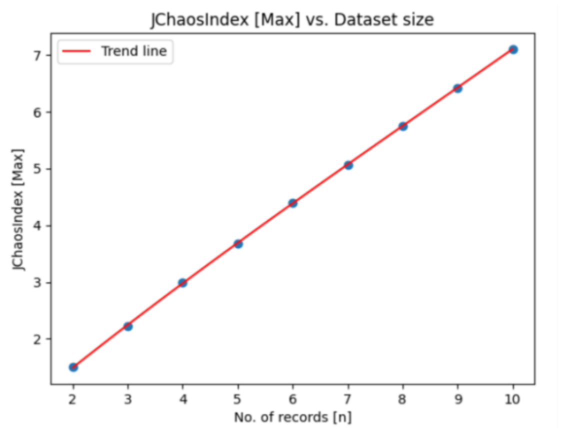
Figure 7: Distribution of Permutation Entropy and JChaosIndex for various data records sizes for all possible permutations.

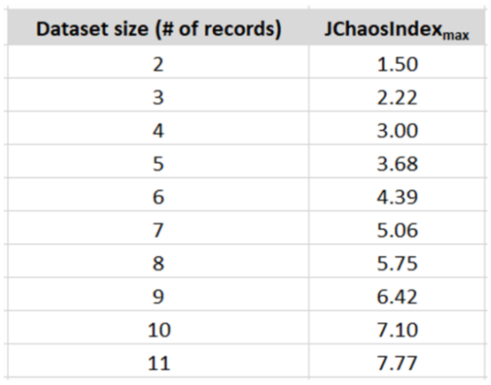
From these analyses, clearly, displacements of 0 or 1 for an 8-element-long data array (25% of data records count or less) are absent in the optimal configurations. Also, for the most optimal ones, the maximum displacement is going all the way up to 7 (which is max possible given the array size of 8).

3.2. Large Data Records

So far, the dataset had only 8 records (data elements). However, in the real world, there are much larger records that keep growing further. Analysis was done on datasets of varying sizes to find out max JChaosIndex for each of these sets. Figure 8 shows the trend of max JChaosIndex as the dataset size is increased.

It is interesting to find that max JChaosIndex has a straight-line-like relationship with the





Dataset size (# of records)	JChaosIndex$_{max}$
2	1.50
3	2.22
4	3.00
5	3.68
6	4.39
7	5.06
8	5.75
9	6.42
10	7.10
11	7.77

Figure 8: Max JChaosIndex for various dataset sizes

number of records (n) for the small-sized datasets. The red dotted-line in the chart is the trend-line, which connects each of the data points almost perfectly.

As detailed in Annexure 8.4, equation of this line is:

$$\textit{Max JChaosIndex} = 0.0007 * n^3 - 0.0182 * n^2 + 0.8317 * n - 0.1041$$

This equation has been derived using data upto 10 data records and tested on the data point of 11 data records with 99.93% accuracy

This can be used to know the max dispersion score (JChaosIndex) for any number of records. For example, a dataset with 100 records, the max JChaosIndex possible is about 601. This number will be useful while calculating the optimal randomness. As seen for the case of 8 record dataset, optimal randomness for any dataset can be found for cases when unpredictability of the combination is highest (permutation entropy is 2.585 for 8-element-long array) and JChaosIndex is also high (5.3125) for them. In general, minimum dispersion of more than 25% of the dataset size can fetch better configurations provided the dispersion distribution across data elements remain unpredictable.

3.3. Benchmarking Various Random Number Generators

Computers usually have two types of random number generators; viz. True Random Number Generators (TRNG) and Pseudo-Random Number Generators (PRNG)[10]. Randomization of dataset is often performed using the Pseudo-Random Number Generators (PRNG) of various programming language libraries. The easiest step is to generate a non-repeated random number and allocate that position to a data point that needs to be randomized. Permutation entropy and JChaosIndex can be calculated for configurations thus generated using above technique. As part of this research, the average and maximum Permutation entropy and JChaosIndex for such randomization was calculated for four popular programming languages. Annexure 8.5 contains the detailed programs used for measuring randomization performed in each of these languages.
Following are the summarized findings:

As seen from Table 1, C-Sharp's library function for random number generation is providing themost unpredictability as compared to Python, Java and C libraries. Interestingly, none of these languages could provide the most unpredictable configuration (permutation entropy of 2.85)!

