National University of Computer and Emerging Sciences

Fast School of Computing Fall 2024

CS1002-Programming Fundamentals (CS-A,B,C,D,E,F,G)
Assignment 04

Instructions for submission:

Dear students we will be using auto-grading tools, so failure to submit according to the below format would result in zero marks in the relevant evaluation instrument.

i. For each question in your assignment, make a separate cpp file e.g. for question 1, make ROLL-NUM_SECTION_Q#.cpp (24i-0001_A_Q1.cpp) and so on. Each file that you submit must contain your name, student-id, and assignment # on top of the file in comments.
ii. Combine all your work in one folder. The folder must contain only .cpp files (no binaries, no exe files etc.).
iii. Run and test your program on a lab machine before submission.
iv. Rename the folder as ROLL-NUM_SECTION (e.g. 24i-0001_A) and compress the folder as a zip file. (e.g. 24i-0001_A.zip). do not submit .rar file.
v. Submit the .zip file on Google Classroom within the deadline.
vi. Submission other than Google classroom (e.g. email etc.) will not be accepted.
vii. The student is solely responsible to check the final zip files for issues like corrupt file, virus in the file, mistakenly exe sent. If we cannot download the file from Google classroom due to any reason it will lead to zero marks in the assignment.
viii. Displayed output should be well mannered and well presented. Use appropriate comment and indentation in your source code.
ix. Total Marks: 150.
x. If there is a syntax error in code, zero marks will be awarded in that part of assignment.
xi. Your code must be generic.
xii. Solve the assignment using the concepts of nested loops and iterative structures, as well as the concepts we have studied previously.
xiii. You cannot use advanced constructs like pointers for this assignment
xiv. Try to submit your assignment 3 hours before the deadline to avoid any problem (e.g. internet issues etc)

Deadline:

Deadline to submit assignment is 17th November, 2024 11:59 PM. You are supposed to submit your assignment on GOOGLE CLASSROOM (CLASSROOM TAB not lab). Only ".ZIP" files are acceptable. Other formats should be directly given ZERO. Correct and timely submission of the assignment is the responsibility of every student, hence no relaxation will be given to anyone. **Late Submission policy will be applied as described in course outline**.

Tip: For timely completion of the assignment, start as early as possible.

Plagiarism: Plagiarism is not allowed. If found plagiarized, you will be awarded zero marks in the assignment (copying from the internet is the easiest way to get caught).

Note: Follow the given instruction to the letter, failing to do so will result in a zero.

General Instructions for the assignment:

1. **Variable:** Use variables that reflect the context of the problem. Avoid generic names like `x`, `y`, or `z`.

2. **Logical Thinking:** In your code comments, explain why you chose specific variable names and why a particular operation (like `+` or `%`) is necessary for the problem's solution. These comments will be checked for correctness.

3. **Comments and Documentation:** Add a comment at the top of your code that includes your name, roll number, and a brief description of the program. Each function should have a comment explaining its purpose and parameters. Use comments to explain any non-obvious parts of your code.

4. **Input/Output Handling:** Provide clear instructions when taking input from the user (except for question 2). Format your output clearly, ensuring it's easy to understand and follows the requirements of the scenario in the assignment.

Evaluation Criteria

1. Your assignment will be evaluated based on:
2. Correctness: Does the program produce the correct results for all inputs?
3. Complexity: Are multiple conditions and adjustments applied using appropriate decision structures (e.g. switch, nested if-else statements and ternary operators etc)?
4. Efficiency: Is the code clean, efficient, and well-commented?
5. Comprehensive Output: Does the program handle all scenarios with clear and concise output?
6. Error Handling: Ensure the program manages invalid inputs gracefully

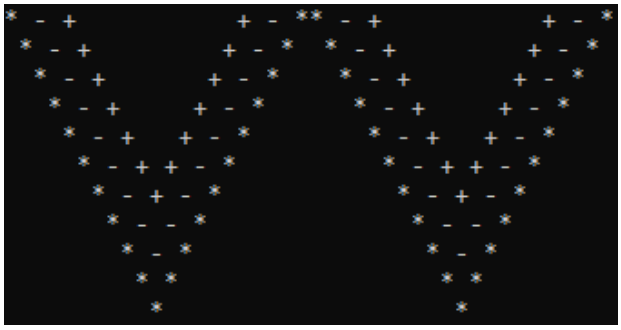
Q1: Printing Simple Pattern [50 Marks]

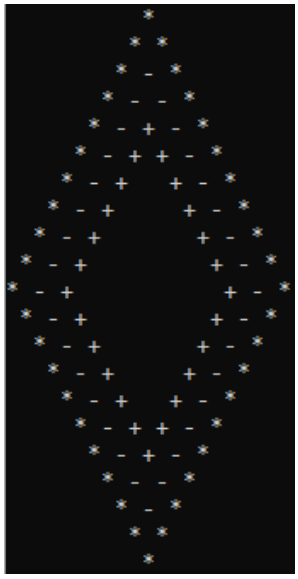
You are required to recreate these exact patterns using nested loops on the terminal.

