

Simon Chaisouang

[shchaiso@uark.edu](mailto:shchaiso@uark.edu)

010878218

CSCE 5613 Artificial Intelligence

Spring 2025

Late Days: 0

## Class and Inheritance - 1

For the first task, the base class **Geometry** is given with the attributes: name, points, and count. The class also contains the following methods: calculate\_area(), get\_name(), and count\_number\_of\_geometry(cls) which returns the total number of created instances. The following sub-classes: **Triangle**, **Rectangle**, **Circle**, and **Polygon** inherits **Geometry** whereas the sub-class **Square** inherits **Rectangle**. This section defines a hierarchy of geometric shapes using Object-Oriented Programming (OOP).

The **Triangle** class takes three points (a, b, c) which corresponds to the vertices of a triangle. The calculate\_area() method uses the shoelace formula to compute the area of the triangle from the three points given by the following formula:

The **Rectangle** class takes two points (a, b) which correspond to the top-left and bottom-right corners of a rectangle. The calculate\_area() method computed the area as follows:

The **Square** class takes the top-left corner (a) and a side length (length). The bottom-right corner (b) is derived by adding the x-coordinate in a with length and subtracting the y-coordinate in a with length. The calculate\_area() method calculates the area of the square by using the following formula:

The **Circle** class takes a center point (o) and radius r. The calculate\_area() method calculates the area of the circle by using the following formula:

The **Polygon** class takes a list of points containing the vertices of a polygon. The calculate\_area() method calculates the area of the polygon by using the shoelace formula which is given by the following:

We can refer to the table below for the example inputs and outputs derived from calling each class:

|  |  |  |
| --- | --- | --- |
| Shape | Input | Output |
| Triangle | (0, 1), (1, 0), (0, 0) | 0.5000 |
| Rectangle | (0, 0), (2, 2) | 4.0000 |
| Square | (0, 0), 2 | 4.0000 |
| Circle | (0, 0), 3 | 28.2743 |
| Polygon | [(0, 0), (0, 1), (1, 1), (1, 0)] | 1.0000 |

**Figure 1. Table of Sample Test Cases for Geometry Classes**

## Matrix Multiplication - 2

For the next section, we implement matrix operations, recursive sequences, and Fibonacci number calculation using both iterative and recursive methods into code.

The **matrix\_multiplication** function takes two matrices A and B of size m x k and k x n to compute the resulting matrix C (size m x n). The matrix C is derived by using the dot product rule which follows:

The **test\_matrix\_mul** function tests the validity of our implementation by running 10 test cases with different random matrices sizes (m x n, n x k). Each test case is asserted to check for the function’s correctness each time.

|  |
| --- |
| [Test Case 0]. Your implementation is correct!  [Test Case 1]. Your implementation is correct!  [Test Case 2]. Your implementation is correct!  [Test Case 3]. Your implementation is correct!  [Test Case 4]. Your implementation is correct!  [Test Case 5]. Your implementation is correct!  [Test Case 6]. Your implementation is correct!  [Test Case 7]. Your implementation is correct!  [Test Case 8]. Your implementation is correct!  [Test Case 9]. Your implementation is correct! |
| **Figure 2.1 Sample Results of test\_matrix\_mul** |

The **recursive\_pow** function computes An recursively using exponentiation by squaring. If n = 0, then A = I (identity matrix) which is a base case. In a recursive case, An//2 is computed then squared. If n is odd, then the result is multiplied by A once more before returning.

The **iterative\_pow** function computed An iteratively using exponentiation by squaring. It starts by deriving the identity matrix (I), and repeatedly squares A and multiplies in a loop. At each step n is halved.

The **test\_pow** function runs test examples for each pow function 10 times. For each test, random square matrices (A) are generated and computes An.