Table 1: Benchmarking of random numbers generated by various language libraries

Language	**Library/method**	**Permutationentropy (PE) andJChaosIndex (JC) in 5 attempts**	**Mean PE, JC**	**Max PE,JC**
Python	random.random()[12]	PE: 2.25, 2,25, 1,58, 1.79, 1.79 JC: 3.78, 4.18, 4.53, 4.09, 3.75	1.93, 4.07	2.25, 4.53
Java	Random.nextInt()[13]	PE: 2.25, 2.25, 1.58, 1.79, 1.79 JC: 3.25, 3.81, 3.25, 2.26, 3.97	2.02, 3.31	2.25, 3.97
C	rand()[14]	PE: 1.91, 1.79, 1.91, 2.25, 2.25 JC: 3.88, 3.25, 2.75, 2.4, 3.31	2.02, 3.12	2.25, 3.88
C Sharp	Random().next()[15]	PE: 1.91, 2.25, 2.25, 2.25, 2.25 JC: 2.75, 4.16, 3.97, 4.09, 3.38	2.18, 3.67	2.25, 4.16

4. DISCUSSION

JChaosIndex is an objective measure of dispersion generated in any data through randomization. It is independent of the actual data and hence domain-agnostic. It uses the indexes of the original data and the ones in the randomized data. JChaosIndex is the total dispersion of all data elements and their neighbours due to randomization, normalized using the square of the number of records in the dataset.

As seen in Fig. 8 earlier, as the number of records increases, the max JChaosIndex also increases. This may be happening since an increased number of records opens up opportunity for more bigger dispersion. This is an important finding which highlights consistency of the JChaosIndex measure.

The challenge with most dispersed configuration is that it is unfair on the central records, where it does not do much displacement. Predictability of most dispersed configuration is the biggest disadvantage.

As seen in Fig 4, most random configurations can be found through finding most unpredictable combinations using permutation entropy measurement technique followed by most dispersed configurations using JChaosIndex measurement technique on them. For an 8-element-long array having 40,320 possible permutations about 2396 permutations have highest permutation entropy (2.585). Out of these highest entropy permutations, two most dispersed configurations could be found to have JChaosIndex of 5.3125. In these configurations, each of the individual data element's displacements was greater than 2 (25% of the data record size) Sometimes, the data may need more randomization of data added recently (or during a specific period). In such cases, the constraints can include a nonlinear function like a Sine function or a logarithmic function. Such provisions can be easily made in programs of controlled randomization included in Annexure 8.3.

5. CONCLUSION

JChaosIndex is a consistent, objective and easily measurable score for dispersion generated by any randomization technique. It is data-agnostic, domain-agnostic as it works only on the indexes before and after randomization. It can be used for measuring dispersion of any size of data that has undergone randomization. Dispersion includes displacement of each data element as well as that of its neighbours vis-`a-vis the data element.

By comparing JChaosIndex, one can arrive at the most dispersed configurations. However, such most dispersed configurations may be exactly the reverse of original sequences, hence are predictable. Hence, while doing randomization, one should look for optimal configurations that are dispersed yet unpredictable. This is possible through combining permutation entropy and NDD techniques, where permutation entropy can help in shortlisting most unpredictable configurations and NDD (measured as JChaosIndex) can identify most displaced configurations among the most unpredictable ones.

Finally, using the permutation entropy, random number generator functions of various programming language libraries were benchmarked. C-Sharp's random number generator library is providing more unpredictable numbers as compared to the other such libraries of Python, Java and C. Python on the other hand provides most dispersed random data.

6. FUTURE WORK

Future work can focus on analysing various configurations of higher order datasets using super computing infrastructure. Future work may also involve defining an unified measure combining unpredictability and JChaosIndex and inclusion of the this measure in the programming language libraries (random number generator libraries). It can also be included as part of database utilities for randomization of data while committing any new data insertion.

