CS-1002 Programming Fundamentals
CS(A, B, C, D, E, F, G, H and J)

Fall 2025: Assignment 3

The learning outcomes of this assignment are to teach how to use loops, arrays and functions and how exactly these constructs are used in day to day applications. The main objective is not only to teach and test the concepts in these constructs but to build your intuition towards their applications and how each and every complex application can be broken down into smaller, basic level steps.

Instructions

I. Displayed output should be well mannered and well presented. Use appropriate comments and indentation in every code segment; anyone caught using AI-generated comments shall receive 0 marks in that question.
II. Combine each question into a single switch statement such that running the .exe file should first ask for input to decide which question shall be run. If a question is attempted on a separate .cpp file, marks will be deducted.
III. Run and test your program on a lab machine before submission.
IV. Make a single .cpp file and name it as ROLL- NUM_SECTION.cpp e.g (25i-0001_A.cpp). The file should contain your name and id on the top of the file in comments.
V. Combine all your work into one folder named ROLL-NUM_SECTION, with only .cpp and .pdf files. Compress it into a .zip file, not .rar, named ROLL-NUM_SECTION.zip
VI. Submit the .zip file on Google Classroom within the deadline.
VII. Submissions other than Google classroom (e.g. email etc.) will not be accepted.
VIII. 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.
IX. For Pattern Printing (Basic) you are not allowed to make functions.
X. For all the questions, you are not allowed to use any libraries(e.g iomanip, bitset, cmath etc) except **iostream** unless explicitly mentioned in the question or in the general instructions**.**
XI. For **Q5, and Q6** you are allowed to use fstream along with iostream.
XII. For **Q2, Q4, Q5, Q6 and Q7,** you are highly encouraged to use functions, in order to enhance modularity, reduce code redundancy, improve maintainability, and manage complexity.
XIII. You are not allowed to use **Global Variables** in any of the questions.

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

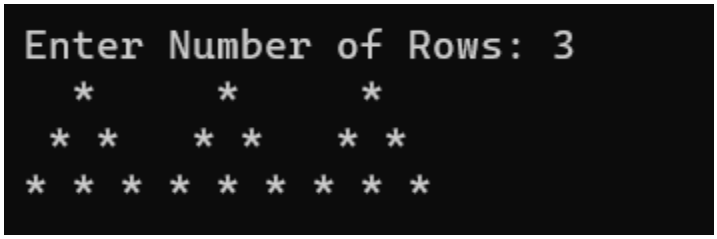
Plagiarism: Plagiarism is not allowed. If found plagiarized, you will be awarded zero marks in the assignment.
Note: Please follow all the given instructions carefully. Failure to do so may result in a substantial deduction of marks.

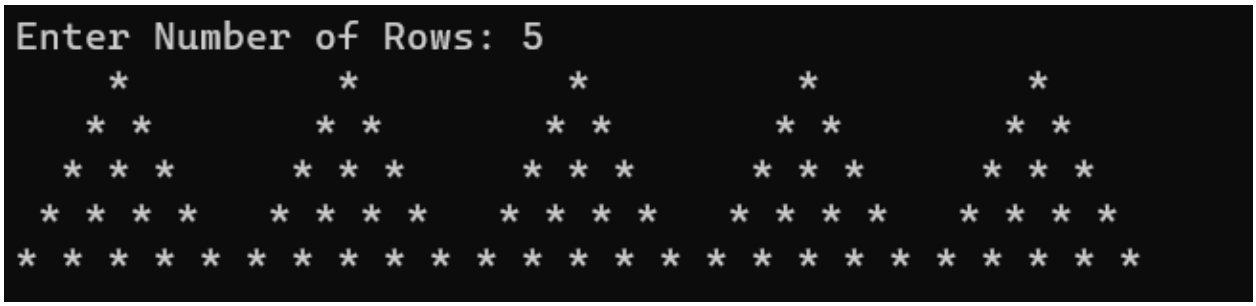
DEADLINE:
The deadline to submit the assignment is **November 16,2025 11:59 PM**. You are supposed to submit your assignment on GOOGLE CLASSROOM (Lecture TAB not lab). Only ".ZIP" files are acceptable. Other formats will be directly given 0. Correct and timely submission of the assignment is the responsibility of every student, hence no relaxation will be given to anyone. Late Submission policy will be applied as described in the course outline.

Q1: Patterns (Basic) [60]

Part I [20]:

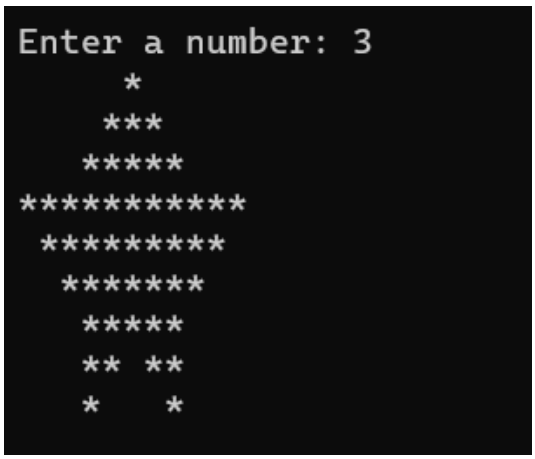
Write a program that prints the following pattern based on the value of the input. **For this you can only use do while loops. Zero marks will be awarded to those who used other iterative structures.**

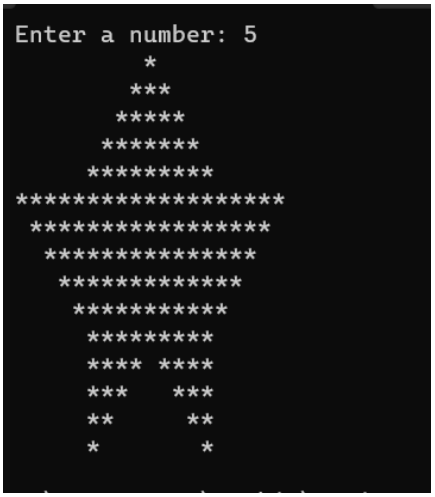




Part II [20]:

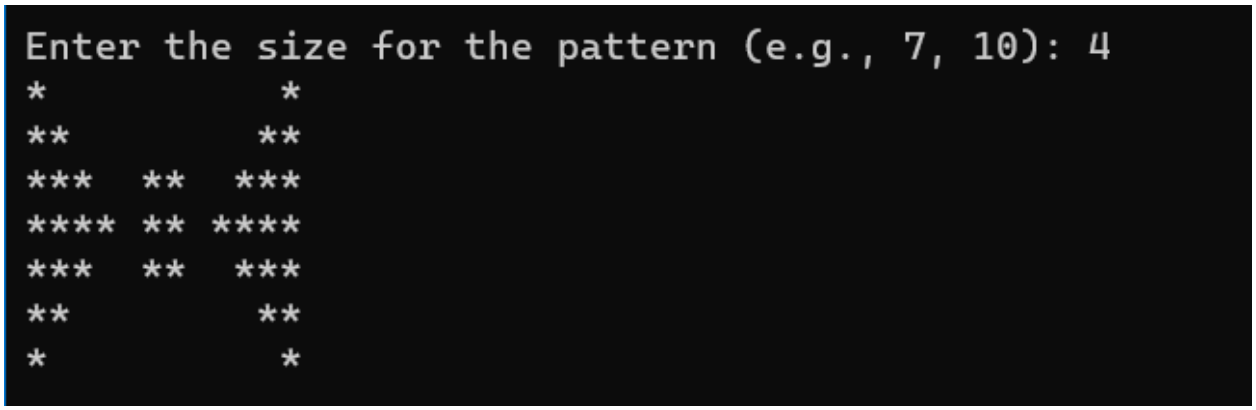
Write a program that prints the following pattern based on the value of the input. **For this you can only use while loops. Zero marks will be awarded to those who used other iterative structures.**