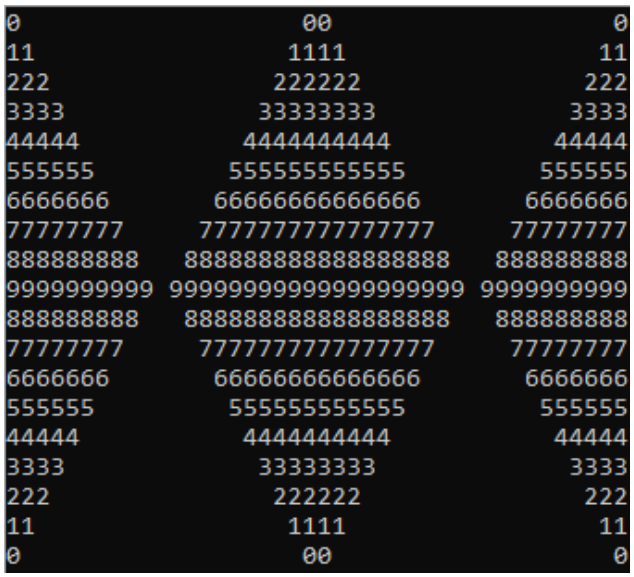
Important Notes:

1. **Hard-coding these patterns will result in zero marks.**
2. The use of setw() and setfill() functions is strictly prohibited and will result in zero marks.

1) A) [10 Marks] B) [10 Marks] C) [10 Marks]







---

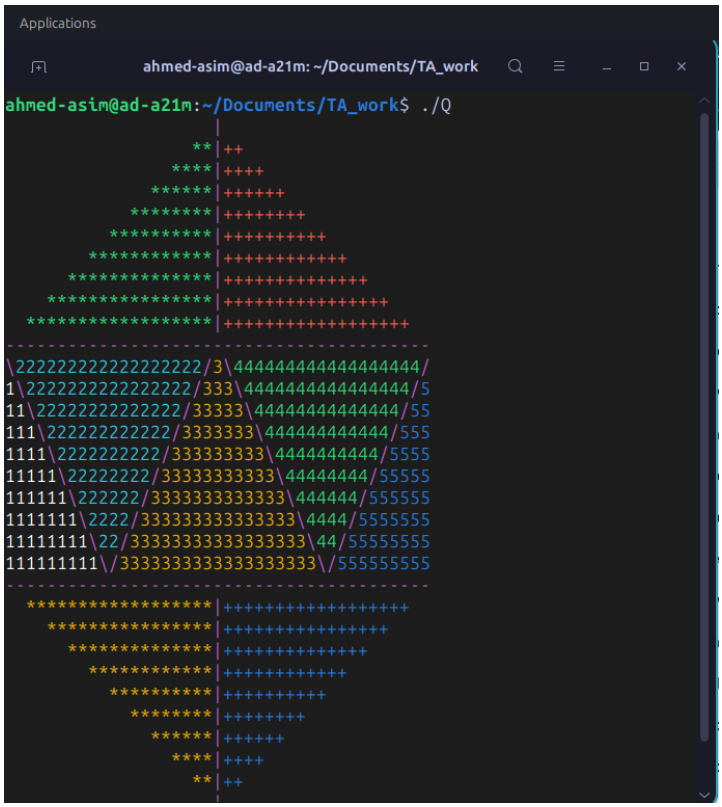
2) [20 Marks] The pattern includes specific colors. **To achieve this, use the command:** cout << "\033[92m" << "Hello World" << "\033[0m";

Here:

- '\033[92m' applies a shade of green to the spaces.
- '\033[0m' resets the color to default after each colored segment.

Note:

You may refer to this guide on GeeksforGeeks for help on changing console colors.



Q2: Matrix Key Finder [20 marks]

You are given an 8x8 matrix initialized with zeros. The actual matrix you'll work with is an n x n section of this matrix, where $3 \leq n \leq 8$. *Starting from index 0,0, use the n x n part of the 8x8 matrix.* You'll receive an integer n representing the matrix size, followed by n x n binary values (0s and 1s) as input.

Input:

- The **first line** of input is an integer, `n` ($3 \leq n \leq 8$).
- The **next `n` lines** each contain `n` binary values, separated by spaces.