|  |
| --- |
| Recursive: A^3 = [[-1.17729826 1.56280582 2.62707024]  [ 8.27898689 -8.01635022 -1.55958823]  [ 0.35603233 0.28781726 3.51788507]]  Recursive: A^3 = [[ 1.37504802 -1.35401839 2.8672233 ]  [-7.34075271 5.54406431 -7.77636587]  [ 5.40311869 -4.14446685 5.97321963]]  Recursive: A^5 = [[ 4.35881097 4.49533999 -2.22606652 -6.88203682 -4.01251593]  [ -6.84012372 43.41743045 -5.14045791 -27.12610589 -1.34282768]  [ 2.96918979 19.44214891 -3.82020814 -13.75055803 -1.78737707]  [ 5.57685091 -8.41406794 0.08160689 2.8973547 -1.11876453]  [ -1.48991607 9.65246706 -1.04864201 -2.33872971 2.59316961]]  Recursive: A^4 = [[-0.83451844 -2.17928416 1.61478835 10.66739896]  [-0.21842687 6.26804197 2.47097369 9.42408297]  [ 4.07616528 3.43708141 4.91338575 2.69414538]  [-3.45203113 -7.91596689 -4.15552713 -1.58693615]]  Recursive: A^2 = [[ 4.75719668 -2.72251146]  [ 1.06050737 -0.40721928]]  Recursive: A^3 = [[-1.53029063 -0.48015789 -3.33582203]  [-0.25919244 -0.39065227 -0.09026439]  [ 0.73686678 0.45174162 -0.31159217]]  Recursive: A^5 = [[-865.39405732 488.9624475 231.64921849 -36.56528359 -27.81198383]  [ 178.39151811 -103.0117843 -47.3765907 5.89915212 8.48100448]  [ 196.25737197 -110.98540701 -48.72197772 8.4841484 5.01810657]  [-367.32168396 206.98548742 94.23854058 -19.56039891 -21.27314875]  [ 125.86762327 -66.98211689 -41.57030349 9.49765489 7.27432969]]  Recursive: A^3 = [[ 2.30044376 0.75511523 0.80170998]  [-0.06546055 -4.5216886 -1.38353652]  [-1.55371969 -0.53934784 -0.51430949]]  Recursive: A^2 = [[ 0.05866506 0.06556798]  [-0.04069347 -0.04546521]]  Recursive: A^5 = [[-0.31691677 -1.52989625 0.15254164 1.26731331 3.68274098]  [ 3.21143784 1.95703532 -4.52520192 6.77525731 3.39013973]  [-7.95062453 5.10384368 -1.93275384 -0.30830723 0.77090194]  [10.23675218 -9.16770224 3.34535543 1.37763842 3.43080006]  [-2.30190209 -7.87067069 1.87341102 -5.28079681 -7.55996396]]  Iterative: A^5 = [[-10.52123245 -25.81877924 -48.16223853 -86.74679386 18.83391474]  [ 0.15173054 7.95436316 11.60399093 27.10675548 -3.83862509]  [ 10.95172648 -10.93087319 -5.26953148 -39.9712106 -3.29606279]  [ 8.47249644 -21.1805192 -22.54181563 -74.58967603 3.11586707]  [ -1.57437565 20.92649932 29.16962285 72.46479054 -7.85023325]]  Iterative: A^5 = [[ 7.53127292 -20.20431993 -35.48895643 19.45189711 -0.84850566]  [ 38.48400204 -54.06369842 39.11374701 -9.1654829 -12.80284628]  [ 41.70546325 -26.68572527 -18.39757565 38.89923683 28.44847357]  [ -4.56250475 2.50594051 -36.30953011 18.0120764 5.07813859]  [-30.97320315 3.24578058 -18.37948643 -16.4193416 -28.30139324]]  Iterative: A^4 = [[19.8067757 2.77707826 13.08630505 -6.77509747]  [15.0760674 1.41590662 11.38944297 -6.53037923]  [10.47971609 -2.26452966 7.11429201 12.95797717]  [10.98989731 0.69147072 6.84897946 1.46354258]]  Iterative: A^3 = [[-1.59426167 0.87064882 -0.54948234]  [-0.64748218 -2.10440027 -0.49503366]  [ 0.14034991 0.44362943 -1.9353784 ]]  Iterative: A^4 = [[ 0.70959981 2.56586673 0.8337171 1.17499527]  [-0.22962943 1.38563714 -3.21531955 -4.9471406 ]  [-0.52526832 0.69032228 -0.67409913 0.0715275 ]  [ 1.6547673 2.64546767 -0.35739997 -1.03796844]]  Iterative: A^2 = [[ 7.74109715 -2.8936672 ]  [-2.97084477 6.07101179]]  Iterative: A^4 = [[ -6.56878425 -8.6371544 -31.71651831 34.11595775]  [ 4.35299231 26.1127456 8.90794985 -4.57055385]  [ 5.81281012 15.21912921 18.26725155 -14.75613099]  [ -0.09036148 15.50237525 -13.26970661 20.14705403]]  Iterative: A^2 = [[ 5.26932433 5.31985842]  [-2.66974427 -2.24413044]]  Iterative: A^3 = [[0.26457263 0.96041492 1.87233309]  [0.27584739 2.42102367 3.45263417]  [0.15513428 1.26038764 1.53081969]]  Iterative: A^4 = [[ 0.05585153 1.79735553 0.48736244 -3.85750989]  [ 1.78308467 0.56839916 0.42911403 -2.2391367 ]  [ 2.20204198 -0.00846131 0.15761469 -1.94762575]  [-0.58748026 -1.5140865 -1.04799212 1.67792452]] |
| **Figure 2.2 Sample Results of test\_pow** |