ACKNOWLEDGEMENTS

I would like to thank my parents, my mentors and my teachers for their encouragement. I would also like to thank my school for allowing me to use the computers.

REFERENCES

[1] W. A. Mathematics and computation - a theory revolutionizing technology and science. 2019.
[2] C. K. e. a. Berger V., Bour L. (2021) A roadmap to using randomization in clinical trials. *BMC Medical Research Methodology*, 21(168).
[3] J. Lin, A. Zhang, M. Lecuyer, J. Li, A. Panda, and S. Sen. (2022) Measuring the effect of training data on deep learning predictions via randomized experiments.
[4] B. C. and P. B. (2002) Permutation entropy: A natural complexity measure for time series. *Physical Review Letters*, 88(17).
[5] M. R. and R. P. (1995) Preface. page ix–xiv.
[6] Test for randomness. (2012).
[7] B. M. and S. F. (2018) An application of the runs test to test for randomness of observations obtained from a clinical survey in an ordered population. *Malays Journal of Medical Science*, 25(4):146–151.
[8] C. A. Morrison A. and C. J. (2014) Primacy and recency effects as indices of the focus of attention. *Frontiers in Human Neuroscience*, 8.
[9] e. Montiel I. (2022) Loot box engagement: A scoping review of primary studies on prevalence and association with problematic gaming and gambling. *PLOS ONE*.
[10] P. W. (2007) Numerical recipes - the art of scientific computing.
[11] Z. L. and et.al. (2009) Forbidden patterns, permutation entropy and stock market inefficiency. *Physica A: Statistical Mechanics and its Applications*, 388(14):2854–2864.
[12] Random - generating pseudo-random numbers (2024).
[13] Class random from oracle documentation (2024).
[14] Cpp reference - rand from cppreference.com (2022).
[15] .net - random class from Microsoft (2024).

ANNEXURES

Program for most displaced randomization

```
#!/usr/bin/env python
# coding: utf-8
#Author: Jui Keskar (Copyright)
import random
from joblib import Parallel, delayed
origList = [11, 12, 13, 14, 15, 16, 17, 18]
OrigIndexMapList = [0, 1, 2, 3, 4, 5, 6, 7]
finalList = [-1, -1, -1, -1, -1, -1, -1, -1]

def findindex(newpos, currentPerm):
    for k in range(len(currentPerm)):
        if newpos == currentPerm[k]:
            return k
    return -1

def calculate_distance(currentPermutation):
    distance = 0
    for j in range(len(currentPermutation)):
locdist = abs(j - findindex(OrigIndexMapList[j], currentPermutation))

        distance += locdist
        for n in range(j):
locdist = abs((j - n) - (findindex(OrigIndexMapList[j], currentPermutation)
            - findindex(OrigIndexMapList[n], currentPermutation)))
            distance += locdist
        for n in range(len(currentPermutation) - j):
            if j + n <len(currentPermutation):
locdist = abs(n - (findindex(OrigIndexMapList[j + n], currentPermutation)
                - findindex(OrigIndexMapList[j], currentPermutation)))
                distance += locdist
    return distance, currentPermutation

def generate_combinations():
    pool = tuple(OrigIndexMapList)
    n = len(pool)
    indices = list(range(n))
    cycles = list(range(n, 0, -1))
    yield tuple(pool[i] for i in indices)
    while n:
        for i in reversed(range(n)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices)
                break
        else:
            return

combinations = generate_combinations()
```

```
results = Parallel(n_jobs=-1)(
    delayed(calculate_distance)(perm)
    for perm in combinations
)
finalDistance, bestPermutation = max(results, key=lambda x: x[0])
for k in range(len(origList)):
finalList[k] = origList[bestPermutation[k]]
print("Original List is", origList)
print("Best Randomization is", finalList)
print("Total displacement =", finalDistance)
print("JChaosIndex =", finalDistance / len(origList) / len(origList))
……………………………………
```

