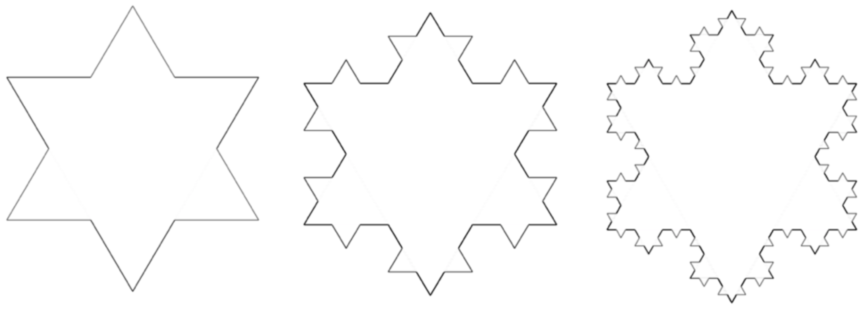
# Discrete Maths – Programming Assignment 2 Report

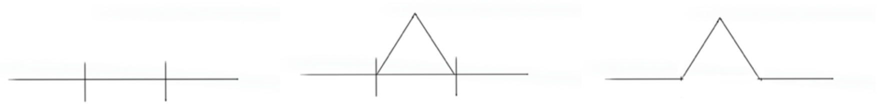
Team 5
21000284 박이삭
21500002 강동인
21500394 양수진
21700083 김도윤
21700646 전혜원

## Fractal 1 – Koch snowflake



This fractal is named as 'Koch snowflake', based on the Koch curve. This fractal can is created with following steps.

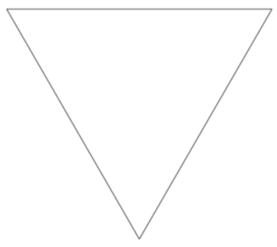
1. Draw an equilateral triangle
2. Divide each side into three and attach an equilateral triangle to the middle part of the division



3. Repeat step 2 infinitely - in our case, as many times as our input from the user

Implementation of the concept above in recursion
function `fractal(initial_point1, initial_point2, angle, depth)`

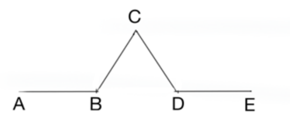
- Parameter `angle` is the degree that the point should turn to.
- Parameter `depth` in `fractal` represents *n* in the proposition below.
- Parameters `initial_point1` and `initial_point2` are starting coordinate and ending coordinate of a line (a side of a shape).



P(*n*) - Koch snowflake can be made with recursion of n times, where *n* is a positive integer.

- **When *n* < 0** : P(n) is always false. Since *n* needs to be a positive integer, `fractal` returns **null** when the input is a negative integer.
- **Basis step:** P(0) is true as Koch snowflake can be made when there is 0 recursion. An equilateral triangle will be formed.
- **Recursive step** :

Within `fractal` , firstly, a line is drawn from `initial_point1` to `initial_point2`. Parameters `initial_point1` and `initial_point2` are utilized to calculate points of internal division and middle point to use those points in the recursion. Then, four `fractal` functions called within the `fractal` function.



| | |
|---|---|
| `fractal(initial_point1, point1, angle, depth-1);` | A -> B |
| `fractal(point1, star_middle, angle-60, depth-1);` | B -> C |
| `fractal(star_middle, point2, angle+60, depth-1);` | C -> D |
| `fractal(point2, initial_point2, angle, depth-1);` | D -> E |

The parameter `depth` of `fractal`s in a recursion is `depth-1` , so the recursion occurs until `depth` becomes 0, performs basis step, and then ends the program.
The calculation for points used in `fractal` are given below.