The **get\_A** function defines the matrix representation of Fibonacci numbers as:

The **fibo** function can be used to compute the n-th Fibonacci number in log(n) time. The Fibonacci recurrence is represented in matrix multiplication as:

The function then returns the results as:

The **f()** function is defined as the recursive function given two numbers n and k. If n < k, then fn = 1 and 1 is returned. In the case where n > k, then the function calculates the n-th term of a sequence defined by a recurrence relation of order k.

|  |
| --- |
| [Test Case 0]. Your implementation is correct!. fibo(2) = 2  [Test Case 1]. Your implementation is correct!. fibo(3) = 3  [Test Case 2]. Your implementation is correct!. fibo(4) = 5  [Test Case 3]. Your implementation is correct!. fibo(5) = 8  [Test Case 4]. Your implementation is correct!. fibo(6) = 13  [Test Case 5]. Your implementation is correct!. fibo(7) = 21  [Test Case 6]. Your implementation is correct!. fibo(8) = 34  [Test Case 7]. Your implementation is correct!. fibo(9) = 55  f(5, 2) = 8  f(5, 3) = 9  f(5, 4) = 7  f(6, 2) = 13  f(6, 3) = 17  f(6, 4) = 13  f(7, 2) = 21  f(7, 3) = 31  f(7, 4) = 25  f(8, 2) = 34  f(8, 3) = 57  f(8, 4) = 49  f(9, 2) = 55  f(9, 3) = 105  f(9, 4) = 94  f(10, 2) = 89  f(10, 3) = 193  f(10, 4) = 181 |
| **Figure 2.3 Sample Results of fibo() and f() functions** |

## Breadth-First Search and Depth-First Search - 3

The **DFS** function performs the Depth-First Search (DFS) path-finding algorithm. The function will search deep into one path before backtracking. It first checks if A is an empty matrix or whether the start or end equals 0. If so, then -1 is returned. If not then the function proceeds to define moveable directions (up, left, right, down), path (stores the current path taken), and visited (a set to track visited cells).

The function then proceeds to a recursive process where it proceeds to find a path in all possible directions in each cell. If no path is found at any time, then the function backtracks to the previously visited cell and proceeds to a new cell that hasn’t been visited. At the end if a path is found it is printed or else it returns -1 therefore no valid path.

The **BFS** function performs the Breadth-First Search (BFS) path-finding algorithm. The function uses a queue to explore all possible paths by each level. Unlike the DFS algorithm, BFS finds the shortest path.

Both implementations of the algorithm are tested using the following A matrix:

The results of each implementation are as follows:

|  |
| --- |
| BFS Output:  (0, 0) → (0, 1) → (0, 2) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (3, 4) → (4, 4)  DFS Output:  (0, 0) → (0, 1) → (0, 2) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (3, 4) → (4, 4) |
| **Figure 3.1 Outputs of Path-Finding Algorithms** |

The **findMinimum** function finds the path with the minimum cost in a weighted matrix (A). A min-heap is defined which stores cost, x, y, path. The function then proceeds to explore all possible paths within the matrix while prioritizing the lowest cost. The following matrix (A) is defined as our sample input:

After running the matrix (A) through the function we achieve the following output:

|  |
| --- |
| Find Minimum Output:  (0, 0) → (0, 1) → (0, 2) → (1, 2) → (2, 2) → (2, 3) → (2, 4) → (3, 4) → (4, 4)  Total value: 10 |
| **Figure 3.2 Output of findMinimum function** |