COMP2611 Spring 2023 Homework #3

(Due 11:55pm, Monday April 17, 2023)

Notes:

- This is an individual assignment; all works must be your own. You can discuss with your friends but never show your code to others.
- Write your code in given MIPS assembly skeleton files. Add your own code under TODOs in the skeleton code. Keep other parts of the skeleton code unchanged.
- Make procedure calls with the registers as specified in the skeleton, otherwise the provided procedures may not work properly. Preserve registers according to the MIPS register convention on slide 76 of the ISA note set.
- Zip the three finished MIPS assembly files into a single zip file, *<your_stu_id>.zip* file (without the brackets). Do not change names of the given skeleton files.
- To submit, first find the Canvas page of COMP2611, homework3, and then upload the "*<your_stu_id>.zip*" file. You can upload for as many times as you like, only the latest one before the deadline will be marked.
- **Solutions of this homework will be posted at the course web right after the deadline, so no late submission will be accepted.**
- **Your submitted program must be able to run under MARS, otherwise it will not be marked**.

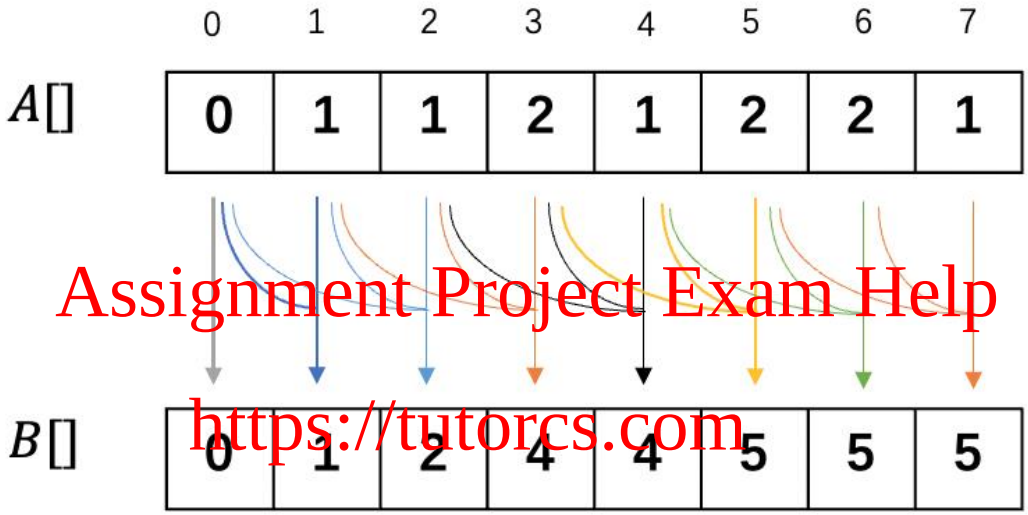
Question 1: 3-cumulative Sum Array (20 marks)

We define an array `B[]` to be the 3-cumulative sum array of array `A[]` if the array `B[]` has the same length as `A[]`, and the elements are calculated according to the following equation:

$$B[i] = \begin{cases} A[0], & i = 0 \\ A[0] + A[1], & i = 1 \\ A[i-2] + A[i-1] + A[i], & i \geq 2 \end{cases}$$

.

The picture below illustrates the idea:



Refer to the following C++ program for the details in calculating and outputting the 3-cumulative sum array `B[]`, for a user-specified integer array `A[]` with **10 elements**.

Your task is to implement `CumulativeSum` MIPS procedure in the skeleton file. The procedure should implement similar functionality as the `CumulativeSum()` function in the C++ program.

Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code, NO new procedure is allowed. But you can add new labels in the `CumulativeSum` procedure as you wish.