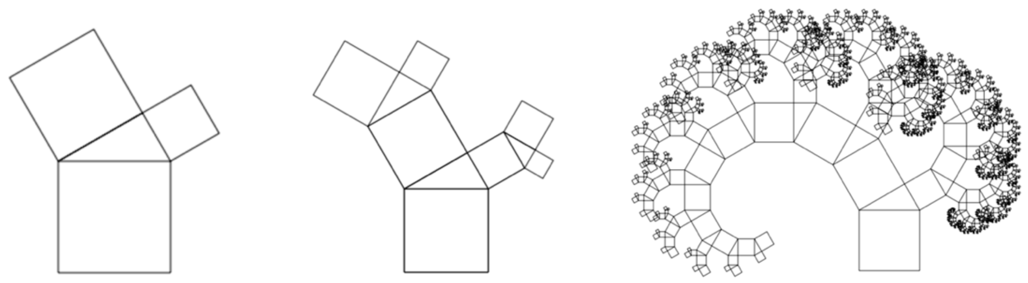
$$point1 = \frac{(internalpoint1 * 2) + internalpoint2}{3} \qquad point2 = \frac{(internalpoint2 * 2) + internalpoint1}{3}$$

$$starmiddlex = point[0] + \frac{cos((angle - 60) * degtorad) * length(initialpoint1, initialpoint2)}{3}$$

$$starmiddley = point[1] + \frac{sin((angle - 60) * degtorad) * length(initialpoint1, initialpoint2)}{3}$$

$$starmiddle = [starmiddlex, starmiddley]$$

## Fractal 2 – Pythagoras Triangle



This fractal is named as 'Pythagoras Triangle', based on Pythagoras theorem.
Pythagoras theorem : The square of the hypotenuse of a right-angle triangle is equal to the sum of the squares of the other two sides.
This fractal can is created with following steps.

1. Draw a square, and then draw a right-angle triangle with the side of a square as a hypotenuse. Then, draw two squares, each with each side of a right-angle triangle as a side of the square. These two squares will be referred as *square a* and *b*.
2. Draw a right-angle triangle with the side of *square a* as a hypotenuse. Draw another right-angle triangle with the side of *square b* as a hypotenuse in the same method.
3. Repeat step 2 infinitely - in our case, as many times as our input from the user

<u>Implementation of the concept above in recursion</u>
function `draw_Pythagorean_tree(x1, y1, x2, y2, angle, depth)`

- `(x1, y1)` and `(x2, y2)` are coordinates of points used in calculation.
- `angle` is the value of angle needed to rotate in order to calculate coordinates.
- `depth` is the degree of recursion.

P(*n*) - Pythagorean Tree can be drawn with recursion of n times, where *n* is a positive integer.
Since *n* needs to be a positive integer, `draw_Pythagorean_tree` does not draw anything when the input is a negative integer.

- **Basis step**: P(0) is true. Since there is 0 recursion, a square is drawn - step 1 in the concept above

this part is drawn with a function `draw_BaseCase(x1, y1, x2, y2)`

- **Recursive step:** `draw_Pythagorean_tree` is a recursive function.

if `depth` is greater than 0, the recursion continues.
First, calculate points needed in drawing two squares that form right-angle triangle in between.
Below are formulas used in the calculation.

$triaglepointx = x1 + (\cos((angle+60)*degtorad)*(length(x1,y1,x2,y2)*(\frac{\sqrt{3}}{2})))$ $trianglepointy = y1 + (\sin((angle+60)*degtorad)*(length(x1,y1,x2,y2)*(\frac{\sqrt{3}}{2})))$

$$leftpointx = x1 + (\cos((angle-30)*degtorad)*(length(x1,y1,trianglepointx,trianglepointy))$$

$$leftpointy = y1 + (\sin((angle-30)*degtorad)*(length(x1,y1,trianglepointx,trianglepointy))$$

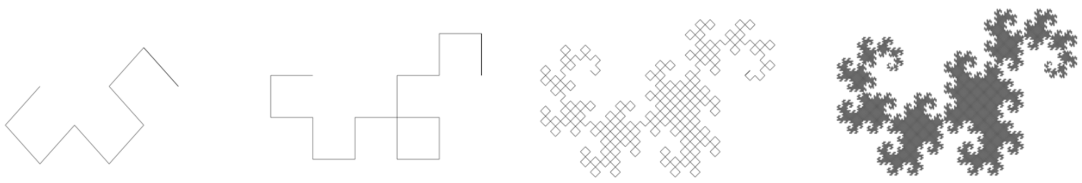
$lp2x = tpx + (\cos((angle-30)*degtorad)*(length(x1,y1,tpx,tpy))$ $lp2y = tpy + (\sin((angle-30)*degtorad)*(length(x1,y1,tpx,tpy))$

