```
; FindMax(Square, N, M)
; Input Parameters
;   $a0: Square is the address of first element of 2D matrix
;   $a1: N is the number of rows in Square
;   $a2: M is the number of columns in Square
; Return Value:
;   $v0: value of maximum element in Square
;
0x1FFF FFB0 FindMax:    li      $v0, -1             # max <-- -1
0x1FFF FFB4             move    $t0, $zero          # i <-- 0
0x1FFF FFB8 NextRow:    slt     $t7, $a1, $t0       # if N<i then $t7 <-- 1
0x1FFF FFBC             bne     $t7, $zero, Return  # if i>=N Return
0x1FFF FFC0             move    $t5, $a0            # p <-- Square
0x1FFF FFC4             move    $t1, $zero          # j <-- 0
0x1FFF FFC8 NextColumn: slt     $t7, $a2, $t1       # if M<j then $t7 <-- 1
0x1FFF FFCC             bne     $t7, $zero, RowDone # if j>=M RowDone
0x1FFF FFD0             mul     $t3, $t0, $a1       # $t3 <-- i*N
0x1FFF FFD4             add     $t4, $t3, $t1       # $t4 <-- i*N+j
0x1FFF FFD8             sll     $t5, $t4, 2         # $t5 <-- 4*(i*N+j)
0x1FFF FFDC             lw      $t6, 0($t5)         # $t6 <-- Square[i][j]
0x1FFF FFE0             slt     $t7, $v0, $t6       # if(max < Square[i][j]) then $t7 <-- 1
0x1FFF FFE4             beq     $t7, $zero NoChange
0x1FFF FFE8             move    $v0, $t6            # max <-- Square[i][j]
0x1FFF FFEC NoChange:   addi    $t1, $t1, 1         # j <-- j+1
0x1FFF FFF0             j NextColumn
0x1FFF FFF4 RowDone:    addi    $t0, $t0, 1         # i <-- i+1
0x1FFF FFF8             j NextRow                   # if i != N goto NextRow
0x1FFF FFFC Return:     jr      $ra
```

Figure 1: MIPS Assembly code for `FindMax` procedure.

Question 1 (10 points): The code for the `FindMax` procedure shown in Figure 1 is not optimized and there are several improvements that could be made to improve this code.

a. (**6 points**) What changes would you do to the code to improve its performance. You do not need to write assembly code — although you may if you wish. You can simply explain how you would transform the code to make it more efficient.

The optimization shown in Figure 2 moves loop-independent code, such as the `i*N` multiplication outside of the loops and transforms the loop to eliminate the execution of jump instructions inside the loop.

The optimization shown in Figure 3 recognizes that the rows of a two-dimensional array are stored in memory sequentially. Thus, the entire computation can be executed as a single loop that scans all the elements of the two-dimensional array.

b. (**4 points**) Given the same invocation of `FindMax` with `N = 10000` and `M = 5000` that we studied in the previous question, would your changes increase or decrease the CPI? Explain your reasoning.

```
; FindMax Optimization 1
; FindMax(Square, N, M)
;
; Optimizations:
;    (1) Test parameters N and M before entering loops
;    (2) eliminate jump at the end of each loop
;    (3) move the multiplication i*N outside of inner loop
;
            li      $v0, -1             # max <-- -1
            slt     $t7, $zero, $a1     # if 0<N then $t7 <-- 1
            beq     $t7, $zero, Return  # if N<=0 Return
            slt     $t7, $zero, $a2     # if 0<M then $t7 <-- 1
            beq     $t7, $zero, Return  # if M<=0 Return
            move    $t0, $zero          # i <-- 0
NextRow:    mul     $t3, $t0, $a1       # $t3 <-- i*N
            move    $t1, $zero          # j <-- 0
NextColumn: add     $t4, $t3, $t1       # $t4 <-- i*N+j
            sll     $t5, $t4, 2         # $t5 <-- 4*(i*N+j)
            lw      $t6, 0($t5)         # $t6 <-- Square[i][j]
            slt     $t7, $v0, $t6       # if(max < Square[i][j]) then $t7 <-- 1
            beq     $t7, $zero, NoChange
            move    $v0, $t6            # max <-- Square[i][j]
NoChange:   addi    $t1, $t1, 1         # j <-- j+1
            bne     $t1, $a1, NextColumn # if j!=M NextColumn
            addi    $t0, $t0, 1         # i <-- i+1
            bne     $t0, $a2, NextRow   # if i != N goto NextRow
Return:     jr      $ra
```

Figure 2: Optimization for FindMax, move loop-independent code outside of loop and use pointers instead of indices.

In both cases the CPI should increase because we reduce the number of instructions with low cycle count that are executed in the loop but we still have to execute the `lw` in each iteration of the loop and this slower instruction dominates the CPI. For instance, for the code in Figure 3, the following holds:

$$\begin{aligned}
\text{Number of Instructions} &= 10 + 4.5 \times N \times M \\
&= 10 + 5 \times 10000 \times 5000 \\
&= 225,000,010 \\
\text{Number of Cycles} &= 6 + 2 + 10 + (10 + 2.5 + 4) \times N \times M \\
&= 18 + 16.5 \times 10000 \times 5000 \\
&= 825,000,018 \\
\text{CPI} &= 3.7 \frac{\text{Clocks}}{\text{Instructions}}
\end{aligned}$$

```
; FindMax Optimization 2
; FindMax(Square, N, M)
;
; Optimizations:
;    (1) Recognize that 2D matrix is a single array in memory
;    (2) Use pointer instead of indices
;
            li      $v0, -1              # max <-- -1
            slt     $t7, $zero, $a1      # if 0<N then $t7 <-- 1
            beq     $t7, $zero, Return   # if N<=0 Return
            slt     $t7, $zero, $a2      # if 0<M then $t7 <-- 1
            beq     $t7, $zero, Return   # if M<=0 Return
            mul     $t0, $a1, $a2        # $t0 <-- N*M
            sll     $t1, $t0, 2          # $t1 <-- 4*N*M
            add     $t2, $a0, $t1        # $t2 <-- &(Square[N][M])
            addi    $t3, $t2, 4          # $t3 <-- &(Square[N][M+1])
NextElem:   lw      $t3, 0($a0)          # $t3 <-- *Square
            slt     $t7, $v0, $t6        # if max < *Square then $t7 <-- 1
            beq     $t7, $zero, NoChange # if max >= *Square NoChange
            move    $v0, $t6             # max <-- Square[i][j]
NoChange:   addi    $a0, $a0, 4          # Square <-- Square+4
            bne     $a0, $t3, NextElem
Return:     jr      $ra
```

Figure 3: Optimization 2 for `FindMax`, recognize that two-dimensional array is stored as a single-dimensional array in memory.