C++ program (copy and paste to run it)

```
#include <iostream>
using namespace std;

// Function that calculates the 3-
cumulative Sum of the A[] array and stores the results in the B[] array
void CumulativeSum(int A[], int B[], int B_SIZE){
    int tmp_sum = 0; // Temporary variable to store the sum
    int a_idx = 0; // Index in the A[] array

    // Iterate through the B[] array
    for (int b_idx = 0; b_idx < B_SIZE; b_idx++){
        // Special case when b_idx is 0
        if (b_idx==0){
            tmp_sum = A[0];
        }
        // Special case when b_idx is 1
        if (b_idx==1){
            tmp_sum += A[0];
            tmp_sum += A[1];
        }
        // General case when b_idx is greater than or equal to 2
        if (b_idx>=2){
            a_idx = b_idx-
2; // Calculate the corresponding start index in A[] array
            tmp_sum += A[a_idx];
            tmp_sum += A[a_idx+1];
            tmp_sum += A[a_idx+2];
        }
        // Assign the 3-cumulative Sum result to the B[] array
        B[b_idx] = tmp_sum;
        tmp_sum = 0; // Reset the temporary sum variable for the next i
teration
    }
}

int main(){
    const int A_SIZE = 10;
    const int B_SIZE = 10;
    int A[A_SIZE]; // Declare the A[] array
    int B[B_SIZE] = {0}; // Declare and initialize the B[] array with z
eros

    // Prompt the user to enter integers for the A[] array
```

```
    cout << "Please enter integers in array A[] one by one, use [Enter]
 to split:"<<endl;
    for(int i = 0; i < A_SIZE; i++){
        cout << "A["<<i<<"]: ";
        cin >> A[i];
    }

    // Call the Cumulativesum() function to calculate the 3-
Cumulative Sum
    CumulativeSum(A, B, B_SIZE);

    // Print the results of the 3-Cumulative Sum
    cout << "Here is the 3-Cumulative Sum result:" << endl;
    for(int i = 0; i < B_SIZE; i++){
        cout << B[i] << ' ';
    }
    cout << "\n";

    return 0;
}
```


A sample execution of the program in MARS

```
Please enter integers in array A[] one by one, use
[Enter] to split:
A[0]: 0
A[1]: 1
A[2]: 1
A[3]: 2
A[4]: 1
A[5]: 2
A[6]: 2
A[7]: 1
A[8]: 0
A[9]: 0
Here is the 3-LocalSum result:
0 1 2 4 4 5 5 5 3 1
-- program is finished running --
```

Question 2: Two-dimensional Convolution (20 marks)

The CNNs (Convolutional Neural Networks) have revolutionized the field of computer vision by achieving great accuracies in image classification, object detection, and segmentation. One basic operator of a convolutional neural network (CNN) is the convolution operation. We will implement a simplified version of convolution operation in this question.

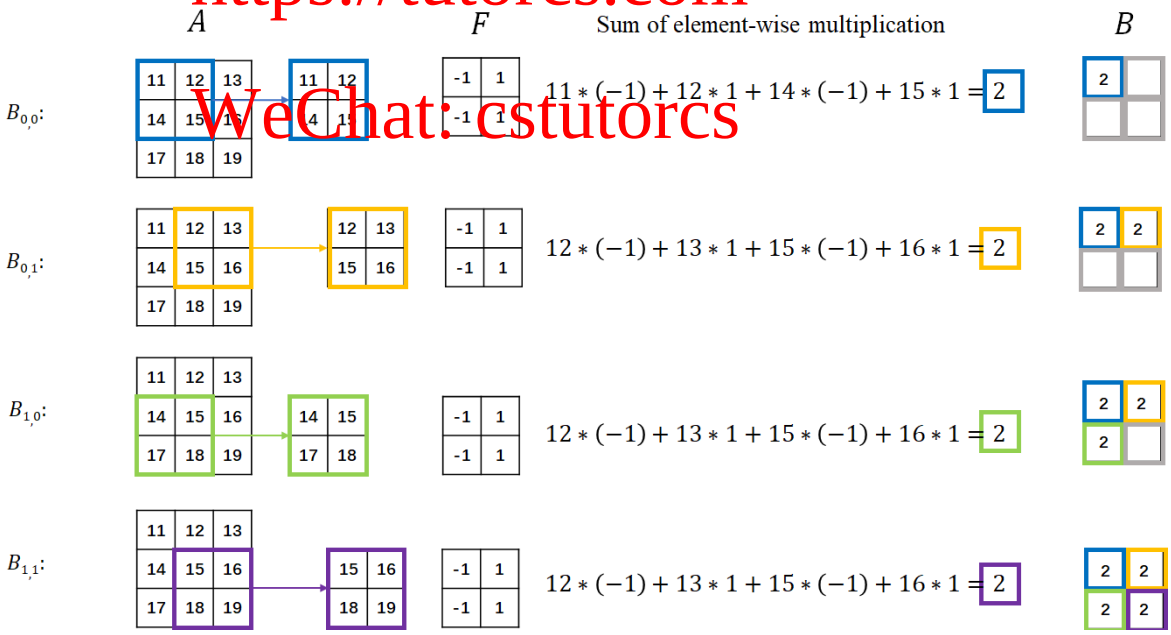
Consider a matrix **A**, and a 2×2 matrix **F** (this matrix is also known as the "Filter" matrix or the "Kernel" matrix). If we denote by $a_{i,j}$, $b_{i,j}$, and $f_{i,j}$ the elements in the i^{th} row and j^{th} column of a matrix **A**, **B** and **F** respectively. We define the following equation for the convolution operation on the input matrix **A** to produce the output matrix **B** (note that the equation assumes **F** to be a 2×2 matrix):