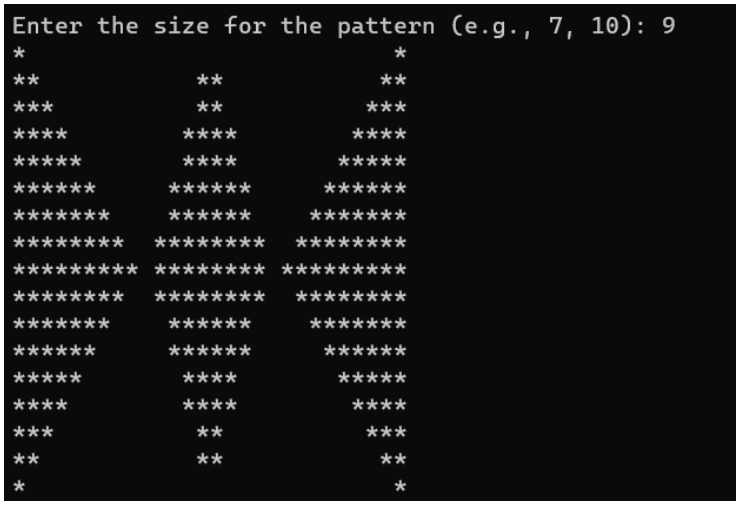




Part III [20]:

Write a program that prints the following pattern based on the value of the input. **For this you can only use for loops. Zero marks will be awarded to those who used other iterative structures.**





Q2: Patterns (Advanced) [60]

PART I [30]:

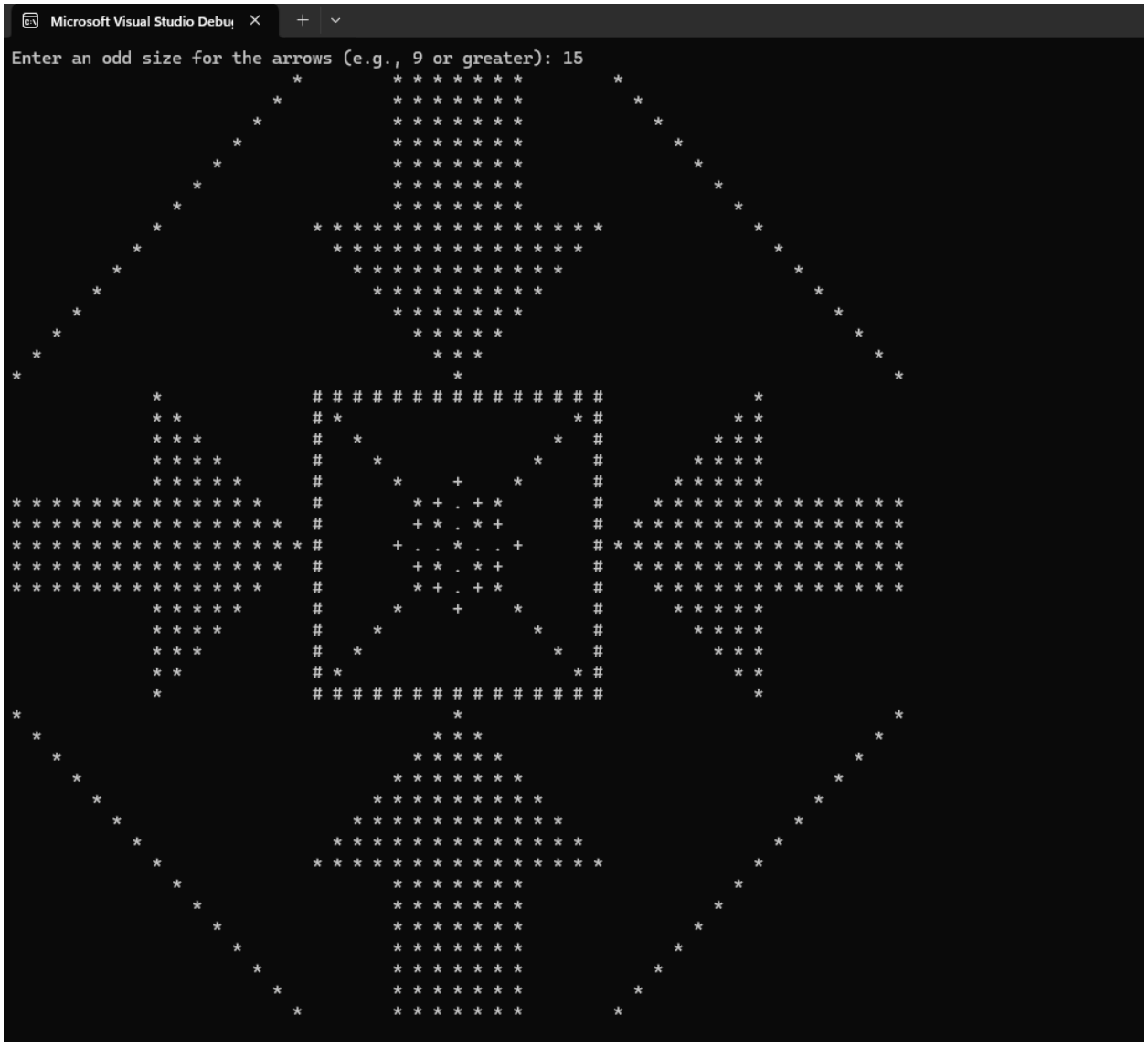
Carefully analyze the sample outputs given below and write a C++ program that generates the same pattern with the following constraints enforced:

1) Input needs to be greater than 9

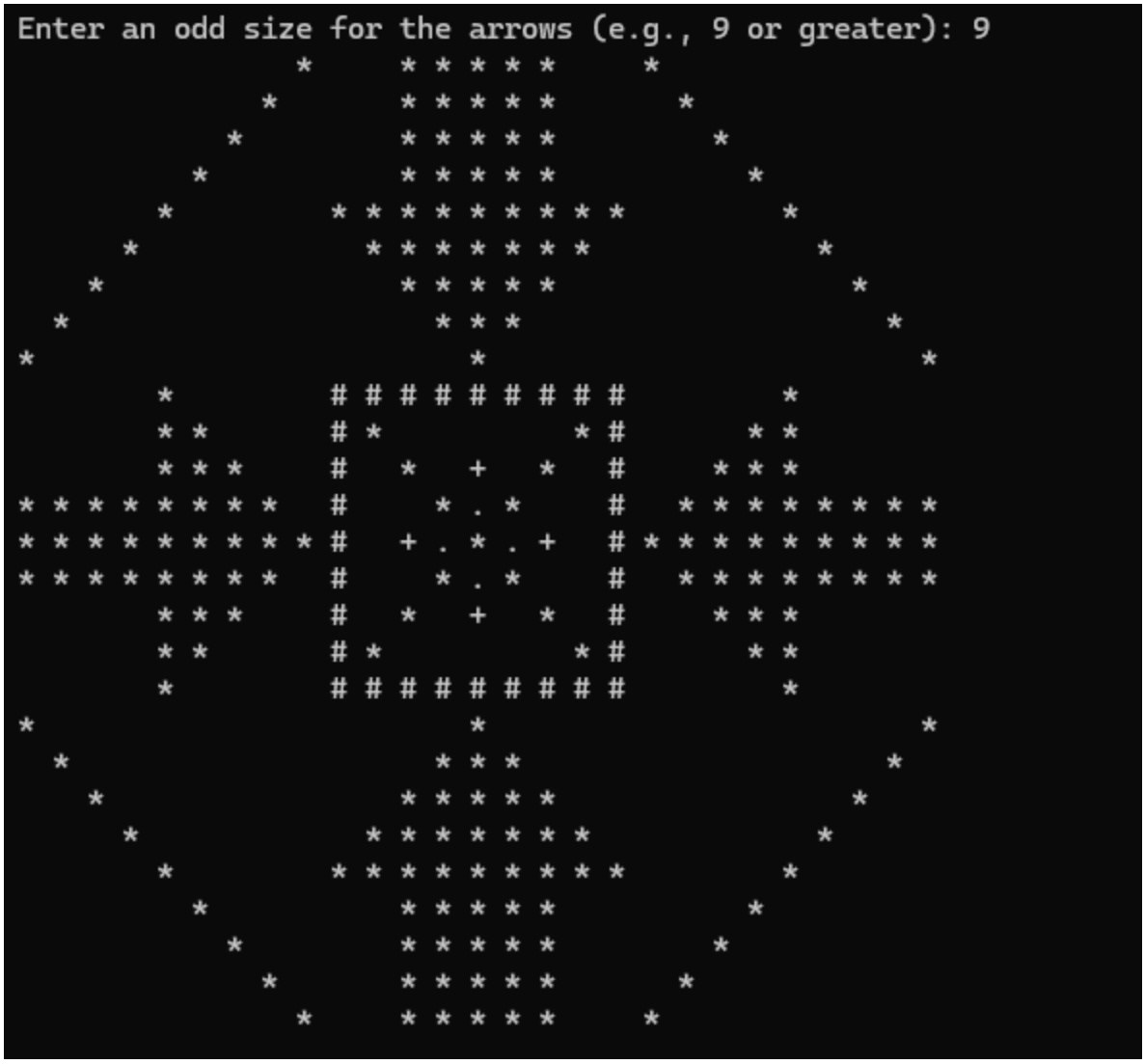
2) Input can never be even.

