Proposal of Selection Method for Evolutionary Algorithms based on Logarithmic Scale ponderation

By Alejandro Ramos @arhcoder

"Logarithmic Random Ponderated Selector" **(LRPS)** is an algorithm whose purposee is to **receive a list of objects and select one of them randomly,** with the bias that the further to the beginning of the list each one is, the more likely it is to be selected; with a **weighting factor given by the shape of the logarithmic-exponential curve.**

Why?

When working with models that require selection mechanisms, it is common to use algorithms based on chance, in which –metaphorically– a roulette defines which of the ***n*** objects will be chosen in each turn. These methods are useful when it is preferred to have the same probability of selection for all elements in a set, however, this may not be the intention.

The idea behind this proposal is born from the analysis of phenomena for which there are events with a greater frequency of being present, but for which the factor that determines these magnitude is not known. Being concrete, ***LRPS*** originates from my own attempt to replicate the composition of music through genetic algorithms with randomness, being clear the fact that depending on the style of music, genre, era, etc., certain composition patterns are more common; for example: 4/4 time signature, long or short notes, simple or complex harmonies, etc. Popular music tends to have with more frequency the 4/4 signature, and only major-minor chords, slow voice melodies, relaxed rhythms, among others; It is then that in order to try to replicate the composition of popular music, we could consider these characteristics as the most common, but without neglecting the fact that it is possible to have more variety of decision.

Note: The above mentioned project of music composition with genetic algorithms can be found at: **https://github.com/arhcoder/M.I.A**.

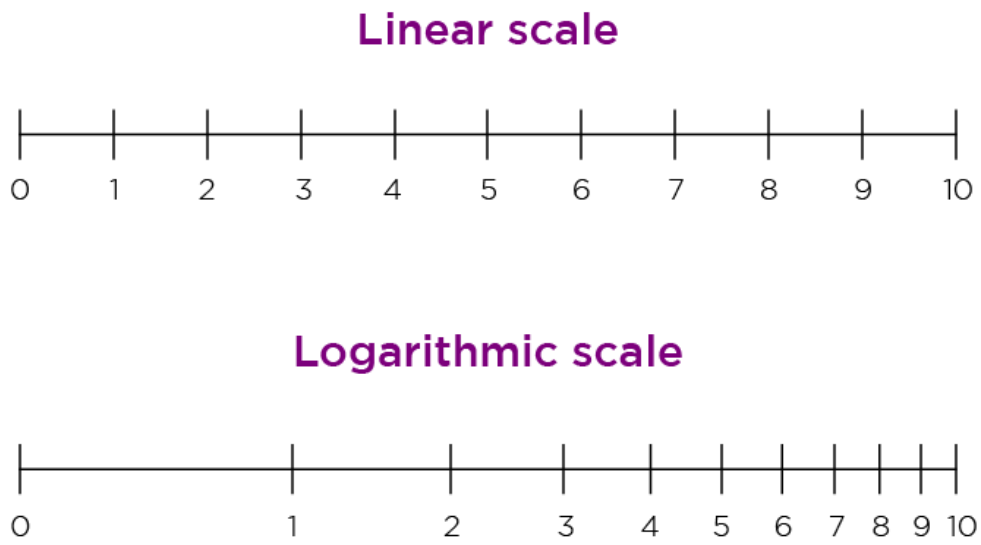
How?

If for decision-making in a model there are biases and weights without a number that represents their magnitude; this algorithm proposal can be useful. **For example:**

Imagine that you want to decide the flavor of an ice cream to eat, based on a list that contains your favorite flavors:

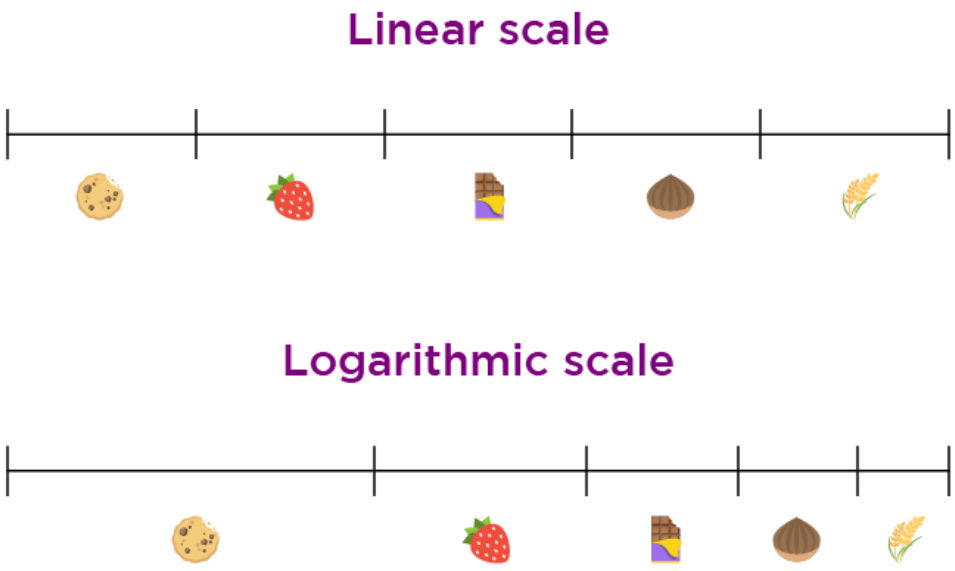
- **1st: Cookies n' Cream.**
- **2nd: Strawberry.**
- **3rd: Chocolate.**
- **4th: Butter Pecan.**
- **5th: Pistachio.**

In the model it has been decided that Cookies n' cream flavor is the favorite, so its selection should be more frequent, followed by strawberry, chocolate, etc. The important issue is that there is no a tangible number that allows determining this importance bias with respect to the other flavors, it is simply known that flavors are preferred more as they are higher on the top. The logarithmic scale can provide a growth ratio based on its exponential mathematical nature:

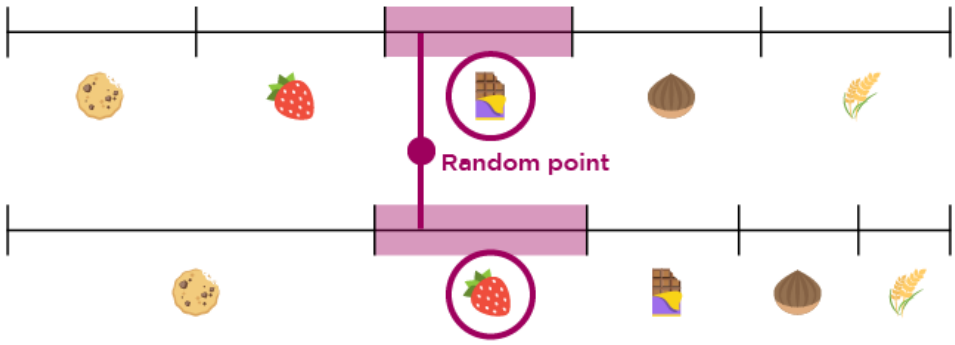


A loose representation of what is the difference between linear and logarithmic scales looks like is shown in the image above; if we take it as an example, we could understand this decision algorithm as placing a point randomly in any of the two scales, where the space between two divisions represents an object to select. For the case of the linear scale, the probability of finding the object between **0 and 1** is the same as that of the object between **1 and 2,** or **2 and 3.** On the other hand, with the logarithmic scale, the probability of falling into the object between **0 and 1** is very different from that of the object between **5 and 6.**

Using real data for the example of choosing an ice cream flavor, we would have the following scales comparison:



This is what the ***LRPS*** algorithm scale looks compared to a roulette algorithm. If we throw a random point in this space; we would obtain the decisions for both types of algorithms, which would be:



- ***Chocolate*: for a roulette algorithm.**
- ***Strawberry*: for the LRPS algorithm.**

As the space of the strawberry is larger than the chocolate ones, it is more likely that we will end up eating strawberry instead of chocolate.

Math

The way to build a logarithmic scale adapted to the ***n*** quantity of objects to be selected is possible through a ***P* set of points on a Cartesian axis** in which each point delimits the selection space of each of the ***n*** elements of the list, using the following formula:

$$\mathbb{P} = \{\ \boldsymbol{p_i} : \boldsymbol{p_i} = (n \log_{n+1}(i+1), 0),$$

$$n \in \mathbb{Z},\ n > 1,$$

$$i\ \in \mathbb{Z},\ \ 1 \leq i \leq n\ \ \};$$

$$\boldsymbol{p_i}\ is\ (x, y)\ \boldsymbol{coordinate};$$

That is, given the same case of building the scale on a Cartesian axis, with ***n*** number of spaces (objects) and therefore ***n*** number of division points, the distance between each point ***i*** and the ***origin*** (coordinate (0, 0)) is given by the formula:

$$\boldsymbol{d_i = n \log_{n+1}(i+1)};$$

If it is wanted to find the coordinate of point 1, for the example of five ice cream flavors (with n = 5), the coordinate would be:

$$p_1 = (5 \log_{5+1}(1+1),\ 0);$$

$$p_1 = (5 \log_6(2),\ 0);$$

$$p_1 = (1.934,\ 0);$$

So, to find all the points needed to build the logarithmic scale of five objects:

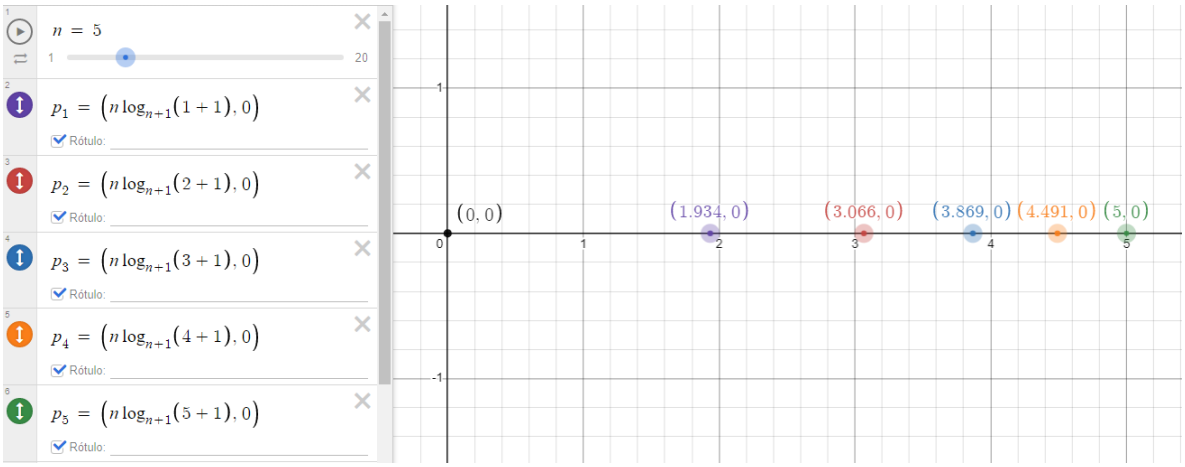
$$p_1 = (5 \log_{5+1}(1+1),\ 0);$$

$$p_2 = (5 \log_{5+1}(2+1),\ 0);$$

$$p_3 = (5 \log_{5+1}(3+1),\ 0);$$

$$p_4 = (5 \log_{5+1}(4+1),\ 0);$$

$$p_5 = (5 \log_{5+1}(5+1),\ 0);$$



Note: The use of the Cartesian axis is unnecessary for the algorithm, since it is only necessary to know the distance between the origin and any other point on the scale; so only the distance formula will be necessary: $d_i = n\log_{n+1}(i+1)$.

Algorithm

The pseudocode of the selection algorithm is shown below:

```
lrps(list objects = [obj1, obj2, obj3, obj4, obj5]):

    # n = amount of objects:
    n = objects.length

    # If there are not objects:
    if n == 0:
        return "Objects list empty :c"

    # If list has only 1 object, returns the obj1:
    if n == 1:
        return obj1

    # Gets a random decimal number between [0, n]:
    randomPoint = random_between(0, n)

    # Loop between 1 and n:
    for i = 1 to n:
        if randomPoint <= (n * log_n+1(i+1)):
            return obj_i
                                        @arhcoder
```

A list of objects is received as input from which the value of ***n*** is obtained as the number of objects in the selection. A couple of stopping cases occur when the list is empty or contains just one element, otherwise proceeds with the rest of the algorithm, where a random decimal number between 0 and ***n*** is obtained, and a loop for 1 to ***n*** is started.