Output upon execution:

```
Original List is [11, 12, 13, 14, 15, 16, 17, 18]
Best Randomization is [18, 17, 16, 15, 14, 13, 12, 11]
Total displacement = 368 Max JChaosIndex = 5.75
```

Output for bigger data arrays:

```
9-element array:
Original List is [11, 12, 13, 14, 15, 16, 17, 18, 19]
Best Randomization is [19, 18, 17, 16, 15, 14, 13, 12, 11]
Total displacement = 520
Max JChaosIndex = 6.419753086419753
10-element array:
Original List is [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Best Randomization is [20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
Total displacement = 710 Max JChaosIndex = 7.1
11-element array:
Original List is [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
Best Randomization is [21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
Total displacement = 940
Max JChaosIndex = 7.768595041322314
```

Note: Parallel processing on high configuration machine was used as bigger arrays had millions of permutations

Program for calculating permutation entropy

```
#!/usr/bin/env python
# coding: utf-8
#Shreeram
#Author: Jui Keskar (Copyright)
import numpy as np
from itertools import permutations
from collections import Counter
import itertools
import random
import csv
origList = [11,12,13,14,15,16,17,18]
OrigIndexMapList = [0,1,2,3,4,5,6,7]
indexMapList=[]
finalList = [16, 18, 11, 12, 15, 13, 17, 14]

def findindex(newpos, currentPerm):
    for k in range(len(currentPerm)):
        if newpos == currentPerm[k]:
            return k
    return -1
```

```
def permutation_entropy(time_series, m, tau):
    n = len(time_series)
ordinal_patterns = []
    # Extract ordinal patterns
    for i in range(n - (m - 1) * tau):
sub_seq = time_series[i:(i + m * tau):tau]
ordinal_pattern = tuple(np.argsort(sub_seq))
ordinal_patterns.append(ordinal_pattern)
    # Count patterns
pattern_counts = Counter(ordinal_patterns)
total_patterns = len(ordinal_patterns)
    # Calculate relative frequencies
pattern_frequencies = np.array(list(pattern_counts.values())) / total_patterns
    # Compute permutation entropy
    entropy = -np.sum(pattern_frequencies * np.log2(pattern_frequencies +
np.finfo(float).eps))
    return entropy

print("Original List is", origList)
print("Permutation_entropy is:",permutation_entropy(finalList, 3,1))
```

Program for all random configuration with permutation entropy as well as JChaosIndex

```
import numpy as np
from itertools import permutations
from collections import Counter
import itertools
import random
import csv
origList = [11,12,13,14,15,16,17]
OrigIndexMapList=[0,1,2,3,4,5,6]
finalList = [-1,-1,-1,-1,-1,-1,-1]
listOfPermutations = list(itertools.permutations(OrigIndexMapList, len(origList)))
finalDistance = 0
bestPermutation = 0
def permutation_entropy(time_series, m, tau):
    n = len(time_series)
ordinal_patterns = []
    # Extract ordinal patterns
    for i in range(n - (m - 1) * tau):
sub_seq = time_series[i:(i + m * tau):tau]
ordinal_pattern = tuple(np.argsort(sub_seq))
ordinal_patterns.append(ordinal_pattern)
    # Count patterns
pattern_counts = Counter(ordinal_patterns)
total_patterns = len(ordinal_patterns)
    # Calculate relative frequencies
pattern_frequencies = np.array(list(pattern_counts.values())) / total_patterns
    # Compute permutation entropy
    entropy = -np.sum(pattern_frequencies * np.log2(pattern_frequencies
    + np.finfo(float).eps))
    return entropy

def findindex(newpos, currentPerm):
    for k in range(len(currentPerm)):
        if newpos == currentPerm[k]:
            return k
    return -1
# Prepare to write to CSV
csv_file = "C:\\Jui\\papers\\Randomization\\code\\data\\permutations_and_distances_new7.csv"
with open(csv_file, mode='w', newline='') as file:
    writer = csv.writer(file)
writer.writerow(["Permutation", "JChaosIndex", "Entropy"])  # Write header
    for i in range(len(listOfPermutations)):
currentPermutation = listOfPermutations[i]
        distance = 0
        for j in range(len(currentPermutation)):
            distance = distance + abs(j - (findindex(OrigIndexMapList[j],
            currentPermutation)))
            for n in range(j):
                if j - n > 0:
                    distance = distance + abs((j - n)
                    - (findindex(OrigIndexMapList[j], currentPermutation)
                    - findindex(OrigIndexMapList[n], currentPermutation)))
            for n in range(len(currentPermutation) - j):
                if j + n <len(currentPermutation) and n > 0:
                    distance = distance + abs((+n)
                    - (findindex(OrigIndexMapList[j + n], currentPermutation)
                    - findindex(OrigIndexMapList[j], currentPermutation)))
            if finalDistance< distance:
finalDistance = distance
bestPermutation = currentPermutation
        # Write permutation and distance to CSV
writer.writerow([currentPermutation,
        distance / (len(OrigIndexMapList)) / (len(OrigIndexMapList)),
permutation_entropy(currentPermutation, 2,1)])
listOfPermutationsPE = list(itertools.permutations(OrigIndexMapList, len(origList)))
print(f"Data written to {csv_file}")
```