Example Outputs:

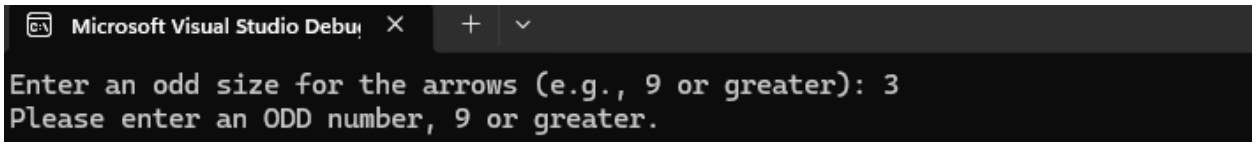
Input = 15:



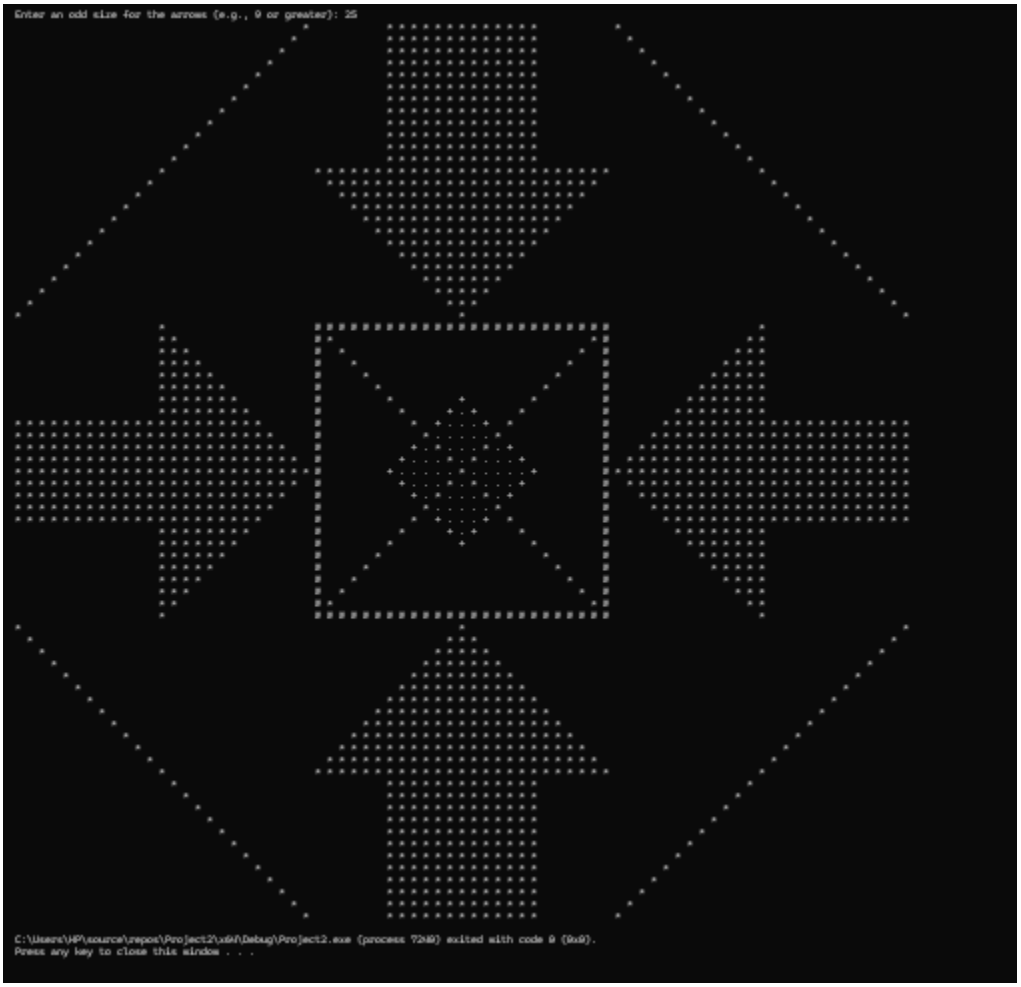
Input = 9:


Enter an odd size for the arrows (e.g., 9 or greater): 9

Input = 3:


Microsoft Visual Studio Debu
Enter an odd size for the arrows (e.g., 9 or greater): 3
Please enter an ODD number, 9 or greater.

Input = 25:



PART II [30]:

A **14-segment display** is an electronic display device used to represent alphanumeric characters (letters and numbers) by illuminating specific combinations of 14 individual segments arranged in a rectangular pattern. Each segment is typically an LED or LCD element that can be turned on or off independently. By selectively activating these segments, the display can form not only digits (like a 7-segment display) but also **letters** and various **symbols**.

Refer to this wikipedia article to further understand the works of a 14-segment display: Fourteen-segment display - Wikipedia

Your job is to create a pattern using this concept. The logical flow is as follows:

- Take input from the user and store it in a char array.
- Display the input in the manner shown in the example outputs below.

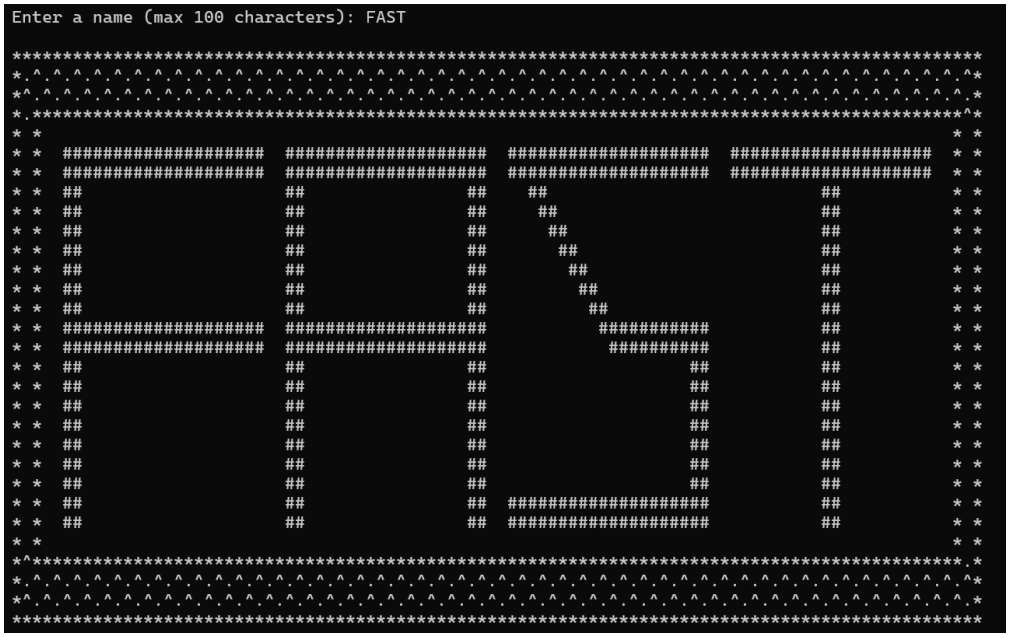
- Use a general formula and proper encoding and decoding techniques on the input to generate the resulting pattern.
- The height of each character is fixed at 20 in the example outputs. Your implementation should work for any given height.
- Encapsulate the text with the border in the samples.