Inside code:

```
int main() {

    int size;
    int matrix[8][8] =
    { { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0},
      { 0, 0, 0, 0, 0, 0, 0, 0} };

    cin >> size;
```

On Terminal Display:

```
3
1 0 1
1 0 1
1 1 1
```

```
8
1 0 1 1 0 1 0 1
1 1 1 0 0 0 1 0
1 0 1 0 1 0 1 1
1 0 0 0 0 0 0 1
0 1 1 1 1 1 1 1
1 0 0 1 1 0 0 1
1 1 0 0 1 1 0 0
1 0 0 0 0 0 0 0
```

```
5
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
1 0 1 0 1
```

Traversal Rules: (It will be further explained in the example)

1. Start at the top-left corner matrix [0][0].
2. The first diagonal movement is determined by the value at matrix[0][0]:
    - If matrix[0][0] = 1, the first movement is upwards (i.e., from bottom-left to top-right).
    - If matrix[0][0] = 0, the first movement is downwards (i.e., from top-right to bottom-left).
3. The traversal proceeds diagonally, switching directions after each diagonal:
    - Traverse all elements on the first diagonal, starting from (0,0).
    - The second diagonal starts from (0,1) and goes towards (1,0) [if matrix[0][0] = 0, (y, x) where 'y' is row and 'x' is column].
    - Continue traversing until all elements have been visited.

Key Generation:

- Collect the binary values encountered during the diagonal traversal in the order they are visited.
- Form a binary number from these values.
- Use **bitwise operations** to convert the binary number to its **decimal (Base 10) representation**.
- Calculate the key as log base 2 of the decimal value, and **print the next highest integer** (e.g., 3.3 would be 4).

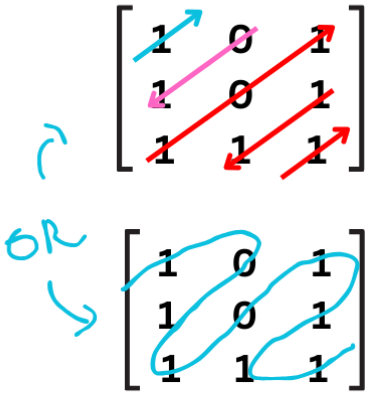
Task:

Write a program that:

1. Traverses the given matrix diagonally according to the rules described above (Your code should be generic and work for all the values of n mentioned above.).

2. Collects the binary values, forms the corresponding binary number, and calculates the key using bitwise operations.
3. Outputs the computed key (rounded to the nearest integer) along with binary number and decimal number.

Example:



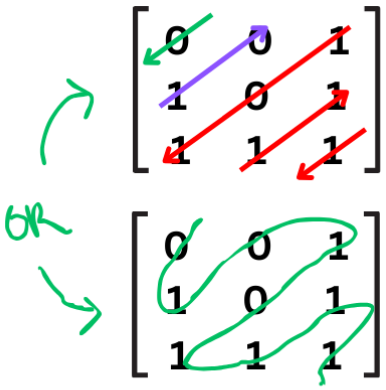
As matrix[0][0] = 1, the first diagonal will face north-east, the second diagonal will face south-west starting from (0,1), so on and so forth

Ans = 101101111(b) = 367(d)

log2(367) = 8.52 ~ 9

```
3
1 0 1
1 0 1
1 1 1
367
101101111
9
```

.....



As matrix[0][0] = 0, the first diagonal will face south-west, the second diagonal will face north-east starting from (1,0), so on and so forth

Ans = 010101111(b) = 175(d)

log2(175) = 7.451 ~ 8

```
3
0 0 1
1 0 1
1 1 1
175
10101111
8
```

Output:

- **First Line**: Print the **decimal number** obtained from the binary sequence.
- **Second Line**: Print the **full binary number**.
- **Third Line**: Print the **computed key**.

```
8
1 0 1 1 0 1 0 1
1 1 1 0 0 0 1 0
1 0 1 0 1 0 1 1
1 0 0 0 0 0 0 1
0 1 1 1 1 1 1 1
1 0 0 1 1 0 0 1
1 1 0 0 1 1 0 0
1 0 0 0 0 0 0 0
13783580349807603336
10111111010010010001110101001100101100110101011000101110100010000
64
```

Constraints:

- Use **nested loops** and **bitwise operators** to solve the problem (log2 functionality is to be done with bitwise operators).

- No libraries other than *iostream* are allowed.

Bonus Opportunity:

Upload your code to the Coding Showding contest on Hackerrank. If all test cases pass, you'll earn a bonus!

Q3: Maze Traversal Challenge [20 Marks]

You are given a maze represented by an N×M binary matrix. In this matrix:

- **0** represents a wall (a cell that cannot be traversed).
- **1** represents a valid path (a cell that can be traversed).

A rat is initially located at the starting cell of the maze, (0,0), and it must find a path to reach a specified target cell, (x, y), where cheese is located. Your task is to write a C++ program that determines if a path exists for the rat to reach the cheese cell from its starting point. If a path is found, you will display the path in the maze.