Note: Parallel processing on high configuration machine was used as bigger arrays had millions of permutations

Appendix III Mathematical model for JChaosIndexmax given the size of data records

```
# coding: utf-8
#Author: Jui Keskar (Copyright)
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Given data points\newline
x_data = np.array([10, 9, 8, 7, 6, 5, 4, 3, 2])
y_data = np.array([7.1, 6.419753086, 5.75, 5.06122449, 4.388888889, 3.68,
3, 2.222222222, 1.5])
# Define a polynomial function (e.g., quadratic, cubic)
def polynomial(x, a, b, c, d):
    return a * x**3 + b * x**2 + c * x + d
# Perform curve fitting
params, _ = curve_fit(polynomial, x_data, y_data)
# Extract the coefficients
a, b, c, d = params
# Print the polynomial equation\newline
print(f"The ;polynomial ;equation ;is: y = {a:.4f}x^3 + {b:.4f}x^2 + {c:.4f}x + {d:.4f}")
# Plot the data points and the fitted curve
```

```
plt.scatter(x_data, y_data, label='Data points')
x_fit = np.linspace(min(x_data), max(x_data), 100)
y_fit = polynomial(x_fit, *params)
plt.plot(x_fit, y_fit, label='Fitted curve', color='red')
plt.xlabel('No. of records [n]')
plt.ylabel('JChaosIndex [max]')
plt.title('JChaosIndex [Max] vs. Dataset size')
plt.legend()
plt.show()
………………………………………..
```

Output upon execution: The polynomial equation is:

$y = 0.0007 *x^3 - 0.0182 *x^2 + 0.8317 *x - 0.1041$

Programs for bench-marking various random number generators

Sample Python program for randomization of data

```
#!/usr/bin/env python
# coding: utf-8
#Shreeram
#Author: Jui Keskar (Copyright)
import numpy as np
from itertools import permutations
from collections import Counter
import itertools
import random
import csv
origList = [11,12,13,14,15,16,17,18]
OrigIndexMapList = [0,1,2,3,4,5,6,7]
indexMapList=[]
finalList = [-1,-1,-1,-1,-1,-1,-1,-1]
def findindex(newpos, currentPerm):
    for k in range(len(currentPerm)):
        if newpos == currentPerm[k]:
            return k
    return -1

def permutation_entropy(time_series, m, tau):
    n = len(time_series)
ordinal_patterns = []
    # Extract ordinal patterns
    for i in range(n - (m - 1) * tau):
sub_seq = time_series[i:(i + m * tau):tau]
ordinal_pattern = tuple(np.argsort(sub_seq))
ordinal_patterns.append(ordinal_pattern)
    # Count patterns
pattern_counts = Counter(ordinal_patterns)
total_patterns = len(ordinal_patterns)
    # Calculate relative frequencies
pattern_frequencies = np.array(list(pattern_counts.values())) / total_patterns
    # Compute permutation entropy
    entropy = -np.sum(pattern_frequencies *
    np.log2(pattern_frequencies + np.finfo(float).eps))
    return entropy

for i in range(len(origList)):
num = -1
```

```
    j = -1
while(True):
num = int(random.random()*8)
        if num not in indexMapList:
            break
finalList[num] = origList[i]
indexMapList.append(num)
distance = 0
for j in range(len(origList)):
    distance = distance + abs(j-(findindex(OrigIndexMapList[j], indexMapList)))
    for n in range(j):
        if j-n > 0: #and n > 0:
            distance = distance + abs((j-n)
            -(findindex(OrigIndexMapList[j],
indexMapList)-findindex(OrigIndexMapList[n], indexMapList)))
    for n in range(len(indexMapList)-j):
        if j+n<len(indexMapList) and n>0:
            distance = distance + abs((+n)
            -(findindex(OrigIndexMapList[j+n],
indexMapList)-findindex(OrigIndexMapList[j], indexMapList)))
print("Original List is", origList)
print("Permutation_entropy is:",permutation_entropy(finalList, 3,1))
print("Randomized Configuration is:",finalList)
print("JChaosIndex =", distance / len(origList)/ len(origList))
…………………………………………………
```