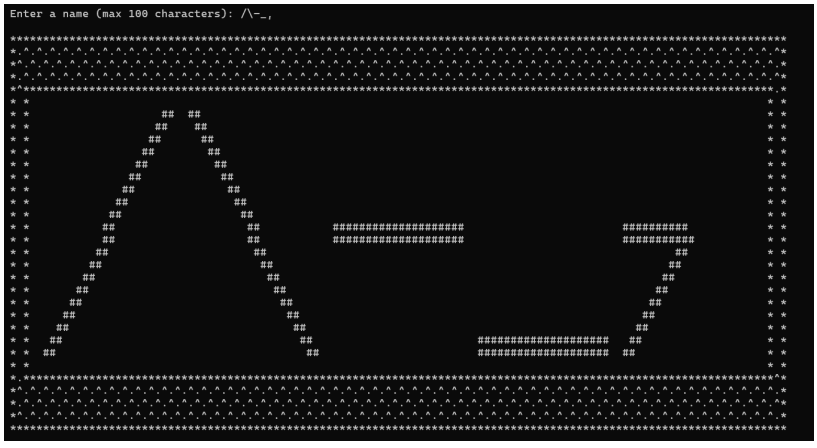
Ensure that the pattern follows the logic and working of a 14 segment display. **Hardcoding the pattern will result in 0 marks.**

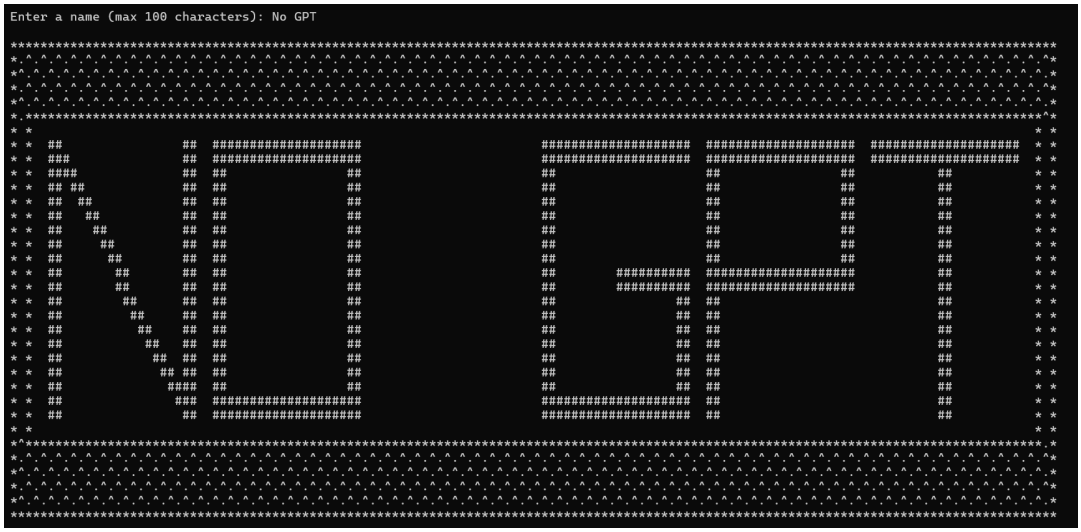
Allowed character: Upper case alphabets, numbers (0 - 9), (/ \ , - _)

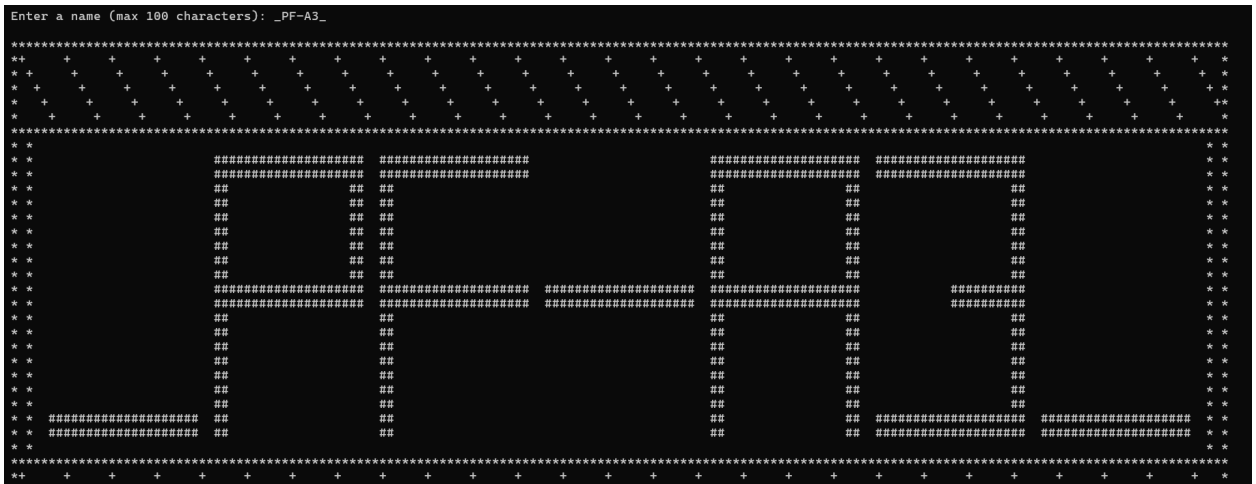
Note that the border must be the same as given in the sample outputs below

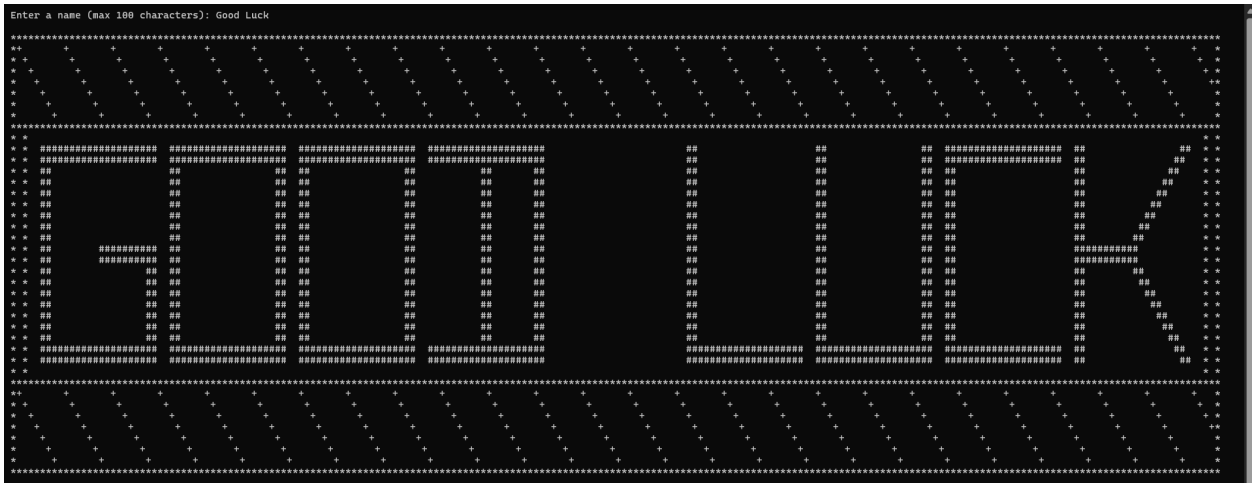
Example Outputs:






Enter a name (max 100 characters): No GPT


Enter a name (max 100 characters): _PF-A3_


Enter a name (max 100 characters): Good Luck

Q3: A Matter of Points [50] (Evaluation from Hackerrank)

You are given an array **a** of **n** positive integers. You will play a game with this array along with your TA and take alternating turns, with you going first.

At each turn, the player must choose a value **x** greater than 0 that appears at least once in **a** and the player earns one point for each occurrence of **x** in the array.

Then for every position **i** where $\mathbf{a_i = x}$, set $\mathbf{a_i = x-1}$. However, if the original **x** was divisible by the number of its occurrences in the array, then all those positions are instead set to zero**.**

After these updates, every element of **a** that was odd at the start of the turn will also decrease by 1 and the game will end when no **x** can be chosen anymore.

Given that both players want to maximize their points and play optimally, calculate the amount of points that each player will end up with and the number of turns it will take them to end the game.

Input:

The first line of input contains an integer **n** ($1 \le \mathbf{n} \le 10^5$) - the size of the array.

The second line of input contains **n** integers $\mathbf{a_1, a_2, \ldots, a_n}$ ($1 \le \mathbf{a_i} \le 10^9$) - the elements of the array.

Output:

The first line of output contains the number of combined turns it took to play the complete game.
The second line of output contains two integers, the amount of points you and your TA will get respectively.