Requirements

1. **Input**
   - You are given matrix sizes N, M.
   - An N×M matrix where each cell contains either **0** (wall) or **1** (path).
   - The destination coordinates, x and y, representing the cell where the rat needs to reach.
   - These inputs will be manually entered in code as shown below

```
const int n = 3, m = 3;
int reachRow = 3, reachCol = 3;
int maze[n][m] = {{1, 1, 0},
                  {0, 1, 0},
                  {1, 1, 1}};
```

```
const int n = 6, m = 6;
int reachRow = 1, reachCol = 6;
int maze[n][m] = { {1, 1, 0, 1, 1, 1},
                   {0, 1, 0, 1, 0, 1},
                   {0, 1, 0, 1, 1, 1},
                   {1, 1, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1, 1},
                   {1, 1, 1, 1, 0, 1} };
```

```
const int n = 8, m = 8;
int reachRow = 7, reachCol = 8;
int maze[n][m] = { {1, 1, 1, 1, 0, 0, 0, 0},
                   {0, 1, 0, 0, 0, 0, 0, 0},
                   {1, 1, 0, 1, 0, 1, 0, 1},
                   {0, 1, 1, 1, 1, 1, 1, 1},
                   {1, 0, 0, 0, 0, 1, 0, 0},
                   {1, 0, 0, 0, 0, 1, 0, 0},
                   {0, 1, 1, 1, 1, 1, 1, 1},
                   {0, 1, 0, 0, 0, 0, 0, 0} };
```

2. **Path Display**
   - If a path is found:
     - Display the maze with the path highlighted using '**–**' for horizontal moves and '|' for vertical moves.
     - If no path is found, output "Path not reachable."
3. **Traversal Mechanism**
   - Traverse the matrix by following cells with a value of **1**.
   - You are required to store the coordinates of the mouse (x, y), as well as the direction of mouse before each movement into a 2d array of constant size.
   - **For example:** Use a constant-size array, data[3][M * N], to store the rat's movement history:
     - data[0][n] for row coordinates,
     - data[1][n] for column coordinates, and
     - data[2][n] for movement direction.
   - If the path leads to a dead end, you should go **back** to the last cell where movement was possible (this will be your most recent value which has been added to the array of data), mark the dead-end cell as **0** (so it won't be revisited), and continue searching for the target cell from the previous position.

Examples

```
const int n = 8, m = 8;
int reachRow = 7, reachCol = 8;
int maze[n][m] = { {1, 1, 1, 1, 0, 1, 1, 1},
                   {0, 1, 0, 0, 0, 1, 0, 0},
                   {1, 1, 0, 1, 0, 1, 0, 1},
                   {0, 1, 1, 1, 1, 1, 1, 1},
                   {1, 0, 0, 1, 0, 1, 0, 0},
                   {1, 0, 1, 0, 0, 1, 0, 0},
                   {1, 1, 1, 1, 1, 1, 1, 1},
                   {0, 1, 0, 0, 0, 1, 0, 0} };
```

```
Path is avalible!
[-|XXXXXX]
[X|XXXXXX]
[X|XXXXXX]
[X----|XX]
[XXXXX|XX]
[XXXXX|XX]
[XXXXX--*]
[XXXXXXXX]
```

```
const int n = 8, m = 3;
int reachRow = 1, reachCol = 3;
int maze[n][m] = { {1, 0, 1},
                   {1, 0, 1},
                   {1, 0, 1},
                   {1, 0, 1},
                   {1, 0, 1},
                   {1, 0, 1},
                   {1, 0, 1},
                   {1, 1, 1} };
```

```
Path is avalible!
[|X*]
[|X|]
[|X|]
[|X|]
[|X|]
[|X|]
[|X|]
[--|]
```

```
const int n = 6, m = 6;
int reachRow = 1, reachCol = 6;
int maze[n][m] = { {1, 1, 0, 1, 1, 1},
                   {0, 1, 0, 1, 0, 1},
                   {0, 1, 0, 1, 1, 1},
                   {1, 1, 1, 0, 0, 1},
                   {1, 0, 0, 1, 1, 1},
                   {1, 1, 1, 1, 0, 1} };
```

```
Path is avalible!
[-|XXX*]
[X|XXX|]
[X|XXX|]
[|-XXX|]
[|XX--|]
[---|XX]
```

```
const int n = 6, m = 8;
int reachRow = 1, reachCol = 8;
int maze[n][m] = { {1, 1, 1, 1, 0, 1, 1, 1},
                   {0, 1, 0, 0, 0, 1, 0, 0},
                   {1, 1, 0, 1, 0, 1, 0, 1},
                   {0, 1, 1, 1, 1, 1, 1, 1},
                   {1, 0, 0, 1, 0, 1, 0, 0},
                   {0, 1, 0, 0, 0, 1, 0, 0} };
```

```
Path is avalible!
[-|XXX--*]
[X|XXX|XX]
[X|XXX|XX]
[X----|XX]
[XXXXXXXX]
[XXXXXXXX]
```

```
const int n = 3, m = 3;
int reachRow = 3, reachCol = 3;
int maze[n][m] = {{1, 1, 0},
                  {0, 1, 0},
                  {1, 1, 1}};
```

```
Path is avalible!
[-|X]
[X|X]
[X-*]
```

```
const int n = 6, m = 8;
int reachRow = 1, reachCol = 8;
int maze[n][m] = { {1, 1, 1, 1, 0, 1, 1, 1},
                   {0, 1, 0, 0, 0, 1, 0, 0},
                   {1, 1, 0, 1, 0, 1, 0, 1},
                   {0, 1, 1, 1, 1, 1, 1, 1},
                   {1, 0, 0, 1, 0, 1, 0, 0},
                   {0, 1, 0, 0, 0, 1, 0, 0} };
```

```
Path is not avalible
```