$$b_{i,j} = a_{i,j} \times f_{0,0} + a_{i,j+1} \times f_{0,1} + a_{i+1,j} \times f_{1,0} + a_{i+1,j+1} \times f_{1,1}$$

As an illustration example, using the above equation, $b_{0,0}$ is:

$$b_{0,0} = a_{0,0} \times f_{0,0} + a_{0,1} \times f_{0,1} + a_{1,0} \times f_{1,0} + a_{1,1} \times f_{1,1}$$

The following is the picture view to illustrate the operations in producing the output Matrix **B**:



For simplicity, we assume that the values of **elements in filter F can only be ± 1 or 0**, so that we can implement the convolution operations using additions and subtractions. You may also assume that **the size of F is always 2×2** (i.e. 2 rows by 2 columns), and **A** is always a square matrix (with the same number of rows and columns). Moreover, you may assume that

the matrices **A**, **B**, **F** are stored in the 1-D array in row order in MIPS. For example, the following matrix:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

is stored as 1-D array in MIPS as:

$$A[\,] = \{a_{0,0}, a_{0,1}, \ldots, a_{0,n-1}, a_{1,0}, a_{1,1}, \ldots, a_{1,n-1}, \ldots, a_{n-1,0}, a_{n-1,1}\ , \ldots, a_{n-1,n-1}\}$$

By referring to the C++ program given below, implement `TwoDimensionConv` MIPS procedure in the skeleton file. The procedure should implement the same functionality as the `TwoDimensionConv()` function in the C++ program.

Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code, NO new procedure is allowed. But you can add new labels in the `TwoDimensionConv` procedure as you wish.