Submission Instructions:

- Carefully read the problem statement
- Implement your solution only in c++
- Make sure that your code produces the exact output format as described above
- After testing your program locally, upload your solution to the hackerrank link provided.
- You are required to submit this problem only on hackerrank using the link provided. Submission through the GCR (for this problem only) will not be accepted.
- Submissions that do not match the output format, use hardcoded outputs, or fail hidden test cases will not be accepted.
- Hackerrank account must be made from university email and your hackerrank username must be in the format i25XXXX

Hackerrank Submission Link: https://www.hackerrank.com/a-matter-of-points

Q4: Memory Pattern [60]

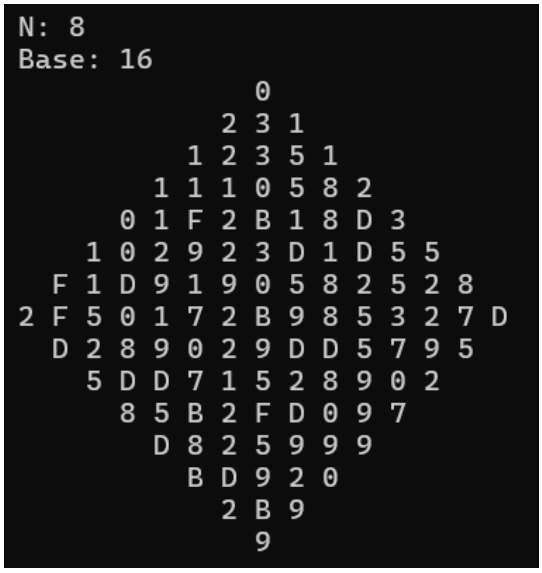
Analyze the sample outputs given below and identify the formulas and relations and write a C++ program that generates the same patterns for any n input. You are not allowed to use arrays. The program must display the pattern numbers in the user-specified number system (from binary till hexadecimal).

Example Outputs ():

```
N: 3
Base: 7
    0
  6 0 1
1 5 4 6 1
  5 6 2
    3
```

```
N: 4
Base: 8
      0
    1 0 1
  7 5 5 1 1
2 0 1 0 2 1 2
  5 5 7 2 3
    5 3 5
      0
```

```
N: 5
Base: 13
        0
      C C 1
    0 1 0 B 1
  C C 0 8 1 A 2
1 2 8 B 1 8 1 8 3
  B A 5 3 2 5 5
    3 5 3 0 8
      8 5 0
        8
```



HINT: The pattern expands in the form of a spiral. The numbers expand in the following way: fibonacci sequence % base.

Q5: Building Los Alamos: The SCUR Protocol [60]

It’s the year 1942. It has been 3 years since World War II started. Following the attack on Pearl Harbor the USA has thrown its full industrial and scientific might into the conflict. In the highest echelons of government, a terrifying race has begun against the Axis powers: a race to build an atomic weapon. Under the utmost secrecy, a new endeavor, codenamed the "Manhattan Project," is recruiting the nation's most brilliant minds. These scientists and engineers are being sent to a remote mesa in New Mexico—a place that doesn't officially exist—to build a city and a laboratory from scratch. The speed required is unprecedented, and the need for absolute secrecy is paramount, forcing every process, from designing labs to building barracks, to be re-imagined.

You, being the most talented architect in the team, receive this job:

Technical Details:

The **Structure Component Unified Representation (SCUR)** system that works on the following flow of data upon which a certain algorithm will apply processing to output information.

The Input Data:

You’ll be provided with a “.txt” file in which the first 3 numbers will represent width, length, and height of the blue prints followed by the top down blue print itself.

The blue print will be based around the following legend:

- `X` → Solid wall
- `-` → Horizontal window
- `|` → Vertical window
- `‘ ’(Space)` → Door

(Test inputs are provided in text files)

Processing:

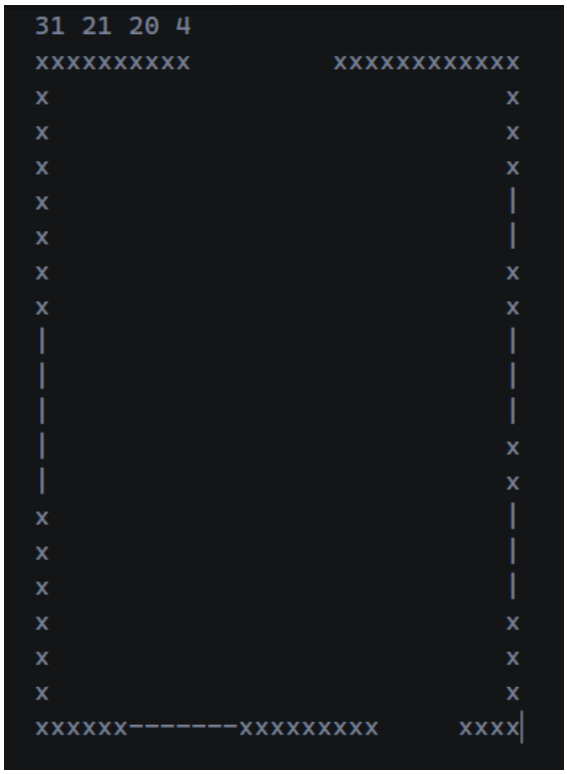
First you have to input the blueprint from the text file into a 2d Array and you are only allowed to traverse each text file only once (if you traverse the file more than once, marks will be deducted).
Then the SCUR system will display the inputted blue print and show a menu where you can choose to perform the following parsing on a selected file based on the following legend:

- 1 → top wall
- 2 → left wall
- 3 → bottom wall
- 4 → right wall

Elevation rendering directives

Once a wall is selected, A directive for procedural generation of elevation based on the height inputted from the file will be performed and the resultant 2d Array of the side on view of the wall will be displayed.

Sample Output (The provided text file):



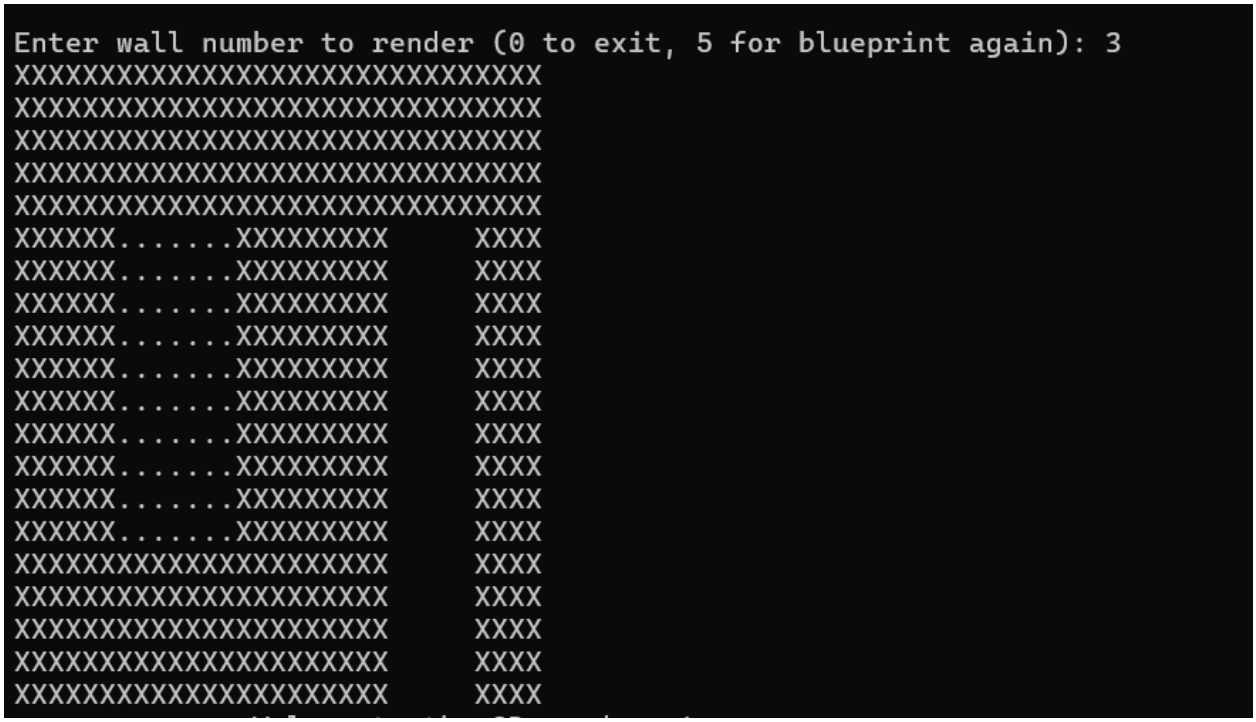
OUTPUT:

```
--------------Welome to the 3D renderer!------------

BLUEPRINT DIMENSIONS: Width: 31 Length: 21 Height: 20
xxxxxxxxxx         xxxxxxxxxxx
x                           x
x                           x
x                           x
x                           |
x                           |
x                           x
x                           x
|                           |
|                           |
|                           |
|                           x
|                           x
x                           |
x                           |
x                           |
x                           x
x                           x
x                           x
xxxxxx-------xxxxxxxxx     xxxx


Enter wall number to render (0 to exit, 5 for blueprint again):
```

```
Enter wall number to render (0 to exit, 5 for blueprint again): 1
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
XXXXXXXXXX          XXXXXXXXXXX
```



