For this project you have two options:

- Coding option: Implement *polygon triangulation*.
- Experimental option: Study *polygon decomposition* with CGAL.

The options are described in detail below.

### 1. Coding option - Polygon Triangulation

**Problem specification.** Implement the polygon triangulation algorithm discussed in Chapter 3 of the textbook, i.e. a plane sweep to decompose an arbitrary simple polygon into monotone pieces, followed by a stack-based algorithm to triangulate each piece. For the second step you can use either the direct approach in the text or the variation involving a decomposition into “mountains” that was presented in lecture.

The evaluation of your project will include testing your program on several polygons. Therefore, it is very important to conform to the following input and output format specifications.

**Input.** Your program will read a list of points from standard input (stdin). Each line of input will contain the coordinates \( x \ y \) of one point, separated by whitespace characters (e.g. space, tab). The points will be the vertices of a simple polygon \( P \) in counterclockwise order. The first vertex in the list will have the largest \( y \) coordinate and will be leftmost among such vertices in case of a tie. For example, the following input describes a square inscribed in the unit circle:

\[
0 \ 1 \\
-1 \ 0 \\
0 \ -1 \\
1 \ 0
\]

**Output.** Write a list of triangles to standard output (stdout), one per line, defining a triangulation of \( P \). A triangle is described by three integers, representing the zero-based indices of its vertices in the input list. On each line of output, the triangle vertices will appear in counterclockwise order, starting from the one with smallest index. For example, the following represents one possible triangulation of the polygon described above:

\[
0 \ 1 \ 2 \\
0 \ 2 \ 3
\]

In this sample output, the first line describes the triangle with vertices \((0, 1), (-1, 0),\) and \((0, -1)\).
Implementation advice. The textbook assumes that the input polygon is represented by a doubly-connected edge list (DCEL), but implementing a full DCEL is not necessary for this assignment. You may instead want to work with simple arrays of vertices stored in CCW order, breaking the algorithm into additional steps to maintain and update these structures, i.e.

1. Find monotone decomposition diagonals using a plane sweep
2. Use diagonals to break the input list into vertex lists for the monotone pieces
3. Sort the vertex list for each monotone piece by y-coordinate and triangulate.

Step (2) involves some nontrivial algorithm work that is not discussed at all in the text. Plan to devote some time to this point.

Each vertex should be stored in a structure that includes its coordinates and its index in the input list, since these indices are needed for the output format.

Language choice. The same programming language policies apply as in Project 1. Contact me for approval if necessary.

Built-in functions and libraries. Your code may not use any built-in computational geometry functions.

Your code may use built-in data structures such as arrays, linked lists, hashes, search trees, stacks, or queues, if such structures are provided by the language or CAS you choose. If you want to use a library that implements any of these basic data structures, contact me for approval.

What to submit. Write a report on your implementation process. Start by briefly describing the triangulation problem and the algorithm. Then focus on the design decisions that were involved in your implementation.

Create test cases for your code that demonstrate its correctness for various types of input (a convex polygon, a monotone polygon, a polygon requiring several monotone pieces, etc.). Display these test polygons and the triangulations produced by your program in the report.

Submit the report, source code, and test cases by email (ddumas@math.uic.edu), following the standards for coding option and source code submissions from the description of Project 1.

2. Experimental option - Polygon decomposition

Experiment 1: Random polygons. How should one generate a “random” simple polygon with \( n \) vertices? One approach is to repeatedly generate lists of \( n \) random points (in a certain planar region) until the list forms the CCW boundary of a simple polygon. CGAL provides a more sophisticated polygon generator, \texttt{CGAL::random_polygon_2()}. Compare these two methods by generating polygons with various numbers of vertices. Sample generator programs are included in the source code listings at the end of this document.

It should become apparent that the naive algorithm is not useful for polygons with more than a dozen or so vertices. Based on your results, do you have a guess for its expected running time as a function of \( n \)?