Sample Java program for randomization of data

```
// Author: Jui Keskar (Copyright)
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
public class ListRandomizer {
    public static void main(String[] args) {
        List<Integer>origList = Arrays.asList(11, 12, 13, 14, 15, 16, 17, 18);
        List<Integer>origIndexMapList = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7);
        List<Integer>indexMapList = new ArrayList<>();
        List<Integer>finalList =
        new ArrayList<>(Arrays.asList(-1, -1, -1, -1, -1, -1, -1, -1));
        Random random = new Random();
        for (int i = 0; i<origList.size(); i++) {
            int num;
            do {
num = random.nextInt(8);
            } while (indexMapList.contains(num));
finalList.set(num, origList.get(i));
indexMapList.add(num);
        }
        double distance = 0;
        for (int j = 0; j <origList.size(); j++) {
            distance += Math.abs(j - findIndex(origIndexMapList.get(j), indexMapList));
            for (int n = 1; n <= j; n++) {
                distance += Math.abs((j - n)
                - (findIndex(origIndexMapList.get(j),
indexMapList) - findIndex(origIndexMapList.get(j - n),
indexMapList)));
```

```
            }
            for (int n = 1; j + n <indexMapList.size(); n++) {
                distance += Math.abs(n
                - (findIndex(origIndexMapList.get(j + n),
indexMapList) - findIndex(origIndexMapList.get(j),
indexMapList)));
            }
        }
System.out.println("Original List is " + origList);
System.out.println("Best Randomization is: " + finalList);
System.out.println("JChaosIndex = " + (distance / origList.size() / origList.size()));
    }

    private static int findIndex(int newpos, List<Integer>currentPerm) {
        for (int k = 0; k <currentPerm.size(); k++) {
            if (newpos == currentPerm.get(k)) {
                return k;
            }
        }
        return -1;
    }
}
```

…………………………………………….