Q6: Image Editing and Enhancements [70]

Introduction

You are an imaging specialist at NASA responsible for reading and enhancing images transmitted from various satellites. Your mission is to write a C++ program that processes 64x64 image fragments sent from the Voyager 1 probe. Each image is a matrix of 64 rows and 64 columns for each channel namely **alpha, beta, and gamma.**

Core Data Structure [5]

The **Image Matrix** is represented as a static 3D integer array with a fixed size: `int matrix[64][64][3];`

- The first dimension is the row (`64` total).
- The second dimension is the column (`64` total).
- The third dimension holds the 3 energy channels for each **row and col index**:
  - `matrix[row][col][0]` = Alpha channel (α)
  - `matrix[row][col][1]` = Beta channel (β)
  - `matrix[row][col][2]` = Gamma channel (γ)

Each channel's value is an integer ranging from `0` (no energy) to `255` (maximum energy).

The Visualization Protocol (Rendering) [5]

To visualize the `Image Matrix`, you must send a series of low-level command strings directly to the console. These commands, known as **ANSI escape sequences**, control the color of each character cell.

The specific command for setting a 24-bit background color is a string with the following format: `\033[48;2;<alpha>;<beta>;<gamma>m`

Here is a breakdown of the command:

- **`\033[`**: This is the **Control Sequence Introducer**. It signals to the terminal that a special command is beginning.
- **`48`**: This is the code for setting the **background color**.
- **`;2`**: This sub-code specifies the color mode. `2` means you are providing a full 24-bit RGB color.
- **`;<alpha>;<beta>;<gamma>`**: These are the integer values for the three energy channels, in order.
- **`m`**: This final character **executes the command**.

After setting a color for a nexus point, you must immediately reset the terminal's color settings. Otherwise, the color will "bleed" across the rest of the console. The reset sequence is: `\033[0m`.

Therefore, to render a single pixel, you must print the color sequence, 2 space (because 1 character is double the height as its width) to act as the pixel, and then the reset sequence. For a a single index with channel values `{α=64, β=128, γ=255}`, your C++ code would be:

```
cout << "\033[48;2;64;128;255m" << "  " << "\033[0m"; for 1 pixel
```

Your Task

Implement a C++ program that:

1. Loads 5 fixed-size (`64x64`) energy matrix fragments.
2. Presents a menu to select a fragment.
3. For a selected fragment, enter an **Analysis Menu** for performing the operations below.

4. Includes a rendering function that you write yourself. This function must iterate through the entire matrix and print the correct ANSI escape codes for each pixel to display the full image.

Operational Directives

You must implement the following matrix operations. After each operation, the updated matrix must be rendered to the console.

1. Convert to Monochrome [10]

- **Specification:** For each pixel `(row, col)`, calculate the average of its three channels: `avg = (alpha + beta + gamma) / 3`. Set all three channels to this `avg` value.

2. Change the brightness by a a value k [10]

- **Specification:** Add or subtract a fixed value (e.g., `20`) from every channel of every nexus point.
- **Constraint:** The final value of any channel must be "clamped" within the `0-255` range.

3. Increase / Decrease contrast [10]

- **Specification:** Increase or decrease the channels' values.
  - **Increase:** `newValue = (1.2 * (currentValue - 128)) + 128`.
  - **Decrease:** `newValue = (0.8 * (currentValue - 128)) + 128`.
- **Constraint:** Clamp the `newValue` to the `0-255` range.

4. Rotation [10]

- **Specification:** Re-align the matrix's axis by 90 degrees clockwise.

5. Blurring [10]

- **Specification:** Smooth out the noise by calculating the weighted average using the kernel matrix given below.
  1 2 1
  2 4 2
  1 2 1

- Sum the values of the 9 pixels in the grid multiplied by their kernel weight, then divide by the total weight (`16`).

6. Sharpening [10]

- **Specification**: A kernel matrix that amplifies the details within a matrix. Use the kernel matrix below and apply this to each of the pixels in the matrix.

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

- **Constraint**: The total weight of this kernel is 1. The resulting value from the operation must be clamped to the **0-255** range, as the calculation can produce negative values or values over 255.

Example walk through of Blurring:

Let's start with a random 5x5 **Image Matrix**. For simplicity, we will only look at the **Alpha (α)** channel values.

Initial Matrix (Alpha Channel)

80	95	100	115	120
75	180	**210**	190	110
70	175	255	185	105
65	150	240	160	100
60	85	90	105	110

Our goal is to calculate the new, blurred value for the pixel **210**. We will use the 3x3 grid of its neighbors and the specified kernel.

The Kernel

The weighted kernel for Nexus Diffusion is:

1	2	1
2	4	2
1	2	1

The **total weight** of the kernel is `1 + 2 + 1 + 2 + 4 + 2 + 1 + 2 + 1 = 16`.

Iteration on a Single Point

To find the new value for the point `(row=1, col=2)`, which is **210**, we perform a weighted average.

1. **Match Kernel to Neighbors**: Place the kernel over the 3x3 grid surrounding our target point.
Matrix Values	Kernel Weights
95, 100, 115	1, 2, 1
180, **210**, 190	2, **4**, 2
175, 255, 185	1, 2, 1
2. **Multiply and Sum**: Multiply each matrix value by its corresponding kernel weight and add them all together.
    - **Top Row:** `(95 * 1) + (100 * 2) + (115 * 1) = 95 + 200 + 115 = 410`
    - **Middle Row:** `(180 * 2) + (210 * 4) + (190 * 2) = 360 + 840 + 380 = 1580`
    - **Bottom Row:** `(175 * 1) + (255 * 2) + (185 * 1) = 175 + 510 + 185 = 870`
3. **Total Sum** = `410 + 1580 + 870 = 2860`
4. **Divide by Total Weight**: Divide the total sum by the kernel's total weight (16) to get the final blurred value.
   New Value = `2860 / 16 = 178.75`
5. **Final Integer Value**: Since the matrix stores integers, we truncate the decimal. The new Alpha value for this nexus point is **178**.

After performing this process on all non-border points, the value `210` would be updated to `178`, resulting in a smoother transition between it and its neighbors.

Note that you'll be provided with an "images.txt" file that contains all the channels of all the pixels in all the rows in a single line in row major format such that 1 line corresponds to 1 image. You'll have to manually parse each channel of each pixel into the image matrix's appropriate row and column and the appropriate channel.

SAMPLE OUTPUT is attached on GCR

Q7: Project G-LOC: Guarded Link Over Combat [70]

Mission Briefing
Date: 03 May 2025
From: PAF Intelligence Command
To: [CLASSIFIED OPERATIVE]
Subject: Project G-LOC (Guarded Link Over Combat)

Tensions with the Indian Federation have reached a breaking point following the Pulwama incident. Hostile rhetoric has escalated, and all assets are on high-alert. Intelligence indicates a high probability of localized skirmishes escalating into full-scale dogfights.

Standard communication channels are compromised. We require a dynamic, asymmetric encryption system to secure real-time tactical messages sent to our pilots (e.g., "Bandit, 3 o'clock, low," "RTB, Winchester").

Your objective is to build the C++ prototype for this system. It must generate secure keys based on pilot credentials and be robust enough to ensure that even if a data stream is breached, it remains indecipherable. You are the architect of this new "digital shield." Failure is not an option.

I.F.F. Seed Generation [5]

The first process is generating the I.F.F. (Identify Friend or Foe) Handshake Primes. The system must ask the user for the their 4-digit ID (e.g., 1234) and store it inside a 64-bit unsigned integer in a repeating pattern:

abcdabcdabcdabcd, where abcd are the 4 digits.

Once stored, the script must loop through this 64-bit number bit-by-bit, counting the number of 1's in **even** bit positions (let e) and 1's in **odd** bit positions (let o).

From these counts, we get the eth and the oth prime numbers. These two primes become our **Handshake Primes (P1 and P2)**.