C++ program (copy and paste to run it)

```
#include <iostream>
using namespace std;

//The multiply() function is a custom implementation of integer multipl
ication using repeated addition.
int multiply(int a, int b) {
    int result = 0;
    for (int i = 0; i < b; i++) {
        result = result + a;
    }
    return result;
}

//This function iterates through each element in the filter (Matrix F),
 multiplies the filter elements with the corresponding elements in A,
//and then accumulate it to calculate B_ij.
//To make the programming task simpler, we assume elements of Matrix F
can only be -1, 1, or 0, and the size of F is always 2 x 2.
//The result of this function is the value for one element in B (i.e. B
_ij).
int ConvForElement(int A[], int F[], int B_i, int B_j, int A_size, int
F_size){
    int local_sum = 0; //variable to accumulate the B_ij calculated at
each of the steps
    int F_idx = 0; //an index for filter matrix F
    int A_idx = 0; //an index for input matrix A

    for (int F_i = 0; F_i < F_size; F_i++) { //we assume F to be a 2 x
2 matrix, F_i is the row number of the matrix, therefore F_i=0, 1
        for (int F_j = 0; F_j < F_size; F_j++) { //we assume F to be a
2 x 2 matrix, so F_j=0, 1

            // scan matrix F in order of F[0][0], F[0][1], F[1][0], F[1
][1] (i.e row 0 first, then row 1)
            F_idx = multiply(F_size, F_i) + F_j; //calculate index for
the current F element

            // calculate the corresponding element in matrix A to multi
ply with the current matrix F element
            A_idx = multiply(A_size, B_i + F_i) + B_j + F_j; //calculat
e index for corresponding element in input matrix A
```

```
            if (F[F_idx] > 0) { //we assume the matrix F is holding +1,
 -1, or 0, here the element of F in F[F_idx] is +1
                local_sum = local_sum + A[A_idx];
            }

            if (F[F_idx] < 0) {  //we assume the matrix F is holding +1
, -1, or 0, here the element of F in F[F_idx] is -
1, we need doing nothing when F[F_idx]==0
                local_sum = local_sum - A[A_idx];
            }
        }
    }
    //return the completely calculated B_ij
    return local_sum;
}

//The function TwoDimensionConv() calculate the output matrix B accordi
ngly using matrices A and F
//It calculates B_ij by calling ConvForElement() for each iteration of
the nested for loop (B_i loop and B_j loop)
//The function does not have return value, but it calculates every elem
ent of the output matrix B (using ConvForElement()) and store the resul
t to B[]
void TwoDimensionConv(int A[], int F[], int B[], int A_size, int F_size
, int B_size) {
    int local_sum = 0; //variable to hold the B_ij returned by calling
ConvForElement()
    int B_idx = 0; //an index for output matrix B
    // the nested loop below calculates B in the order B[0][0], B[0][1]
,..., B[1][0], B[1][1],..., B[2][0],B[2][1],...
    for (int B_i = 0; B_i < A_size - F_size + 1; B_i++) { //calculating
 the elements in the i-th row of B (where i is in the variable B_i)
        for (int B_j = 0; B_j < A_size - F_size + 1; B_j++) { // calcul
ating element B_ij of B (where j is in the variable B_j)
            // calculate index for B_ij
            B_idx = multiply(B_size, B_i) + B_j;

            local_sum = 0;

            // calculate the B_ij value
            local_sum = ConvForElement(A, F, B_i, B_j, A_size, F_size);

            // store the B_ij to matrix B
            B[B_idx] = local_sum;
```

```
        }
    }
}

//The main() function takes user input for A, F, and their sizes.
//It then calculates the size of the output matrix B based on the sizes
 of A and F.
//Finally, it calls TwoDimensionConv() and prints the resulting matrix
B.
int main() {
    int A[100]; //input matrix, allocate more than enough
    int A_size; //size of input matrix
    int F[50];  //filter matrix, allocate more than enough
    int F_size; //size of filter matrix
    int B[100]; //output matrix, allocate more than enough
    int B_size; //size of output matrix
    int idx;    //to hold index of current element being input

    // User input size of A
    cout << "Please enter the size of your matrix:" << endl;
    cin >> A_size;

    // User input elements of A
    cout << "Please enter integers in array A[] one by one:" << endl;
    for (int i = 0; i < A_size; i++) {
        for (int j = 0; j < A_size; j++) {
            cout << "A[" << i << "]";
            cout << "[" << j << "]: ";
            idx = multiply(A_size, i);
            idx = idx + j;
            cin >> A[idx];
        }
    }

    // User input size of F
    cout << "Please enter the size of your filter:" << endl;
    cin >> F_size;

    // User input elements of F
    cout << "Please enter integers in array F[] one by one:" << endl;
    for (int i = 0; i < F_size; i++) {
        for (int j = 0; j < F_size; j++) {
            cout << "F[" << i << "]";
            cout << "[" << j << "]: ";
```

```
            idx = multiply(F_size, i);
            idx = idx + j;
            cin >> F[idx];
        }
    }

    B_size = A_size - F_size + 1; //calculate size of output matrix

    TwoDimensionConv(A, F, B, A_size, F_size, B_size); //apply convolut
ion to input matrix A with filter F, store result in output matrix B

    // Print B
    cout << "After convolution:" << endl;
    for (int i = 0; i < B_size; i++) {
        for (int j = 0; j < B_size; j++) {
            cout << "B[" << i << "]";
            cout << "[" << j << "]: ";
            idx = multiply(B_size, i);
            idx = idx + j;
            cout << B[idx] << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

A sample execution of the program in MARS

```
Please enter the size of your matrix A:
3
Please enter integers in array A[] one by one:
A[0][0]: 11
A[0][1]: 12
A[0][2]: 13
A[1][0]: 14
A[1][1]: 15
A[1][2]: 16
A[2][0]: 17
A[2][1]: 18
A[2][2]: 19
Please enter the size of your matrix F:
2
Please enter integers in array F[] one by one:
F[0][0]: -1
F[0][1]: 1
F[1][0]: -1
F[1][1]: 1
After convolution:
B[0][0]: 2  B[0][1]: 2
B[1][0]: 2  B[1][1]: 2

