Foreword

All the codes are tested with `g++ 11.2.0` with the following compiler flags on

```
-Wall -Wextra -Wconversion -static -Wl,--stack=268435456 -O2 -std=c++20
```

```
PS C:\Users\REDACTED\Documents> g++ --version
g++.exe (MinGW-W64 x86_64-posix-seh, built by Brecht Sanders) 11.2.0
```

Question 1 - Iota

There are 2 main ways that we can solve this problem. We can either start processing it from the lowest digit or the highest digit.

- Lowest Digit First: In this case, we will use modulo and we will have to reverse the string at the end. We will have to handle the negative sign separately as we can't append it at the start (since we are reversing it).
- Highest Digit First: In this case, we will find the highest exponent that is within the range and we wouldn't have to reverse the string.

I decided to go with the second (highest digit first) approach.

Complexity Analysis

Let N be the number of digits in $value$.

- $Time\ Complexity - O(N)$
- $Space\ Complexity - O(1)$

Code

```
#include <string>
#include <iostream>

/**
 * Convert input value into base between 2 - 16 in string format
 * The output string is allocated on behalf of the caller
 *
 * If the value is negative, '-' is added to the front of the resulting string.
 * Any alphbetic letters in the string will be in lower case
 *
 * Input:
 * value (int) - decimal value to be converted to a string
 * base  (int) - base used to represent the value as a string
 *
 * Output:
 * (char*) - A pointer to the resulting null-terminated string
 */
char* itoa(int value, int base){
    if (base <= 1 || base > 16){
        throw std::invalid_argument("base must satisfy 2 <= base <= 16 \n");
    }

    // I am assuming the following
    // - .size()       member   functions are all banned
    // - realloc()     built-in functions are all banned.
    // - int()         casting  functions are all banned.
    // - There is can no function call except for malloc() and new keyword
    // - itoa(-10, 2) is valid. That is, value can be negative.
    // - iota(10, -2) or iota(10, 1) is not valid.

    std::string letter = "0123456789abcdef"; // all the letters we need
    std::string tmp;
    int sz = 0;
    if (value < 0){ // if negative, make it positive and insert -
        tmp += '-';
        value *= -1;
        ++sz;
    }

    // Find the highest digit we care about
    int curBase = 1;
    while(1LL * curBase * base <= value){
        curBase *= base;
    }
    // Then, we start calculating from high to low digits
    while(curBase){
        int curDigit = value / curBase;
        value -= curDigit * curBase;
        tmp += letter[curDigit];
        curBase /= base;
        ++sz;
    }

    // We loop to sz inclusive, tmp[sz] will null-terminated for us
    char* ans = new char[sz+1];
    for (int i = 0; i <= sz; ++i){
        ans[i] = tmp[i];
    }
    return ans;
}
```

Question 2 - BuildStringFromMatrix

There are 3 ways that I come up with for this problem. The high-level idea is that we can walk the matrix and if our next move with the current direction is illegal (i.e. visited or out of bound), then we will change the direction clockwise. We have 3 ways to implement this. The most obvious way would be to have a 2 dimensional bool array but I quickly threw that idea out of the window as it is far from optimal. The second idea that I had is having a visited bool array for rows and columns, but we can do better than that. What about a **constant space** solution where we adjust the boundary accordingly?

Let me elaborate: At first, we have the following boundary for `[top, right, bottom, left] = [0, numColumns-1, numRows-1, 0]`, and when we go from $\rightarrow$ and turn $\downarrow$, we adjust the top boundary by adding 1 to it. We will do the same (`+1` or `-1` accordingly) for all the turns we make. That way, we will be able to achieve $O(1)$ constant space for this problem.

Assumption

I am assuming on the spec sheet

```
2, 3, 4, 8
5, 7, 9, 12
1, 0, 6, 10
```

comes from this 1-row array

```
2, 3, 4, 8, 5, 7, 9, 12, 1, 0, 6, 10
```

My solution is coded based on that assumption.

Complexity Analysis

- $Time\ Complexity - O(numRows \cdot numColumns)$
- $Space\ Complexity - O(1)$

Code

```
#include <string>
#include <array>
#include <iostream>

/**
 * Given a row-major matrix of integers, builds and returns
 * a string with the entries of that matrix appended in clockwise order
 * separated by ", " for each element.
 *
 * This runs in O(1) Space.
 *
 * Input:
 * value      (int*) - the 1D row of integers
 * numRows    (int)  - the number of rows in the matrix
 * numColumns (int)  - the number of columns in the matrix
 * outBuffer  (char*)- the output string. Guaranteed to be large enough to hold the answer.
 *
 * Output: void
 */
void BuildStringFromMatrix(int* matrix, int numRows, int numColumns, char* outBuffer){
    // edge case
    if (matrix == nullptr){
        throw std::invalid_argument("matrix can't be nullptr.\n");
    }
    if (numRows <= 0 || numColumns <= 0){
        throw std::invalid_argument("numRows and numColumns both must be positive.\n");
    }

    // initial data that we need
    int clockwiseDir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    std::array<int, 4> validRange {0, numColumns, numRows, 0}; // this is the boundary I elaborated on in the
.md doc
    std::array<int, 4> lost {1, -1, -1, 1}; // this is the adjustment for boundary in each direction.
    int curDir = 0, row = 0, col = 0, total = numColumns * numRows;

    // helper function to check if (row, col) is within bound.
    auto withinBound = [&validRange](int row, int col) -> bool{
        return row >= validRange[0] && row < validRange[2]
            && col >= validRange[3] && col < validRange[1];
    };

    // compute the answer
    while(total--){
        int value = matrix[row * numColumns + col];
        int size = snprintf(nullptr, 0, "%d, ", value);
        int nextRow = row + clockwiseDir[curDir][0];
        int nextCol = col + clockwiseDir[curDir][1];
        sprintf(outBuffer, "%d%s", value, total? ", " : "");
        if (!withinBound(nextRow, nextCol)){
            validRange[curDir] += lost[curDir];
            curDir = (curDir + 1) % 4;
        }
        row += clockwiseDir[curDir][0];
        col += clockwiseDir[curDir][1];
        outBuffer += size;
    }
}
```

Thank you for reading my submission! I hope you find my explanation helpful.