Sample C program for randomization of data

```
// Online C compiler to run C program online
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 8
int findIndex(int newpos, int currentPerm[], int size) {
    for (int i = 0; i< size; i++) {
    if (newpos == currentPerm[i])
        return i;
    }
    return -1;
}

int main()
{
    int origList[SIZE] = {11, 12, 13, 14, 15, 16, 17, 18};
    int origIndexMapList[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7};
    int indexMapList[SIZE];
    int finalList[SIZE];
    int usedIndices[SIZE] = {0}; // To keep track of used indices
    double distance = 0.0;
srand(time(NULL)); // Seed the random number generator
// Initialize finalList to -1
    for (int i = 0; i< SIZE; i++) {
finalList[i] = -1;
    }
// Generate random positions and update finalList and indexMapList
    for (int i = 0; i< SIZE; i++) {
        int num;
        do {
num = rand() % SIZE;
```

```
        } while (usedIndices[num] == 1);

finalList[num] = origList[i];
indexMapList[i] = num;
usedIndices[num] = 1; // Mark this index as used
    }
// Calculate distances
    for (int j = 0; j < SIZE; j++) {
        distance += abs(j - findIndex(origIndexMapList[j], indexMapList, SIZE));
        for (int n = 1; n <= j; n++) {
            distance += abs((j - n) - (findIndex(origIndexMapList[j],
indexMapList, SIZE) - findIndex(origIndexMapList[n], indexMapList, SIZE)));
        }
        for (int n = 1; j + n < SIZE; n++) {
            distance += abs(n - (findIndex(origIndexMapList[j + n],
indexMapList, SIZE) - findIndex(origIndexMapList[j], indexMapList, SIZE)));
        }
    }

// Print the results
printf("Original List is: ");
    for (int i = 0; i< SIZE; i++) {
printf("\%d ", origList[i]);
    }
printf("\textbackslash n");
printf("Best Randomization is: ");
    for (int i = 0; i< SIZE; i++) {
printf("%d ", finalList[i]);
    }
printf("\textbackslash n");
printf("JChaosIndex = %f\n", distance / SIZE/SIZE);
    return 0;
……………………………………………
```

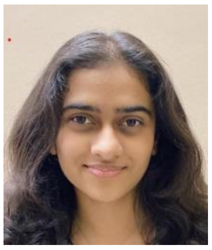
Sample C Sharp program for randomization of data

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        List<int>origList = new List<int> {11, 12, 13, 14, 15, 16, 17, 18};
        List<int>origIndexMapList = new List<int> {0, 1, 2, 3, 4, 5, 6, 7};
        List<int>indexMapList = new List<int>();
        List<int>finalList = new List<int>(new int[8]);
        Random random = new Random();
// Initialize finalList with -1
        for (int i = 0; i<finalList.Count; i++)
        {
finalList[i] = -1;
        }
// Generate random positions and update finalList and indexMapList
        for (int i = 0; i<origList.Count; i++)
        {
            int num;
            do
            {
num = random.Next(8); // Generates a random number between 0 and 7
            } while (finalList[num] != -1);

finalList[num] = origList[i];
indexMapList.Add(num);
        }
        double distance = 0;
// Calculate distances
        for (int j = 0; j <origList.Count; j++)
        {
            distance += Math.Abs(j - FindIndex(origIndexMapList[j], indexMapList));
            for (int n = 1; n <= j; n++)
            {
                distance += Math.Abs((j - n) - (FindIndex(origIndexMapList[j],
indexMapList) - FindIndex(origIndexMapList[n], indexMapList)));
            }
            for (int n = 1; j + n <indexMapList.Count; n++)
            {
                distance += Math.Abs(n - (FindIndex(origIndexMapList[j + n],
indexMapList) - FindIndex(origIndexMapList[j], indexMapList)));
            }
        }
Console.WriteLine("Original List is: " + string.Join(", ", origList));
Console.WriteLine("Best Randomization is: " + string.Join(", ", finalList));
Console.WriteLine("JChaosIndex = " + (distance / origList.Count/ origList.Count));
    }

    static int FindIndex(int newpos, List<int>currentPerm)
    {
        return currentPerm.IndexOf(newpos);
    }
}
```
……………………………………………

AUTHOR



Jui Keskar is a dedicated researcher and innovator with a strong focus on the intersection of computer science and data science, particularly as they apply to healthcare technology. Jui has developed wearable devices like PDCheck3D and JTremor3D, which leverage computation and data analytics. Devices may assist to monitor and assist in the management of Parkinson's Disease. Jui has been recognized internationally for her contributions to biomedical computation, earning prestigious awards such as the Prime Minister’s award for Innovation (India’s top civilian honour for under 18) and being featured in Forbes India’s 30-under-30 Special Mention. She also won two international awards at the International Science and Engineering Fair, conducted by Society for Science, USA. Jui has a patent pending in the US. With a solid academic foundation in Physics, Mathematics, and Computer Science from the International Baccalaureate program, and participation in elite programs like MIT’s Research Science Institute, Jui continues to march towards her mission of leveraging technology for the betterment of patients.