- **Note:** If $e < 5$, set $e = 5$.
- **Note:** If $o < 5$, set $o = 5$.

The Session Modulus (N) [5]:

The Session Modulus (N) is calculated by first refining the two Handshake Primes and then multiplying them. This N will be the foundation for all cryptographic operations.

The refining process uses the e and o counts:

- **If e >= o:**
  - Set RefinedPrime = RawPrime
  - Continuously add (e % 5) + 1 to RefinedPrime
  - Break when RefinedPrime becomes prime again
  - If RefinedPrime ever goes above 61, subtract 50 from it
- **If e < o:**
  - Set RefinedPrime = RawPrime
  - Continuously subtract (o % 5) + 1 from RefinedPrime
  - Break when RefinedPrime becomes prime again
  - If RefinedPrime ever goes below 11, add 50 to them
- **Clamping:** If after refinement any prime is < 11, make it 11. If any prime is > 61, make it 61.
- **Uniqueness:** If after refinement P1 == P2, increment P2 until it becomes the next prime number.

Finally, Session Modulus = Refined_P1 * Refined_P2.

System Totient [5]

The System Totient is a critical internal value derived from the refined primes. It is calculated by decreasing each of the refined primes by 1 and then multiplying them.

System Totient = (Refined_P1 - 1) * (Refined_P2 - 1)

The Binding Exponent [5]

The Binding Exponent is the public part of the key. It's calculated by finding the kth number that is co-prime to the System Totient.

- Note that k = e + o (from the bit counts).

The Unbinding Exponent [5]

The Unbinding Exponent is the private part of the key. It's calculated by finding a number n between 1 and the System Totient that satisfies the following condition:

(n * Binding Exponent) % System Totient = 1

The Decryption Keypair [2.5]

The Decryption Keypair is the complete private key. It must be loaded securely into the pilot's avionics. It consists of:

Decryption Keypair = { Session Modulus, Unbinding Exponent }

The Encryption Keypair [2.5]

The Encryption Keypair is the public key, which command-and-control will use to send messages to the pilot. It consists of:

Encryption Keypair = { Session Modulus, Binding Exponent }

Encryption Protocol: "Vector" [20]

This protocol "twists" the tactical message. Each character of the message undergoes transformation via the following formula:

Encrypted_Number = ((character's ASCII value) ^ (The Binding Exponent)) % The Session Modulus

You must find the maximum number of digits this Encrypted_Number could possibly have (based on the maximum possible Session Modulus) and store it inside a character array of that appropriate, fixed size.

- **Padding:** If the number of digits in the Encrypted_Number is less than the maximum, pad the *start* of the character array with zeros.
    - **Example:** For a max digit count of 6, the number 123 becomes ['0', '0', '0', '1', '2', '3'].

All encrypted character arrays must be combined into a singular 1D data stream, with each number-array separated by the _ symbol.

- **Example Stream:** ['0', '0', '0', '1', '2', '3', '_', '0', '0', '0', '1', '2', '4', ...]

After processing, the system must display the final encrypted data stream.

Decryption Protocol: "Angels" [20]

This protocol "untwists" the breached or received data stream. The process involves:

1. Extracting the sub-arrays by splitting the stream at the _ symbol.
2. Converting each sub-array (e.g., ['0', '0', '0', '1', '2', '3']) back into its integer form (e.g., 123).
3. Transforming the integer back into the original character's ASCII value using the **Decryption Keypair**:

Original_ASCII = ((Encrypted_Number) ^ (The Unbinding Exponent)) % The Session Modulus

After transformation, each character is added to the unbound message array one by one. The system must display the final, decrypted tactical message.

Sample Output:

```
Enter the 4-digit Squadron ID (e.g., 1234): 0774
---------------------------------
Decryption Keypair: {527, 293}
Encryption Keypair: {527, 77}
---------------------------------

Enter tactical message to encrypt: This was previously lord of the rings

Encrypting message...
Encrypted Data Stream: 0322_0406_0352_0353_0342_0068_0388_0353_0342_0113_0303_0033_0067_0352_0076_0478_0353_0333_0474_0342_0333_0076_0303_0359_0342_0076_039
1_0342_0277_0406_0033_0342_0303_0352_0145_0069_0353

Decrypting message...
Enter encrypted data stream: 0322_0406_0352_0353_0342_0068_0388_0353_0342_0113_0303_0033_0067_0352_0076_0478_0353_0333_0474_0342_0333_0076_0303_0359_0342_00
76_0391_0342_0277_0406_0033_0342_0303_0352_0145_0069_0353

Decrypted Tactical Message: This was previously lord of the rings
```

```
Enter the 4-digit Squadron ID (e.g., 1234): 0767
---------------------------------
Decryption Keypair: {799, 479}
Encryption Keypair: {799, 63}
---------------------------------

Enter tactical message to encrypt: Rafael is a potent aircraft if employed well

Encrypting message...
Encrypted Data Stream: 0181_0707_0408_0707_0288_0122_0535_0550_0106_0535_0707_0535_0097_0648_0436_0288_0729_0436_0535_0707_0550_0180_0555_0180_0707_0408_043
6_0535_0550_0408_0535_0288_0311_0097_0122_0648_0196_0288_0025_0535_0034_0288_0122_0122

Decrypting message...
Enter encrypted data stream: 0181_0707_0408_0707_0288_0122_0535_0550_0106_0535_0707_0535_0097_0648_0436_0288_0729_0436_0535_0707_0550_0180_0555_0180_0707_04
08_0436_0535_0550_0408_0535_0288_0311_0097_0122_0648_0196_0288_0025_0535_0034_0288_0122_0122

Decrypted Tactical Message: Rafael is a potent aircraft if employed well
```

```
Enter the 4-digit Squadron ID (e.g., 1234): 9182
---------------------------------
Decryption Keypair: {713, 623}
Encryption Keypair: {713, 107}
---------------------------------

Enter tactical message to encrypt: The Silent Guy was GOATED

Encrypting message...
Encrypted Data Stream: 0663_0468_0312_0404_0148_0600_0426_0312_0269_0277_0404_0670_0509_0133_0404_0285_0605_0322_0404_0670_0021_0704_0663_0483_0367

Decrypting message...
Enter encrypted data stream: 0663_0468_0312_0404_0148_0600_0426_0312_0269_0277_0404_0670_0509_0133_0404_0285_0605_0322_0404_0670_0021_0704_0663_0483_0367

Decrypted Tactical Message: The Silent Guy was GOATED
```

Note that this is just a sample output just for the purpose of understanding. You will have to implement a proper main with well formatted outputs and a menu where the user may choose to bind or unbind.

Q8: Bubble Bombing Bricks (Triple B) [70 marks]

Watch the video uploaded with the assignment. You have to recreate that game using only the concepts taught in class (Not bitset), ANSI escape sequences for colors and functions within certain libraries:

- **iostream:** cout, endl
- **unistd.h:** usleep(), read(), STDIN_FILENO
- For functions given in the starter code for hiding cursor: you are not allowed to use them except for where they are already used.

Anyone using any other external will be marked 0 in this whole assignment.
You must be able to explain why and where you used the allowed functions above and what exactly these functions do.

Important Note: Your code **MUST be modular**. This means your program should be **divided into well-defined functions**, each handling a specific task. You are required to write up a draft wherein you detail each of the mechanics of the game that need to be implemented, then group similar functionalities together and wrap them in a function. It is up to you how you go about it. Ensure that your code reflects the exact structure of the draft. **You will be cross-checked during the demo.**

Note that the implementation of 1 level is mandatory for the bonus to be marked.

Bonus

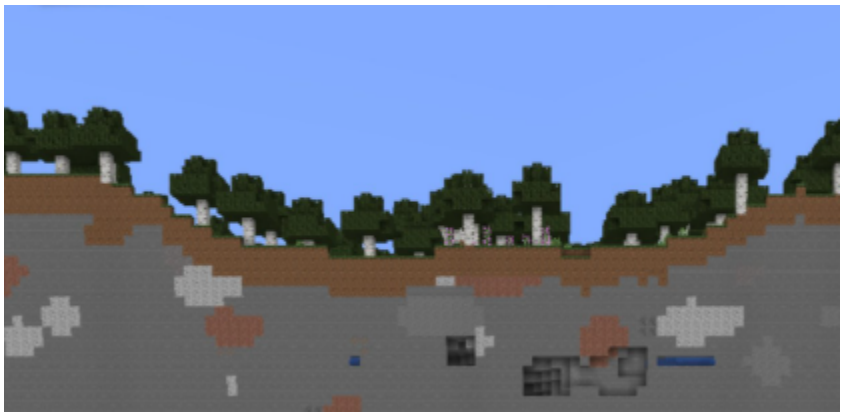
Bonus marks will be awarded for the addition of 2 additional levels, a proper menu, a high score tracking system and the saving of high scores in a text file using filehandling.

