CS1112 Lab Exercise 14

1 Insertion Sort

Download and read function `InsertionSort` from the Exercises page. Write another function that implements the insertion algorithm *in-place* and *in-line*. Do not count the number of comparisons and swaps.

```
function x = InsertionSortInplace(x)
% Sort x in ascending order using the insertion sort algorithm.
% Sort in-place, i.e., without creating another vector.
% Perform the insert process in-line, i.e., no subfunction.
% x is a 1-d array of numbers.
```

2 Writing efficient code

1. Download the script `LargestTriangle` from the Exercises page. The script (also shown below) is a first attempt at finding the largest triangle that can be formed from n points on a unit circle. Add code (`tic`, `toc`) to the script to determine how long it takes to find the answer for $n = 100, 150, 200$. Store the results (time) in vector `t1` such that `t1(i)` corresponds to $n(i)$, $i = 1, 2, 3$.

```
for n=100:50:200
    theta = rand(n,1)*2*pi;  % Angle of random pts on the unit circle
    % Determine how long it takes to compute the largest possible triangle obtained by
    % selecting vertices from the points represented by theta
    A = 0;
    for i=1:n
       for j=1:n
          for k=1:n
             % theta --> Cartesian
             c1 = cos(theta(i)); s1 = sin(theta(i));
             c2 = cos(theta(j)); s2 = sin(theta(j));
             c3 = cos(theta(k)); s3 = sin(theta(k));
             % Area using Heron's Formula
             a = sqrt((c1-c2)^2 + (s1-s2)^2);
             b = sqrt((c1-c3)^2 + (s1-s3)^2);
             c = sqrt((c2-c3)^2 + (s2-s3)^2);
             s = (a+b+c)/2; Aijk = sqrt((s-a)*(s-b)*(s-c)*s); A = max(A,Aijk);
          end
       end
    end
end
```

2. We now start to make the computation more efficient. *Append* the script rather than modify directly—copy and paste your code from Part 1 to Part 2 of the script and make the modification in Part 2.

Notice that there are several levels of inefficiency. The area for each different triangle is computed 6 times. Modify the loop ranges to eliminate this redundancy. Also, there are a lot of redundant sine and cosine evaluations. Address this issue by moving the `c1`, `s1`, `c2` and `s2` assignments. In Part 2, store the time taken to do the computation in vector `t2` such that `t2(i)` corresponds to $n(i)$. How much speed-up did you get? [ ]

Even with the change in *where* we compute `c1`, `s1`, `c2` and `s2`, we are still doing more sine and cosine evaluations than necessary—given n values of `theta` we should only need to make n sine evaluations and n cosine evaluations. This suggests that we can reduce the time further by *precomputing* the sine and cosine of `theta`. We will combine this insight with another improvement in Part 3 below.

3. There is additional redundancy associated with the side length computations `a`, `b`, and `c`. In Part 3, eliminate this redundancy by precomputing an $n \times n$ array `D` with the property that D(i,j) is the distance from point (cos(theta(i)),sin(theta(i))) to point (cos(theta(j)),sin(theta(j))). Note that you only need the "upper half" of `D` since `D`(i,j) = `D`(j,i). Store the time taken to do the computation in vector `t3` such that `t3(i)` corresponds to $n(i)$.

4. Draw a plot of the computation time (three graphs of time vs. n). Also show in a table the ratio of `t1` to `t3` for all n.

5. What is the expected computation time for the three methods for $n = 1000$? [ ]

Final note. The speed-up that we get isn't all "free." The speed-up that we gain from precomputation has a cost in computer memory—from version 1 to version 3, the major memory requirement increases from n (length of `theta`) to n^2 (dimension of `D`). The problem at hand, the language, and the hardware are all considerations in the trade-off between speed and memory.