Background Information

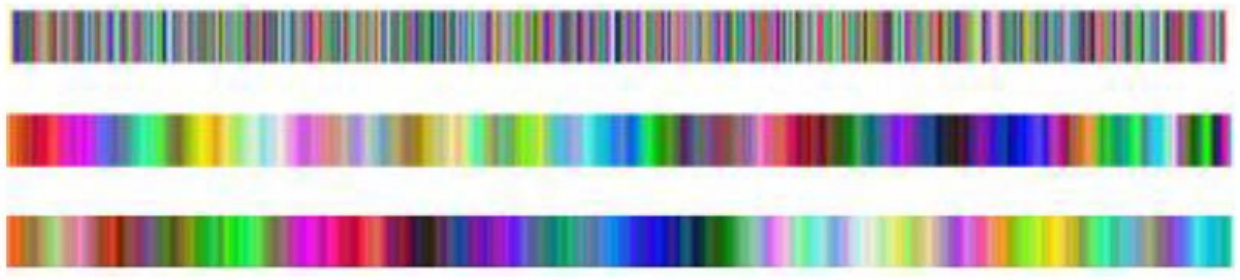
● "Welcome! You're our new employee, right? You should already know that our company specialises in software for editing and managing large collections of photographs. Some of our valued customers want to be able to browse their photograph collection in a visually pleasing fashion with similar coloured photographs following each other. I'm not really sure what that means but you can interpret that as having as little difference as possible in the dominant colour of each consecutive photograph. You have a month to prototype some ideas and report back to me." You start thinking about this task and soon some things become clear to you.

● A colour may be represented by three values that define the levels of red, green and blue (RGB).

● For the difference between two colours, you need some way to measure the distance. Each colour has RGB coordinates so you will use the *Euclidean* distance for your prototype. Since each colour is represented by 3 values, you will need the Euclidean distance for 3 dimensions. Here is the formula.

● Your problem now boils down to organising a permutation of elements whilst minimising the sum of the distances between adjacent elements. This looks suspiciously like the Traveling Salesman Problem you saw in ITNPBD8! Actually, it is a little bit simpler since you only need to consider a sequence and not a cycle. That is you do not need to return to the starting point.

● Find below 3 example solutions visualised. The first example corresponds to a random permutation of colours and the following two are permutations arranged (optimised) with different algorithms.



What you need to do

● You can use either Python or Java to develop your programming assignment.

● You are given a **file with 1000 colours** split into their RGB coordinates. Write some code to read the file. Use a subset of the colours or all them, as needed, to test different approaches. For example, you can run your algorithms with the first 10 or 100 colours.

● Implement the following 4 search algorithms to find the permutation with an objective function which minimises the sum of the distance between adjacent elements of the permutation:

1. **Simple random search**. Generate a fixed number of solutions at random within a loop.
2. **A hill-climber**. Start from a randomly generated solution and implement a first-improvement local search method. You have alternative options for the move operator or neighbourhood. The simplest move operator is simply *swapping* two colours. For example, given solution [1,2,3,4,5,6], then [1,**4**,3,**2**,5,6] or [1,2,**6**,4,5,**3**] would be in its neighbourhood. Another better operator consists in *inverting* the order of the colours between any two colours selected at random. This corresponds to the neighbourhood of the 2-opt move in the TSP literature. For example, given solution [1,2,3,4,5,6], then [**5**,**4**,**3**,**2**,**1**,6] or [1,**4**,**3**,**2**,5,6] would be in its neighbourhood.
3. **An iterated local search**. This algorithm will be based on the hill-climber you implement in part 1. As indicated in the lectures, you need to implement an additional loop and a *perturbation* operator. For example, doing two consecutive swaps or two inversions on the solution.
4. **An evolutionary algorithm**. This algorithm should generate a population of individuals and implement a mutation and a recombination operator. As discussed in class you can implement a steady-state algorithm with tournament selection.

● The tasks are the following:

1. Describe your choices of move operators and termination criteria.

2. Apply your algorithm to 3 problem sizes: 10, 100 and 1000.

3. Conduct a total of 20 runs for each algorithm and problem size and compute the average and standard deviation of the best fitness value obtained at the end of the run. You can add this results in a table. You can gain additional marks if you go beyond average/standard deviation and produce boxplots displaying the distribution of fitness values across the 20 runs for the 4 algorithms and the 3 problem sizes.

4. In order to visualise and compare the dynamics of the algorithms, you may want to generate line plots with the trace of selected runs, showing the best fitness found so far at each iteration of the algorithm. You may choose to visualise the best run for each algorithm.

5. Generate plots of the colour permutations of the best-found solutions for problem sizes 100 and 1000, using the provided source code for visualisation. Since you have 4 algorithms and 2 problem sizes, a total of 8 solutions should appear.