$rp1x = tpx + (\cos((angle+60)*degtorad)*(length(tpx,tpy,x2,y2))$ $rp1y = tpy + (\sin((angle+60)*degtorad)*(length(tpx,tpy,x2,y2))$

$rp2x = x2 + (\cos((angle+60)*degtorad)*(length(tpx,tpy,x2,y2))$ $rp2y = y2 + (\sin((angle+60)*degtorad)*(length(tpx,tpy,x2,y2))$

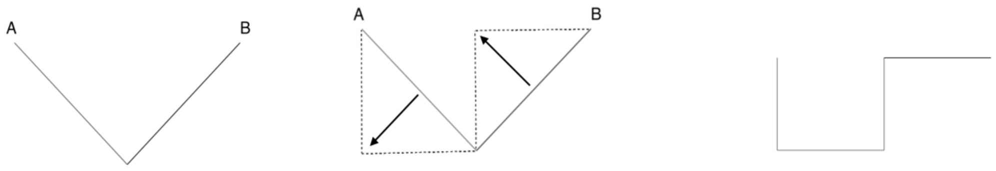
Then use multiple `drawLine` functions to draw two squares that form right-angle triangle between.
Then two `draw_Pythagorean_tree` functions are used as recursion, each used for each side of a square with depth decreased by 1.

## Fractal 3 – Dragon curve



This fractal is named as 'Dragon curve'. This curve is drawn using relative movement. The shape resembles a dragon as the depth increases, thus called 'Dragon curve'. This fractal can is created with following steps.

1. Draw two line. these two lines form a right angle.
2. Find a middle point of each segment by dividing each segment into two. The first segment moves outwards and the second segment moves inwards.



3. Repeat step 1 and 2 infinitely - in our case, as many times as our input from the user

<u>Implementation of the concept above in recursion</u>
function - `drawDragon(start_point_1, start_point_2, mode, depth)`

- `start_point_1` and `start_point_2` are coordinates given to draw two lines forming a right angle.
- `mode` indicates which way a segment should turn.
- `depth` is use d to inform degree of recursion.

P(*n*) - Dragon curve can be made with recursion of n times, where *n* is a positive integer.
Since *n* needs to be a positive integer, `drawDragon` returns **null** when the input is not a positive integer.

- **Basis step** (*n* = 0): P(1) is true as two lines that form right angle is drawn.
- **Recursive step** (*n* > 1)**:**

`drawDragon` calculates a point depending on the `mode`. `mode` is a parameter input to decide which way a segment should turn towards.

if mode is 1, point for recursion is calculated to turn right, using formula below.

$$pointx = startpoint1[0] + \frac{\cos((angle+45)*degtorad)*length(startpoint1,startpoint2)}{\sqrt{2}}$$

$$pointy = startpoint1[1] + \frac{\sin((angle+45)*degtorad)*length(startpoint1,startpoint2)}{\sqrt{2}}$$

if mode is 2 - point for recursion is calculated to turn left, using formula below.

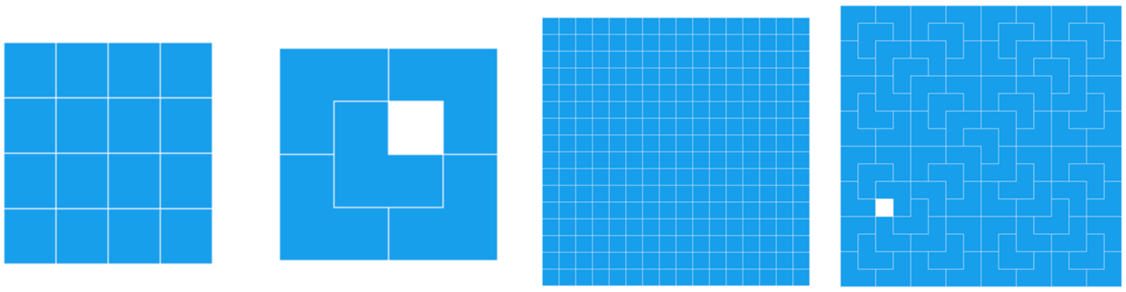
$$pointx = startpoint1[0] + \frac{\cos((angle - 45) * degtorad) * length(startpoint1, startpoint2)}{\sqrt{2}}$$