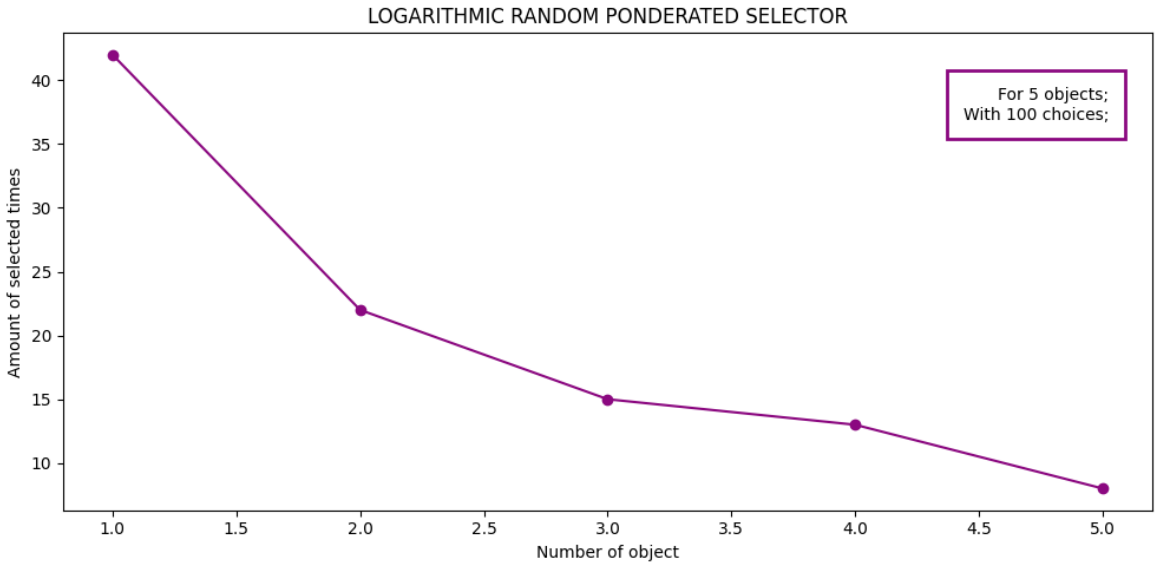
Within the loop, the distance from the ***"origin"*** to the first point of the logarithmic scale **(p_1)** is calculated using the same analogy of a Cartesian plane, using the distance formula described above. If the random point is between 0 and the distance of the first point of the scale; that is, if the random point was before p_1 **(random point <= distance to p_1),** then the decision is made to take the object from these space (object number 1), in case the random point is above this first scale point, it continues with the cycle until finding the space in which the random point is.

LRPS Algorithm has **linear complexity, O(n)** with ***"Big O notation",*** remembering that ***n*** is the number of objects in the initial decision list.

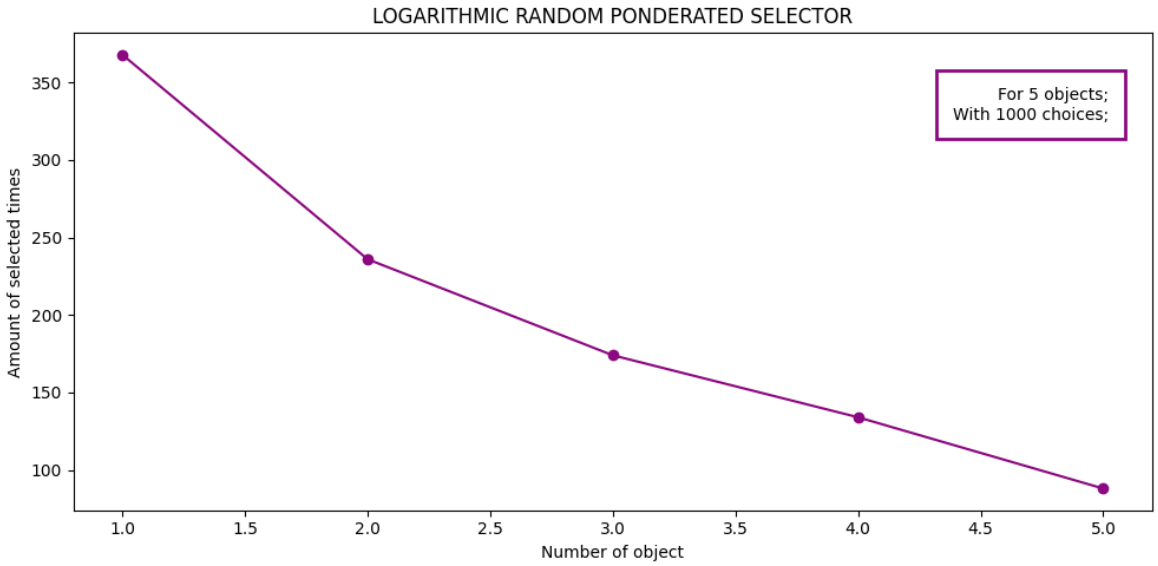
Experiments

Doing count tests for the decisions made by the algorithm in order to check the bias in the choices, **repeating the selection making a certain number of times and counting how many times each of the objects was chosen, was graphed:**