Q4: Simulate Game of Life [20 Marks]

Conway's *Game of Life* is a captivating simulation that shows how complex, lifelike behavior can emerge from a simple set of rules. Created in 1970 by mathematician John Conway, this "game" reveals surprising patterns that evolve over time—cells organize, oscillate, replicate, and even move across the grid in ways that feel almost alive. Through this assignment, you'll experience the beauty of these interactions firsthand, as a 30x30 grid of cells transforms into dynamic patterns, uncovering hidden structures and unexpected behaviors in each generation.

Your task is to simulate Conway's *Game of Life*, which operates on a grid of cells that can be either alive or dead. The grid evolves over generations based on specific rules. For this assignment, follow the instructions and requirements below to complete the simulation.

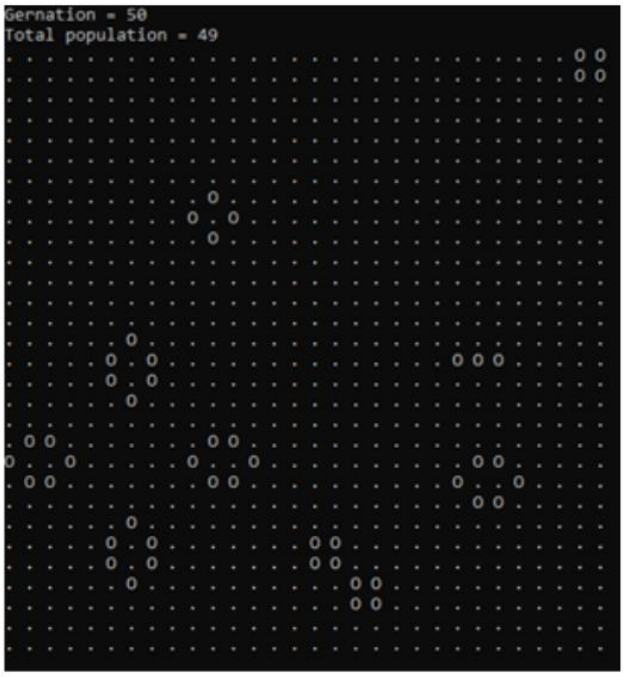
Instructions and Requirements:

1. **Grid Initialization**:
    - Define a 2D array of constant size **30x30** to represent the grid for the Game of Life.
    - **Load data** for this grid from a text file. The file will contain a 30x30 matrix of 1s and 0s, where 1 represents a live cell and 0 represents a dead cell.
    - **File Input**: Prompt the user to enter the file name, storing the file name in a character array (not a string object). File name will be no longer than 20 characters including ".txt".
2. **Game Rules**: Implement the following rules to determine the state of each cell in the next generation:
    - A live cell with fewer than two live neighbors dies (underpopulation).
    - A live cell with two or three live neighbors survives.
    - A live cell with more than three live neighbors dies (overpopulation).
    - A dead cell with exactly three live neighbors becomes alive (restoration).
3. **Simulation Steps**:
    - Prompt the user to input the number of generations to simulate.
    - For each generation:
        - Display the current grid with the **generation number** and the **population count** (total number of live cells).
        - Apply the Game of Life rules to determine the state of the next generation.
        - **Use system("clear")** to clear the console after each generation.
        - Use usleep() from <unistd.h> to add a small delay to simulate animation.
4. **File Output**:
    - After each generation, **save the updated grid to the same file** from which it was loaded, overwriting previous data.

Additional Information:

- A **simulation video has also been shared** to demonstrate the expected output and behavior of the simulation.