Read the CGAL documentation to determine what method \texttt{CGAL::random_polygon_2()} uses and the expected asymptotic behavior of the running time \( T(n) \) of this algorithm as a function of \( n \). Include a brief description in your report. Do you observe the expected
running time in your experiments? It may be useful to graph \( \log(T(n)) \) as a function of \( \log(n) \); in such a graph, a running time function that behaves like \( Cn^\alpha \) would appear as a line of slope \( \alpha \).

Extra credit opportunity: The naive random polygon generator and the CGAL generator induce different probability distributions on the set of all simple \( n \)-gons in \([-1,1] \times [-1,1]\). Can you observe any differences between the polygons generated by these two algorithms? If so, can you justify your observation theoretically, in terms of the generator algorithms?

**Experiment 2: Monotone decomposition.** CGAL implements a plane sweep algorithm to decompose a given CCW-oriented simple polygon into monotone pieces with the function `CGAL::y_monotone_partition_2()`. The example program `y_monotone_partition_2` in `examples/Partition_2` demonstrates its use, and a modified version of this program which reads input from `stdin` is shown in the source code listings at the end of this document.

We analyzed this plane sweep algorithm in class and obtained a bound of \( O(n \log n) \) for its running time. Is this the behavior that one sees in practice, or does this represent a worst-case that is very unlikely? Test the CGAL implementation with random polygon input and analyze the results. Can you distinguish linear growth from \( n \log(n) \)?

Be careful to measure only the time used by the monotone decomposition, and not the time necessary to generate the random polygons used as input.

Extra credit opportunities:

- Generate monotone polygons and use these to test for output-sensitive behavior in the partition algorithm.
- Generate polygons that require many pieces in the monotone decomposition and use these to test for output-sensitive behavior in the partition algorithm.
- For a given value of \( n \), make a histogram showing the number of pieces in the monotone decomposition for a large number of random \( n \)-gons. Comment on the results.

**Experiment 3: Convex decomposition.** A *convex decomposition* of a polygon \( P \) is a set of diagonals that cut \( P \) into convex pieces. For example, a triangulation is a convex decomposition. A convex decomposition of \( P \) is *optimal* if there does not exist a convex decomposition with fewer pieces.

CGAL provides an algorithm to find an optimal convex decomposition of a given polygon with the function `CGAL::optimal_convex_partition_2()`. The example program `optimal_convex_partition_2` in `examples/Partition_2` demonstrates its use, and a modified version of this program which reads input from `stdin` is shown in the source code listings at the end of this document.

Read the CGAL documentation to determine the worst-case running time of this algorithm as a function of \( n \). Experiment with random polygonal input and compare your results to this worst-case bound. Try to find a real number \( \alpha \) so that the running time behaves like \( Cn^\alpha \) for some \( C > 0 \).

**Guidelines for data collection.** When examining the running time of an algorithm as a function of the input size \( n \), use several values of \( n \) distributed in either an arithmetic progression (e.g. 100, 200, 300) or approximate geometric progression (e.g. 2, 4, 8, 16 or 10, 20, 50, 100, 200, 500). For arithmetic progressions, the total range of values (i.e. \( n_{\text{max}} - n_{\text{min}} \)) should be much larger than the smallest value \( n_{\text{min}} \). Generally speaking, you
should use 8 or more values of $n$. Start with a value of $n$ that is large enough to exceed the overhead of initializing the data structures; failure to do so might appear as running times that are almost the same for the first few values of $n$.

**What to submit.** Write a report that describes your experiments, presents the data, and interprets the results. The report should have one section for each of the three experiments described above. It is not necessary to include tables listing all of the raw data; rather, include graphs and tables that summarize enough of the data collected to justify your conclusions.

Submit the report, source code, and test cases by email (ddumas@math.uic.edu), following the standards for experimental option and source code submissions from the description of Project 1. For the report, a single PDF file is preferred.