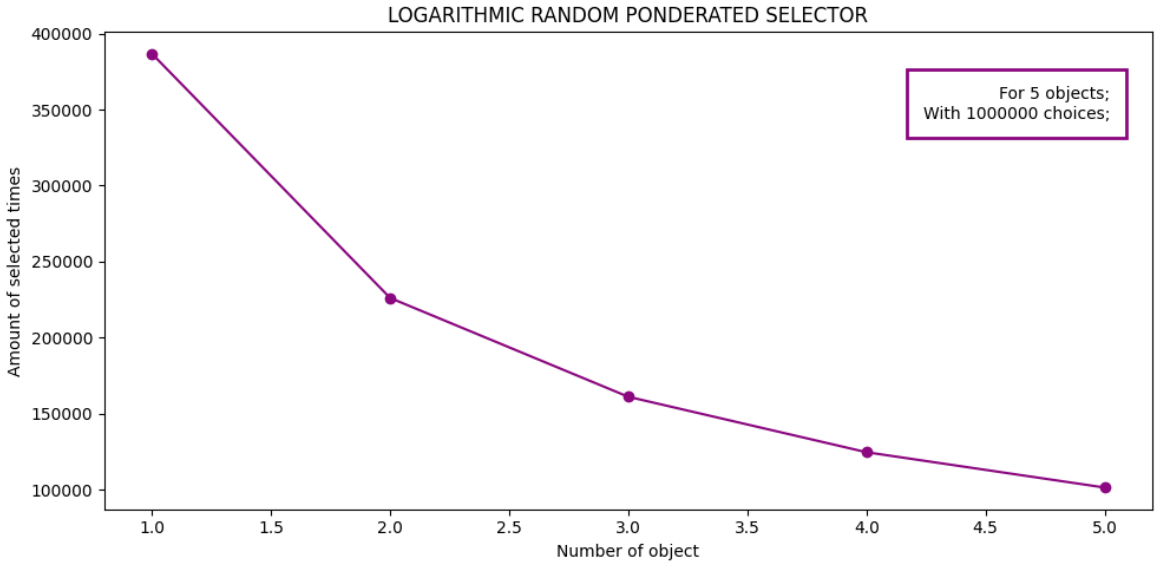
1. Having 5 objects and 100 repetitions:



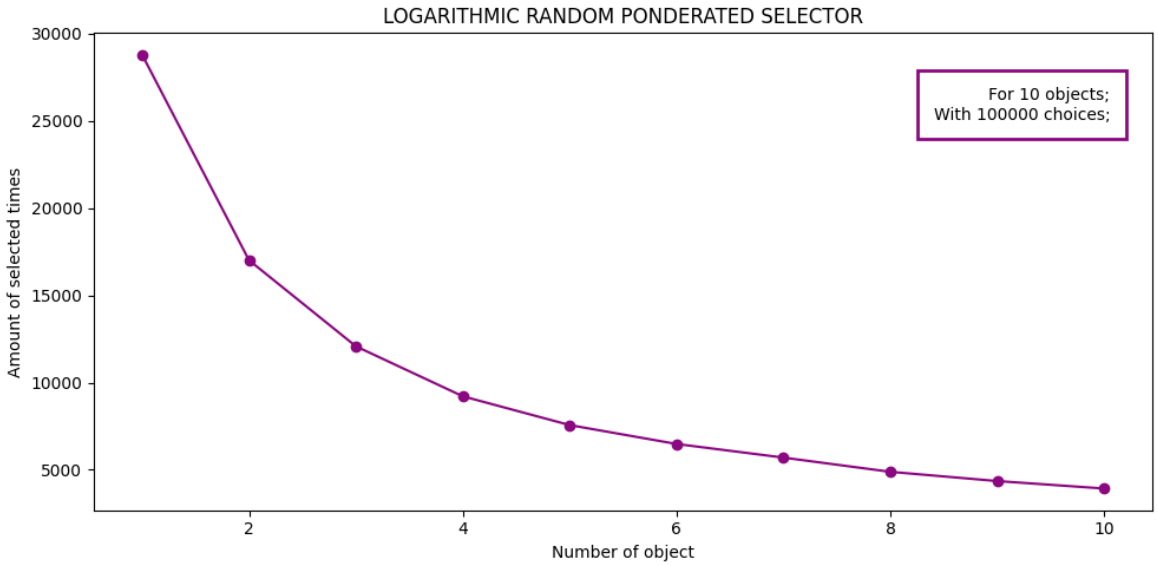
2. Having 5 objects and 1,000 repetitions:



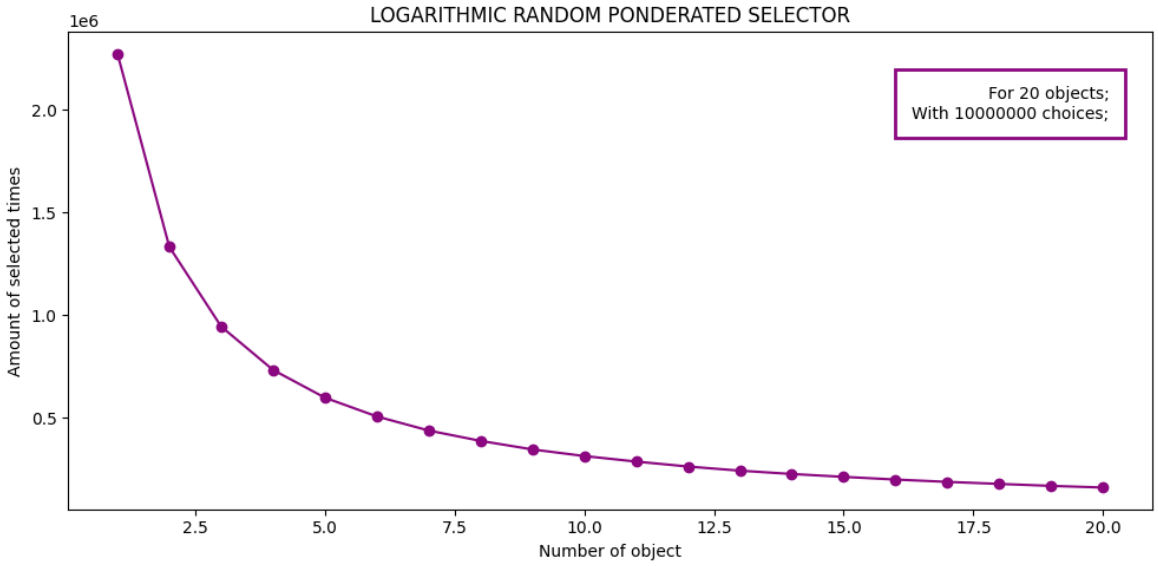
3. Having 5 objects and 1,000,000 repetitions:



4. Having 10 objects and 100,000 repetitions:



5. Having 20 objects and 10,000,000 repetitions:



It can be verified that the shape of a logarithm curve is obtained.

Note: For random numbers generation, the ***random*** function of the **64-bit Python v.3.11 core** was used. The counting-graphing script as well as the LRPS algorithm can be found on ***this repository***.

Conclusions

This is just a proposal of algorithm, it may be useful in; for example, decisions making at **genetic algorithms with elitism.** If we have a list of individuals in a population, ordered by their fitness, and we want to make decisions that affect the best individuals more frequently, but we don't want to lose the chance of considering the less fit individuals, this can be a good selection method option. It can also be an interesting proposal to find the optimal point between **exploration and exploitation in metaheuristics.**

In the event that for a specific problem it is desired that two or more objects have the same selection probability, it can be considered –for this implementation– that these objects are included in a data structure that contains them as **ONE** single object, so that if the algorithm chooses them, it will throw the objects as a whole, later and by means of a roulette-type decision (such as the linear scale) one of these can be chosen.

Check the algorithm code from the **main repository** at **https://github.com/arhcoder/LRPS-Algorithm**, any **collaboration and/or improvement proposal (pull request)** will be welcome, just like a **GitHub star.**

Write to me at **arhcoder@gmail.com** for any query, or if this contribution was useful to you in any application. Also look for me on social media as **@arhcoder**.

Thanks for reading 😀

Alejandro Ramos, @arhcoder

❤