Before



After

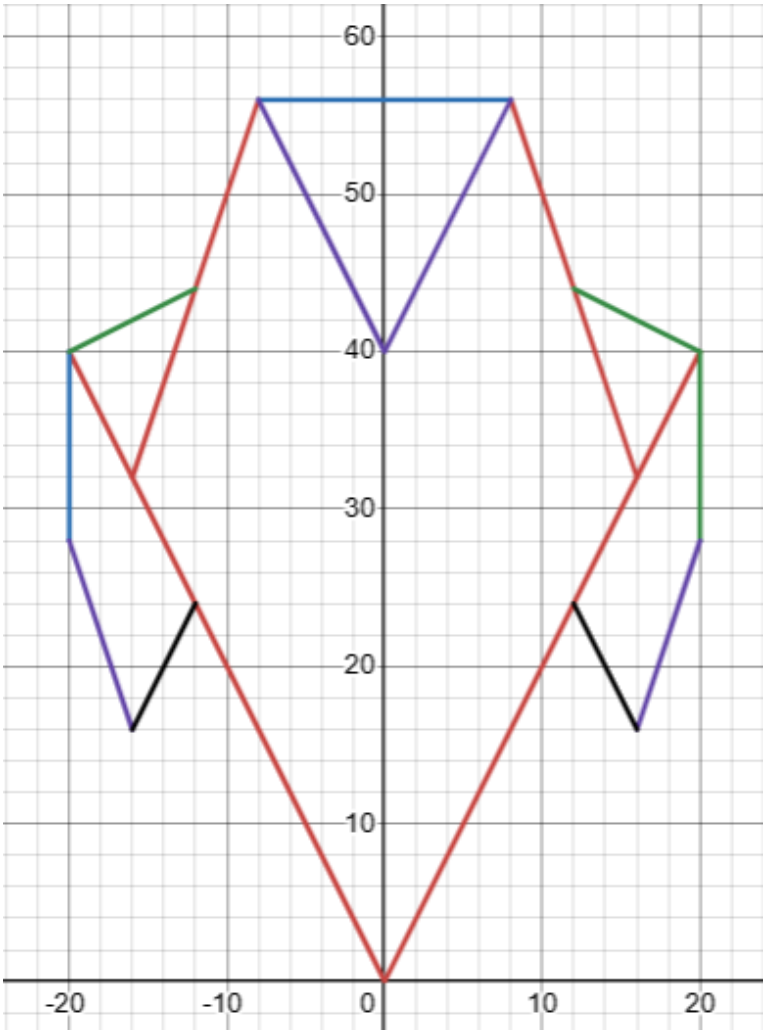
Q5: Art Pattern with Nested Loops [20 Marks]

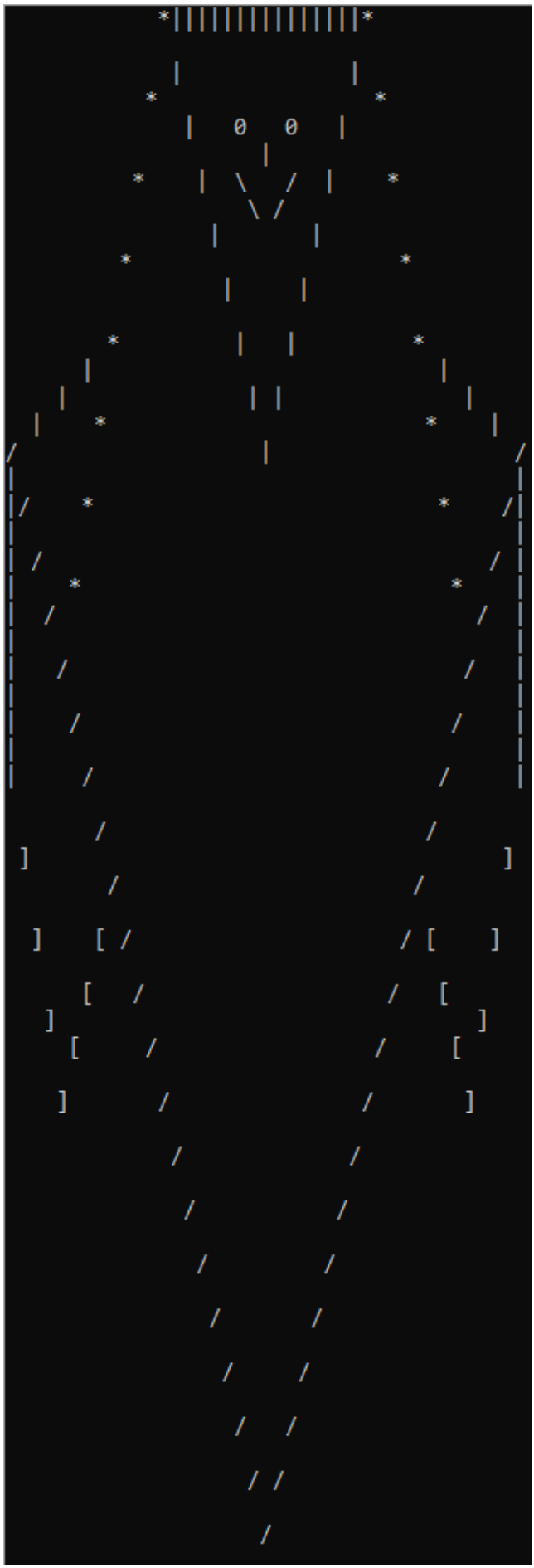
You are required to recreate this exact pattern using nested loops on the terminal. The first image (on right) shows the expected terminal output, and the second image (below) provides a grid view for reference.

Hint: Use a divide-and-conquer approach by creating a function for each line segment and plotting it inside a 2D nested loops using for loops, representing the x-axis and y-axis. (Hint is further explained in next page)

Note: While students are encouraged to use the divide-and-conquer approach with functions, they are free to apply any method as long as the shape is created using loops. Hard-coding the pattern will result in zero marks. Additionally, students may only use the characters '|', '/', '\', '*', '0', '[', and ']' to construct the pattern. The use of setw() and setfill() functions is strictly prohibited and will result in zero marks.