-- program is finished running --
```

Question 3: The game of NIM (20 marks)

Nim is a simple two-player game. There are many variations of this game, see https://en.wikipedia.org/wiki/Nim for some of the examples. In our implementation of Nim, the game board consists of three rows of rocks. At the start of our version of the game, Row A has 3 rocks, Row B has 5 rocks, and Row C has 8 rocks.

The following illustrates what is being shown on the screen at the beginning of the game. At the left of each row the program outputs the row name (for instance "`ROW A:`"). Then an ASCII character (lowercase "o", ASCII code x006F) is shown to represent a single rock in the row. The initial state of the game board (with 3 rocks in Row A, 5 rocks in Row B and 8 rocks in row C) is:

```
ROW A: ooo
ROW B: ooooo
ROW C: oooooooo
```

The rules of the game are as follows:

1. Each player takes turn to remove **one or more rocks from a single row**.
2. A player **cannot remove more rocks that that is remaining** in the chosen row.
2. The game ends when a player removes the last rock from the game board. The player who **removes the last rock loses**.

For simplicity, we assume in the assembly program that each row can have a maximum of 9 rocks only (the C++ program does not have this limitation). To specify the row and the number of rocks to remove, the player inputs a letter and followed immediately by a digit from "1" to "9". The letter (A, B, or C) specifies the row, and the digit specifies how many rocks to be removed. The player does not need to press "Enter"/"Return" in the assembly program after entering the inputs (but he/she needs doing so in the C++ program). The program will make sure the player's move consists of a valid row and valid number of rocks. If the player's move is invalid, the program will output an error message and prompt the same player for input again.

For example, if Player 1 wants to remove invalid number of rocks from the game (assume we have 3 rocks in Row A, 5 rocks in Row B and 8 rocks in row C):

```
Player 1, choose a row and number of rocks: D4
Invalid move. Try again.
Player 1, choose a row and number of rocks: A9
Invalid move. Try again.
Player 1, choose a row and number of rocks: A*
Invalid move. Try again.
Player 1, choose a row and number of rocks: &4
Invalid move. Try again.
Player 1, choose a row and number of rocks:
```

After a player has inputted a valid move, the program should check if the last remaining rock has been removed from the board. If this is the case, the game is over and the player removes the last rock loses. The program should display the winner and stops the game. If there is/are still rock(s) on the board, the program should update the state of the game board to reflect the move, re-display the updated game board, and continue with the next player's turn.

Your task is to implement `validate_and_move` MIPS procedure in skeleton file. The procedure should implement similar functionality as the `validate_and_move()` function in the C++ program.

Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code, NO new procedure is allowed. But you can add new labels in the `validate_and_move` procedure as you wish.

C++ program (copy and paste to run it)

(Note: In this C++ program, the player needs to press Enter/Return before the program gets the input. In the assembly program, the player just needs to input a character and a digit then the program will get the input immediately.)

```
#include <iostream>
using namespace std;
string player1 = "Player 1, choose a row and number of rocks: ";
string player2 = "Player 2, choose a row and number of rocks: ";
string tryagain="Invalid move. Try again.\n";
string win1="\nPlayer 1 Wins.";
string win2="\nPlayer 2 Wins.";
string p_rowa= "\nROW A: ";
string p_rowb= "\nROW B: ";
string p_rowc= "\nROW C: ";
string rock= "o";
int ROWA= 0x0003,ROWB= 0x0005,ROWC= 0x0008;
char NUM = 0, ROW = 0, A_ascii=0x41,B_ascii=0x42,C_ascii=0x43;

// Function to print the rocks in a row
void print_rocks(int num){
    for (int i=0;i!=num;i++){
        cout << rock;}
}

// Function to print the current state of the game
void print_state(){
    cout << p_rowa;
    print_rocks(ROWA); // Calls the print_rocks function to print the numbe
r of rocks in row A
    cout << p_rowb;
    print_rocks(ROWB); // Calls the print_rocks function to print the numbe
r of rocks in row B
    cout << p_rowc;
    print_rocks(ROWC); // Calls the print_rocks function to print the numbe
r of rocks in row C
    cout << endl;
}