$$pointy = startpoint1[1] + \frac{\sin((angle - 45) * degtorad) * length(startpoint1, startpoint2)}{\sqrt{2}}$$

Then `drawDragon` function is called twice within `drawDragon` function. The first function is called to turn the first segment outwards. The second function is called to turn the second segment inwards.

```
drawDragon(start_point1, point, 1, depth-1);
drawDragon(point, start_point_2, 2, depth-1);
```

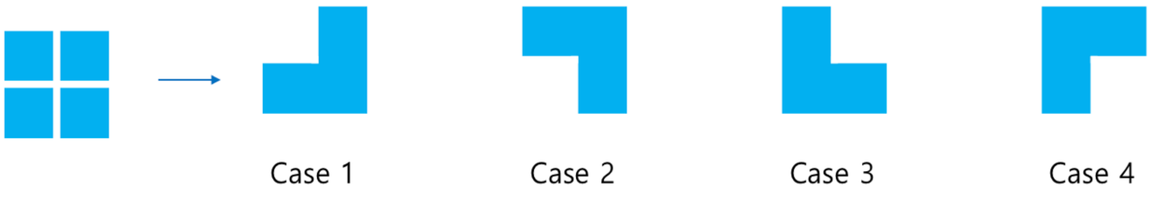
## Fractal 4 - Solving Triomino - Tiling Problem



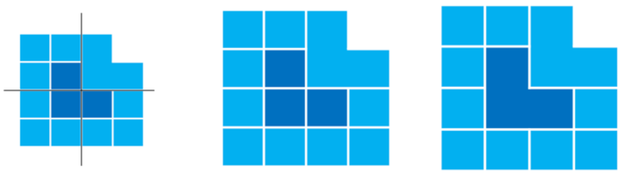
Triomino is an L-shaped polygon formed with three squares connected together. The purpose of this Tiling Problem is to fill in the $2^n$ X $2^n$ checkerboard with just triominos.

The solution to this tiling problem can be created by following steps.

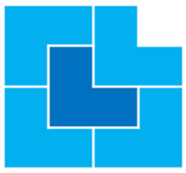
1. Connect three squares, excluding an empty square - must be L-shaped (4 cases exist)



2. Make three squares attached to an just-made square into a triomino.



3. Repeat step 1 and 2 infinitely - in our case, as many times as our input from the user



Implementation of the concept above in recursion

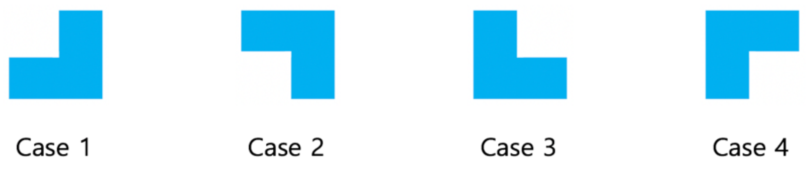
This program is programmed to enter number *n* to create $2^n$ X $2^n$ checkerboard.

function `findBlock(start_x, start_y, zero_x, zero_y, num)`

- `start_x` and `start_y` are starting coordinates of the calculation.
- `zero_x` and `zero_y` are are the coordinate of a block that has been clicked - empty square.
- `num` is the degree of recursion, which is derived from the number of blocks.

- **Basis case (num = 2) :** There are four base cases as illustrated below.