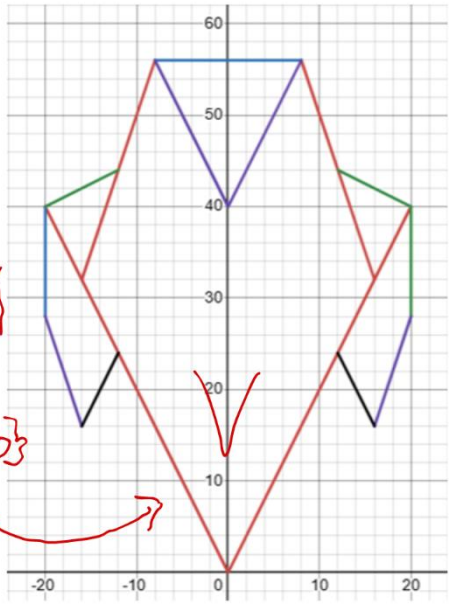
Don't forget to draw the face ☺





Further explanation of hint:

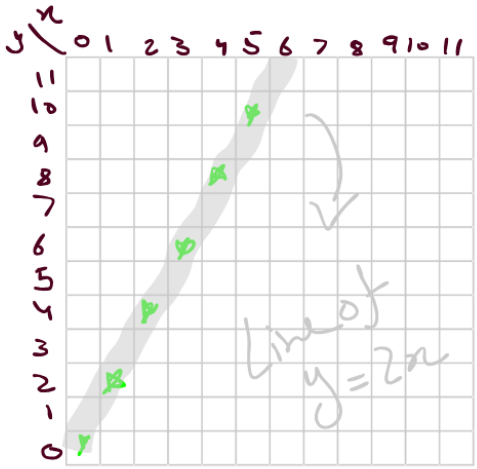
- Imagine a grid underlying the shape shown above (as seen on the terminal).
- Create functions for each line segment of the shape, specifying its domain and range. For each "grid box" or coordinate within the 2D array, determine whether to place a character by checking if it falls on a plotted line.
- You can use Desmos Graphing Calculator to help visualize and plot each line of the shape. Use the format y = |-2x| { -20 < x < 20} to plot individual lines and understand their domain and range.



f(x) = |-2x|
where
{-20 ≤ x ≤ 20}

For Example :-

```
— y loop —
  — x loop —
    if(y==2*x)
      cout << '*';
    else
      cout << ' ';
```



Q6: Shape Drawing with Nested Loops [20 marks]

You are required to write a general code that can draw a diamond-shaped pattern using nested loops. The code should be flexible and able to generate the pattern based on user input. Below are the requirements and examples for clarification.

