Introduction:

Hello everyone, I'm working on an automation task for a research project at my institute. It is a problem that regularly occurs in DNA sequence analysis.

Problem statement:

I have a dataset of DNA sequences, each with a length of 10 bases. My goal is to select a subset of N sequences (where N >= 6) in such a way that all the four bases (A, C, T and G) are distributed fairly balanced across all the 10 base positions.

Moreover, the dataset actually have two columns of sequences, which we can call column A and B, and we want to select a subset such that the criterion is satisfied for both columns separately.

Data representation:

The dataset is structured as follows:

```
index_name:     col_A:          col_B:
SI-TS-A1        AATTTCGGGT      GAGGAGAGAG
SI-TS-A2        CTACTGAATT      AGCGCTAGCG
...             ...             ...
```

Evaluating a solution:

A solution candidate (here for N=6) would for example look like this (skipping the index_name column):

```
col_A:          col_B:
ACCGACTGAT      GAACCCAGGA
CAGGACTCTA      CCATCCTACG
CAAGCGTTAG      CGCATTCGCG
GTTAAACCGC      CAAGCCGGCT
TATGTTGTGC      ATTAAGCCAT
TACATATGCT      TCACGATTAC
```

To evaluate, we look at column A and B as two distinct sets that each need to satisfy the base distribution criterion.

Let's take column A and go through how we evaluate: First we count up the number of bases of each base type in each position, as exemplified by the following table:

Position	1	2	3	4	5	6	7	8	9	10
Base A	1	4	1	2	3	2	0	0	2	1
Base C	2	1	2	0	1	2	1	2	1	2
Base G	1	0	1	4	0	1	1	2	2	1
Base T	2	1	2	0	2	1	4	2	1	2

By dividing all the values by N, we get the fractional representations, as displayed below:

```
Position |  0   |  1   |  2   |  3   |  4   |  5   |  6   |  7   |  8   |  9
======================================================================================
Base A   | 0.17 | 0.67 | 0.17 | 0.33 | 0.50 | 0.33 | 0.00 | 0.00 | 0.33 | 0.17
Base C   | 0.33 | 0.17 | 0.33 | 0.00 | 0.17 | 0.33 | 0.17 | 0.33 | 0.17 | 0.33
Base G   | 0.17 | 0.00 | 0.17 | 0.67 | 0.00 | 0.17 | 0.17 | 0.33 | 0.33 | 0.17
Base T   | 0.33 | 0.17 | 0.33 | 0.00 | 0.33 | 0.17 | 0.67 | 0.33 | 0.17 | 0.33
```

The distribution criterion is then the following:

For each of the 10 positions, we want at least 12.5% and no more than 40% representation for each base. We also require that the median across all positions (for each base) should be in the same interval.

The same is then repeated for column B. If both columns pass, then the solution is valid.

We can see that the above solution canditate does not pass the test, as several elements are outside the desired interval.

What I've tried:

I've attempted various approaches:

- Random search works fine for large N. Starts taking a long time around $N \approx 25$.
- Evolutionary algorithm, as well as an evolutionary algorithm with some steps of greedy search in between every new generation. This works well until we get down to about $N \approx 12$, then it starts struggling.
- Evolutionary algorithm with greedy search. Works a little bit better than just the evolutionary algorithm alone. Was able to find a solution for $N = 11$, which is the lowest N I have found a solution for thus far.
- Tried initializing individuals by maximizing the Levenshtein distance between the components (sequences), but did not help much.

I am not sure how to improve this further (except that I could probably speed it up by using more built in functions from the `Deap` library). Could be that I just need to find better hyperparameters, but I don't have that much experience with these types of algorithms, so I am kind of stuck.

More details:

- An **individual** in the population is one solution candidate, represented by a set of N unique indices, each index corresponding to one row in the dataset (meaning a pair of sequences).
- The **fitness function** is defined as the distance to the desired interval.
  - I did this in the following way:
  - For each position for each base (ref. the above table), if the element is outside of the desired interval, the distance from the desired interval is calculated, and then summed up for all into a final scalar value.
  - To also consider the the median criteria in the fitness function, the same as above is done (taking the distance from the median to it's corresponding desired interval), and this is simply added this (with a custom weight parameter) to the previously discussed fitness value.

  - The fitness is thus defined such that we are **minimizing** it.
- **Parents** are selected probabilistically, where we weigh the probability of selection using the fitnesses.
- **Offspring** are created using a is done using single-point crossover recombination function, and mutation is then applied.
- **Elitism** is implemented by always letting the top 1. individual survive to the next generation,
- The **greedy search** part is done by iterating over all individuals in the population, where for each one, we select a random index in the individual and replace this with a new random one. If the fitness of this new individual is better, we replace it, else we discard it. Repeat this a set amount of times per individual (this is controlled by the `improvement_steps` variable).
- If nothing else works, we can loosen the criteria by ignoring the last two positions. This is implemented using the `ignore_last_k_positions` parameter.

Running the code:

The code is written in Python, using the `Deap` library made for evolutionary algorithms.

Probably best to install the dependencies with Anaconda, as that's what I did.

You can create an environment called `envname` from the `.yml` file with the following command:

```
conda env create --name envname --file environment.yml
```

Then activate the environment by

```
conda activate envname
```

and finally run the code with

```
python3 main.py
```