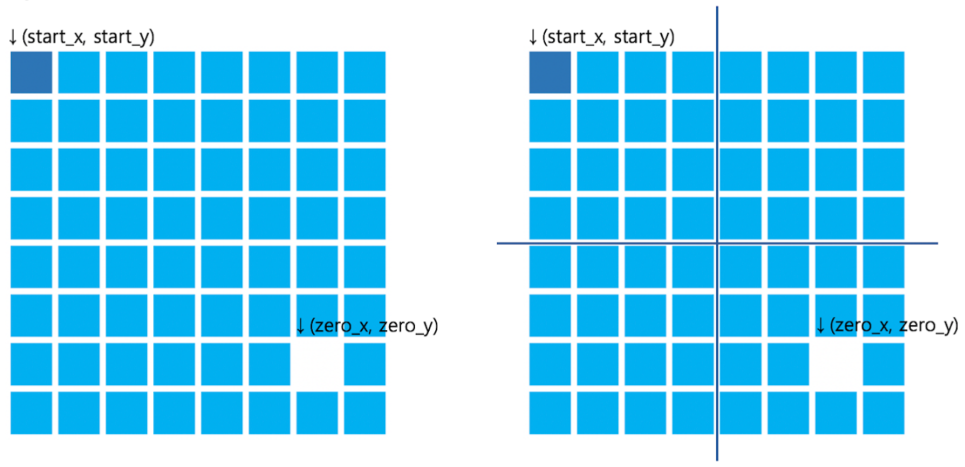
<base case>



- **Recursive case (num > 2):**

Four `findBlock` functions are called within recursive step of `findBlock` function.
The whole recursion is described below.
First, calculate middle point coordinate (x, y) and save the value in `middle_x` and `middle_y`.
`middle_x` and `middle_y` are used to divide the checkerboard into four quarters.

<general case>

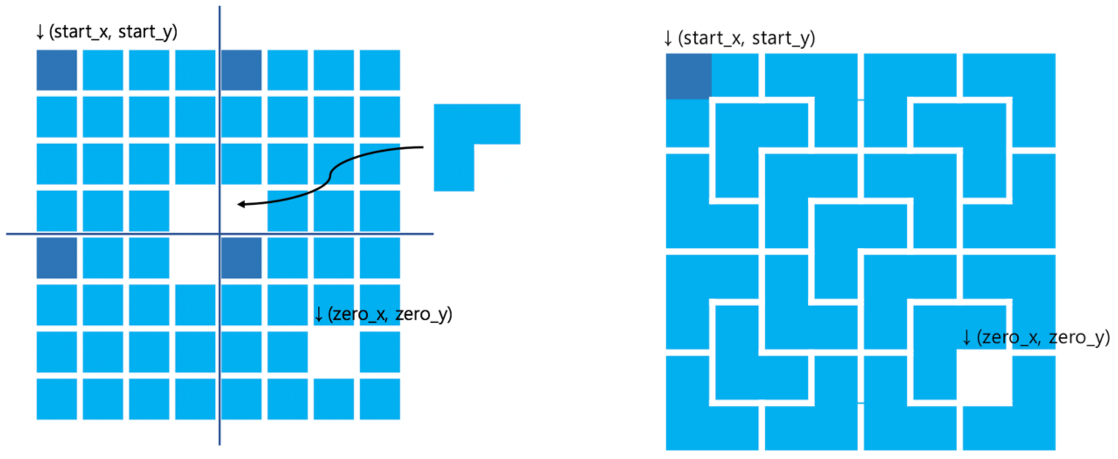


The cases are divided into four, depending on which quarter of the checkerboard the empty square is located at.

case 1: empty square is in the first quarter
case 2: empty square is in the second quarter
case 3: empty square is in the third quarter
case 4: empty square is in the fourth quarter

In each case -

① call `findBlock` for the quarter containing the empty square to fill this quarter with triominos.
② Fill in the arbitrary triomino-shaped empty squares in the middle of the checkerboard with a triomino.
③ call three `findBlock` functions for cases not used in the ①, so that the whole checkerboard can be filled with triominos.


↓(start_x, start_y)
↓(zero_x, zero_y)
↓(start_x, start_y)
↓(zero_x, zero_y)