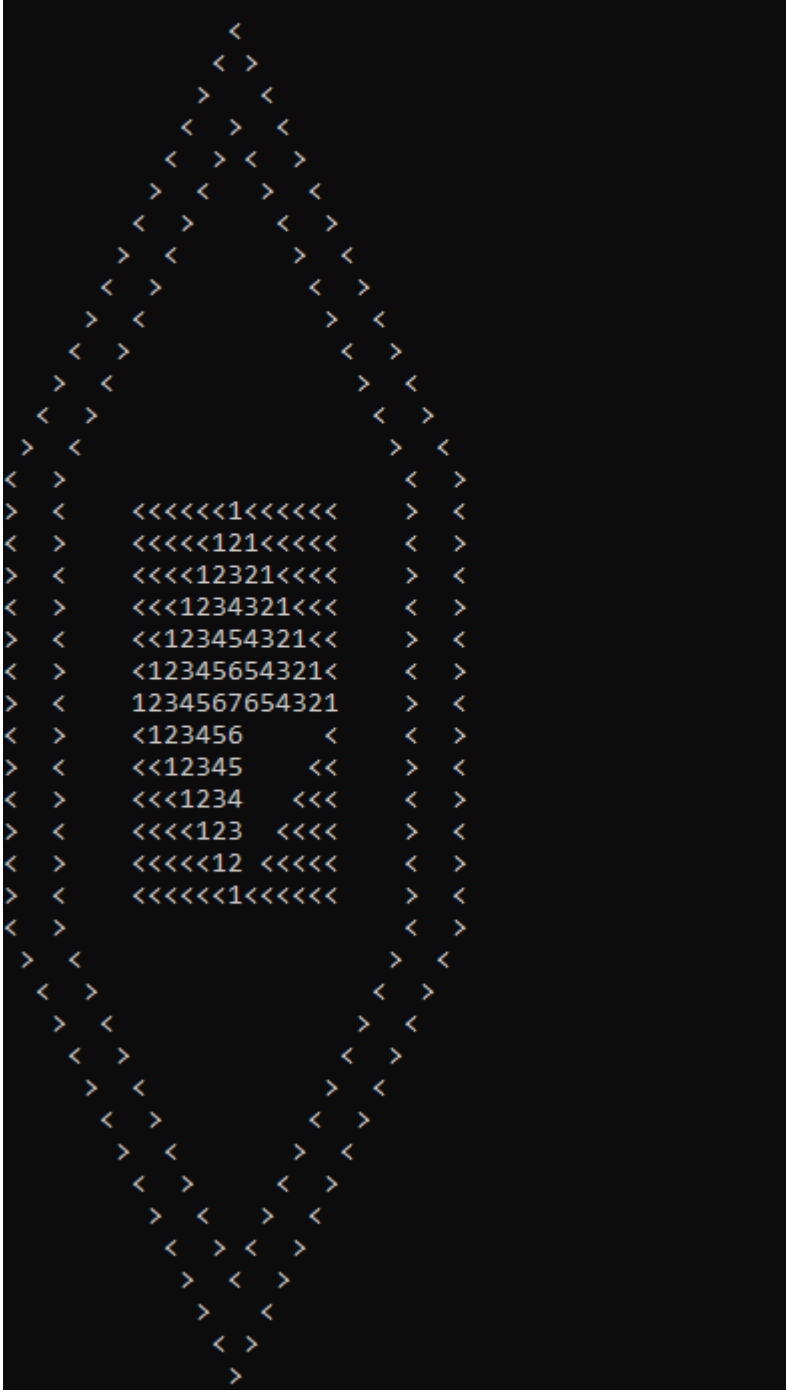
- **Input Specifications:**
  - You need to input **the number of lines** (**n**) which will define the height of the diamond. Where *(n >= 5)*
  - For the borders of the shape, input two characters: these characters will be printed in an alternating pattern as show in the example
- **Output**
  - Output should be the same as shown in the examples below. It is part of the assignment to calculate the size of each sub-shape(s) and the spaces between characters.
- **Examples**

```
Enter the number of lines you want to print : 30
For boders:
 Enter your first character : +
 Enter your second character : =

          +
         + =
        = + =
       = + = +
      + =   + =
     = +     = +
    + =       + =
   = +         = +
  + =           + =
 = +             = +
+ =               + =
= +   ++++1++++   = +
+ =   +++121+++   + =
= +   ++12321++   = +
+ =   +1234321+   + =
= +   123454321   = +
+ =   +1234   +   + =
= +   ++123  ++   = +
+ =   +++12 +++   + =
= +   ++++1++++   = +
+ =               + =
 = +             = +
  + =           + =
   = +         = +
    + =       + =
     = +     = +
      + =   + =
       = + = +
        + = +
         + =
          =
```

```
Enter the number of lines you want to print : 20
For boders:
 Enter your first character : &
 Enter your second character : $

      &
     &$&
    &$ &$
   $&   $&
  &$     &$
 $&       $&
&$         &$
$&  &&1&&  $&
&$  &121&  &$
$&  12321  $&
&$  12321  &$
$&  &12 &  $&
&$  &&1&&  &$
$&         $&
 &$       &$
  $&     $&
   &$   &$
    $& $&
     &$&
      &
```

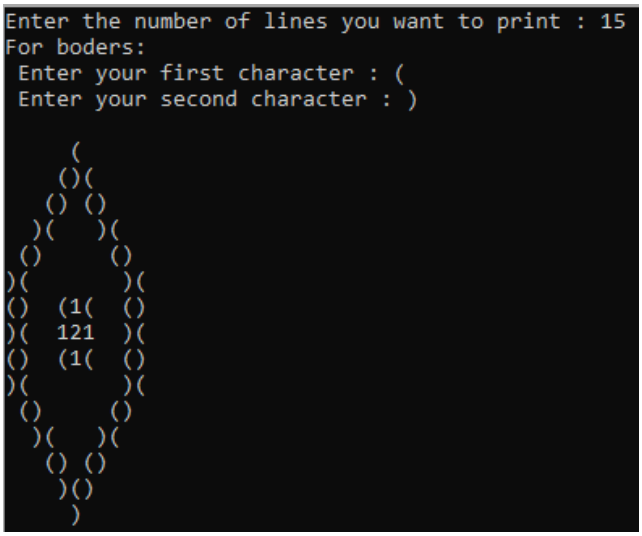
```
Enter the number of lines you want to print : 5
For boders:
 Enter your first character : {
 Enter your second character : }

 {
{ }
} {
 {
```

```
Enter the number of lines you want to print : 43
For boders:
 Enter your first character : <
 Enter your second character : >
```




```
Enter the number of lines you want to print : 10
For boders:
 Enter your first character : @
 Enter your second character : ?
```

```
Enter the number of lines you want to print : 15
For boders:
 Enter your first character : (
 Enter your second character : )
```



```
Enter the number of lines you want to print : 25
For boders:
 Enter your first character : |
 Enter your second character : |
```