Bonus Question: Procedural Terrain Generation (The Great Filter)

Have you ever wondered how games like **minecraft, and terraria** have infinitely vast and unique maps yet they take minuscule space in terms of memory. For reference 1 minecraft world is 8 times the surface area of the Earth yet, and there are 18 quintillion of these worlds yet minecraft's download size is a mere 0.6 GB.

Well, the first thing that comes to mind is that a random height may be chosen at each coordinate and all the blocks get filled below it. Well if you translated it to 2D it would look like this:

And minecraft's worlds are like this:

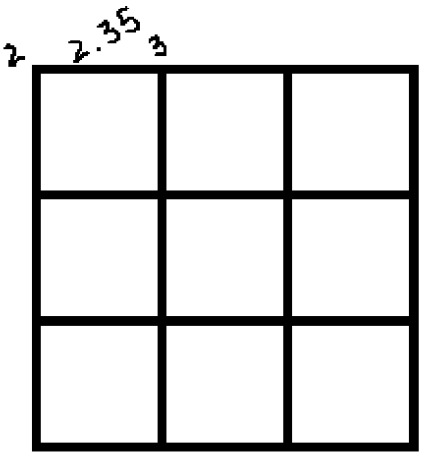


So complete randomness(White Noise) isn't the answer. We have to think of a way to produce a random number that transitions smoothly from its neighbors.

This is where a beautiful algorithm comes into play: **Perlin Noise** by Ken Perlin. It's basically an algorithm that generates natural looking randomness or noise.
Vid 1: How Does Perlin Noise Work?

Now this video demonstrates the pure maths behind this algorithm but in the case of games each height value that is generated randomly isn't in decimal, rather it's a whole integer.

So, for 1D, we first get the seed that acts as the heart of this system (like srand()):



Step 1: The Seed & The "Magic Lookup Table"

First, we get the **seed** from the user. This seed is the "master key" for our entire world.

We use this seed *one time* at the very beginning to build our "magic lookup table"—a big **array** in memory. This array is just the numbers 0 through 255, all shuffled in a random-looking but totally predictable order.

Because we use the *seed* to control the shuffle, the same seed will *always* produce the exact same shuffled array. This is our "deterministic randomness" and it's the most important concept.

Step 2: Finding Our "Noise Coordinate"

For each column of the world, we have an integer `X` coordinate (like 45, 46, 47...). We multiply this by our **frequency** (from the menu) to get a floating-point "noise coordinate."

For example, column `47` might become noise coordinate `2.35`. This floating-point number is the input for our main noise-generating function.

Step 3: The Noise Function's Job

Let's say we pass in `2.35`. The noise function's job is to find a smooth, predictable height value for this *exact* spot. Here's how:

1. **Split the Number:** It splits `2.35` into its integer part (**2**) and its fractional part (**0.35**).
2. **Find the "Boundary Slopes":** The integer part, **2**, tells the function that we are on a number line segment *between* point `2` and point `3`. It now needs to get the pre-defined, "random" slope for both `2` and `3`.
3. **Use the "Magic Table":** It uses the integers `2` and `3` as **array indices** to look up values from our "magic lookup table" (the one we built with the seed). This gives it two

pre-determined, random-looking "gradient" values—one for the "left" point (2) and one for the "right" point (3).

4. **Calculate the Influence:** This is where the fractional part, **0.35**, comes in. It tells us we are 35% of the way from point 2 to point 3. We *could* just do a simple average of the two slopes, but that would create sharp, pointy V-shapes.
5. **Smooth, Fade, and Blend:** To get a natural curve, we first pass our `0.35` into a "smoothing function" (the **fade** algorithm). This "eases in" and "eases out" of the slopes, which is what makes the noise look rounded and organic. Finally, we use this new *smoothed* value to **blend** (or 'interpolate') between the "left" slope and the "right" slope.

The Result

This whole process—split, lookup, fade, and blend—gives us one single, smooth, deterministic number (like `0.418...`), which our code provides as a value between -1.0 and 1.0.

We then take this final value and use it in our `generateChunk` function to calculate the `groundHeight` for that one column. We do this over and over for every column to build the entire world.

Your Task

Your task is to implement a 2D infinitely scrolling (left and right) world that has a block height of 48 blocks and a block width is infinity but divided into chunks of width 16. The program render 5 chunks at a time so you need to develop a system that keeps track of the chunk coordinate and loads/unloads appropriate chunks when moving right or left.

Vid 2: Minecraft terrain generation in a nutshell (Upto 17:35)

The generate chunk function generates each column of the chunk using the perlin noise algorithm but with a catch. Use the minecraft video below as an instruction and develop fractal noise algorithm to implement using 3 different noise maps i.e. continentalness, erosion and peaks n Valleys. So that the terrain generated is even more natural looking.

Why We Use Three Maps

The goal is to create realistic, varied terrain. A single noise function is too simple—it just creates the same repeating hills over and over.

To make it look natural, like in the Minecraft video, we have to combine different types of noise, just like an artist uses different "brushes":

1. A **huge brush** for the main shape (continents vs. oceans).
2. A **medium brush** to decide if it's hilly or flat.
3. A **fine brush** for the small, bumpy details.

Our program does this by calling `getFractalNoise` three separate times with separate lookup tables.

The Three Noise Maps

Here are the three "brushes" we use in our `generateChunk` function:

1. "Continental" Map (The Base Shape) This creates the giant, slow-moving continents and ocean basins.

- **Settings:**
  - `octaves = 1`: We only use one layer. This is what makes it a single, smooth, simple wave.
  - `frequency = 0.0025`: A *tiny* frequency, which means the wave is extremely long and slow.

2. "Erosion" Map (The "Flatness" Control) This map decides *if* an area should be flat (like a plain) or hilly.

- **Settings:**
  - `octaves = 2`: Two layers of noise, making it a bit more varied.
  - `frequency = 0.01`: A medium frequency, creating rolling hills.

3. "Peaks & Valleys" Map (The Details) This adds all the small, rough, and bumpy details that make the ground look "real."

- **Settings:**
  - `octaves = 4`: Our most complex noise, with four layers stacked together.
  - `frequency = 0.025`: A high frequency, which creates fast, rough, and jagged noise.

How We Blend Them (The "Magic Formula")

We don't just add them. We use the **Erosion** map to control the other two. Here is the blend from `generateChunk`, line by line:

1. **`double flatness = (eros + 1.0) * 0.5;`** First, we take our "Erosion" noise (`eros`), which is between -1.0 and 1.0, and convert it into a `flatness` value that's between 0.0 and 1.0.

2. **`double baseHeight = cont * (1.0 - flatness);`** Next, we create our `baseHeight`. We take the "Continental" noise (`cont`) and *multiply* it by `(1.0 - flatness)`.

This is the key: if the "Erosion" map says an area is flat (high `flatness`), this formula *squashes* the continental noise, turning mountains into plains.

3. **`double terrain = baseHeight + pv * (0.5 - flatness);`** Finally, we add our "Peaks & Valleys" noise (`pv`) to the `baseHeight`. But, we also multiply `pv` by `(0.5 - flatness)`. This is another clever trick. It means if the area is a "flat plain" (high `flatness`), this term becomes negative or zero, *preventing* small, bumpy details from being added.

This blending formula is what combines the three simple maps into the final, realistic terrain you see on the screen.

Carefully watch the attached video as it contains a complete example run of the simulator.

Allowed Libraries and functions:

- **Fstream**
- **Iostream**
- **Cstdlib.h: system("clear")**
- **Ctime: time()**
- **Random: only allowed for random number generation like in starter code**
- **Unistd and termios are to be used just for non blocking input (getch immitation) Copy it over from Q8 starter code (No Further allowance of these libs)**

Additional resources for Perlin and Fractal Noise (For math nerds)

https://observablehq.com/@bensimonds/perlin-noise
https://www.cs.umd.edu/class/spring2018/cmsc425/Lects/lect12-1d-perlin.pdf
https://www.cs.umd.edu/class/spring2018/cmsc425/Lects/lect13-2d-perlin.pdf