// Function to validate player and number to move, and remove rocks if vali
d
int validate_and_move(){
    if (NUM<1){ // If the number of rocks to be moved is less than 1
        return 0; // Return 0 to indicate an invalid input
    }
```

```
    if (NUM>9){ // If the number of rocks to be moved is more than 9
        return 0; // Return 0 to indicate an invalid input
    }
    if (ROW==A_ascii){ // If the selected row is row A
        if(NUM>ROWA){ // If the number of rocks to be moved is more than th
e available rocks in row A
            return 0; // Return 0 to indicate an invalid input
        }
        ROWA = ROWA - NUM; // If input is valid, subtract the number of roc
ks from the available rocks in row A
        return 1; // Return 1 to indicate a valid input
    }
    if (ROW==B_ascii){ // If the selected row is row B
        if(NUM>ROWB){ // If the number of rocks to be moved is more than th
e available rocks in row B
            return 0; // Return 0 to indicate an invalid input
        }
        ROWB = ROWB - NUM; // If input is valid, subtract the number of roc
ks from the available rocks in row B
        return 1; // Return 1 to indicate a valid input
    }
    if (ROW==C_ascii){ // If the selected row is row C
        if(NUM>ROWC){ // If the number of rocks to be moved is more than th
e available rocks in row C
            return 0; // Return 0 to indicate an invalid input
        }
        ROWC = ROWC - NUM; // If input is valid, subtract the number of roc
ks from the available rocks in row C
        return 1; // Return 1 to indicate a valid input
    }
    return 0; // Return 0 to indicate an invalid input
}

// Function to check if the game is over
int game_over(){
    if(ROWA!=0 || ROWB!=0 || ROWC!=0){ // If there are still rocks in row A
, B or C
        return 0; // Return 0 to indicate the game is not over
    }
    return 1; // If all the rows are empty, return 1 to indicate the game i
s over
}

// Main function to run the game
```

```
int main(){
    int player_flag = 0; // Initialize a flag to keep track of which player
 is currently playing
    string player_line; // Initialize a string to hold the player prompt
    string win_line; // Initialize a string to hold the winning message
    char c; // Initialize a char variable to hold the player input
    int valid_flag = 0; // Initialize a flag to indicate if the player inpu
t is valid
    int game_over_flag = 0; // Initialize a flag to indicate if the game is
 over

    // Run the game loop
    while (1)
    {   print_state(); // Print the current state of the game
        // Loop for player input and validation
        while (1)
        {   if (player_flag==0) {player_line = player1;} // Initialize the
player prompt
            else {player_line = player2;}
            cout << player_line; // Print the player prompt
            cin >> ROW; // Get the player row selection
            cin >> NUM; // Get the player number of rocks selection
            NUM = NUM-
48; // Convert ascii value to integer value
            valid_flag = validate_and_move(); // Call the validate_and_move
 function to validate the player input and move the rocks

            // If the input is valid, switch the player
            // the implementation in the assembly code is slightly differen
t, but it does the same task
            if (valid_flag>0){
                player_flag = (player_flag + 1)%2;
                break;}
            // If the input is invalid, prompt the player to try again
            else{
                cout << tryagain;}
        }
        game_over_flag = game_over(); // Call the game_over function to che
ck if the game is over
        // If the game is over, print the winner message and end the game l
oop
        if (game_over_flag>0){
```

```
            if (player_flag == 0) {win_line = win1;} else {win_line = win2;
}
            cout << win_line;
            break;           }
    }
    return 0; // End the program
}
```

A sample execution of the program in MARS

```
ROW A: ooo
ROW B: ooooo
ROW C: oooooooo
Player 1, choose a row and number of rocks: A3

ROW A:
ROW B: ooooo
ROW C: oooooooo
Player 2, choose a row and number of rocks: B4

ROW A:
ROW B: o
ROW C: oooooooo
Player 1, choose a row and number of rocks: B2
Invalid move. Try again.
Player 1, choose a row and number of rocks: B1

ROW A:
ROW B:
ROW C: oooooooo
Player 2, choose a row and number of rocks: C5

ROW A:
ROW B:
ROW C: ooo
Player 1, choose a row and number of rocks: C2

ROW A:
ROW B:
ROW C: o
Player 2, choose a row and number of rocks: C1

Player 1 Wins.
-- program is finished running --
```