**COEN 352 Final Project** (program + report: due **Dec 3rd**)

*~ more information and hints will be provided in the lectures ~*

Option A. Retrieve All Stored Vectors within a Threshold Distance of a Cue Vector

You are provided with a large number (T) of vectors of equal size (S). Each vector represents a grey-scale image of L x W pixels. Hence, every vector has LW values, with each value in the range [0,255]. You are free to store the set of vectors in any way that would optimize the performance of your program. Assume distance (D) between any two vectors is the normalized city-block distance between them. For example, if **V**1=(1,2,3) and **V**2=(2,3,4) then D(V1,V2)=(|1-2|+|2-3|+|3-4|)/3=1. Given an unseen cue vector (**V**c), your problem is to design, implement and test algorithm(s) and data structure(s) that would return all stored vectors within a user-provided distance of any user-provided Vc. The time and space complexity of your solution must (a) scale-up as well as possible (approximately, linearly or better) with both the size of vectors (LW) and the number of stored vectors (T); (b) be practically useable. The data needed for the development and testing of your program will be provided by the instructor via moodle. Exact instructions on submission will be provided (via moodle) soon: prepare to demo your program and provide a brief (3-5 page) written report describing its design (e.g., using a flow chart or pseudo-code) and the results of its operation.

Option B. Devise a Meta-Algorithm that Optimizes a Sorting Algorithm

You are provided with a set of data samples belonging to different input data distributions: near-sorted, uniformly random, near-inversely-sorted, normally distributed (with different means and standard deviations). You are provided with a generic representation of a sorting algorithm, methods for generating one (using the generic representation) as well as methods for mutating it and assessing its quality. For example, assume a list L contains elements L[0], L[1], .. , L[N]. Given that instruction *swap*(l,m) compares L(l) to L(m) and swaps these elements iff l<m and L(I)>L(m), a sequence of swap instructions can represent a sorting algorithm (not necessarily a great one). An example of such sorting routines is: (0,N),(1,N-1),(2,N-2), .., (N/2,N/2 +1). Such an algorithm would perfectly sort a perfectly inversely sorted list. In terms of mutation: one could simply exchange any two – different – swaps in the sequence, resulting in a slightly different sorting algorithm. A mutated ‘child’ sorting algorithm would be kept iff it is of better quality than its ‘parent’. The quality of a sorting algorithm has two aspects. One is *efficiency*, which is the total number of comparisons, equal to the number of swap instructions in the sequence, and *effectiveness*, which must reflect how un-perfectly sorted the resulting lists are. Hence, for effectiveness, one would apply the sorting algorithm to a set of input lists, then generate a number (say, average number of inversions) that reflect how far the output lists are from perfect order. Both effectiveness and efficiency must contribute (you decide how) to the computation of quality. Naturally, your sorting algorithms must produce sorted or near-sorted output lists and do so in linear time or better. The data needed for the development and testing of your program will be provided by the instructor via moodle. Exact instructions on submission will be provided (via moodle) soon: prepare to demo your program and provide a brief (3-5 page) written report describing its design (e.g., using a flow chart or pseudo-code) and the results of its operation.

References (on moodle)