Matrices and 2-dimensional Arrays

$$M = \begin{bmatrix} 71 & 62 & 33 & 89 & 85 & 74 \\ 68 & 65 & 75 & 84 & 70 & 72 \\ 87 & 0 & 1 & 90 & 92 & 88 \\ 58 & 72 & 66 & 57 & 76 & 73 \end{bmatrix}$$

- A matrix is represented in algorithms by a 2-dimensional array, i.e., an array of arrays.

- The matrix M is an array of 4 arrays, each with 6 members. If M is regarded as a 2-dimensional array, then

  M[1][2] is 75

  M[2][5] is 88

  M[4][1] is a run-time error

  M[3] is { 58, 72, 66, 57, 76, 73 }

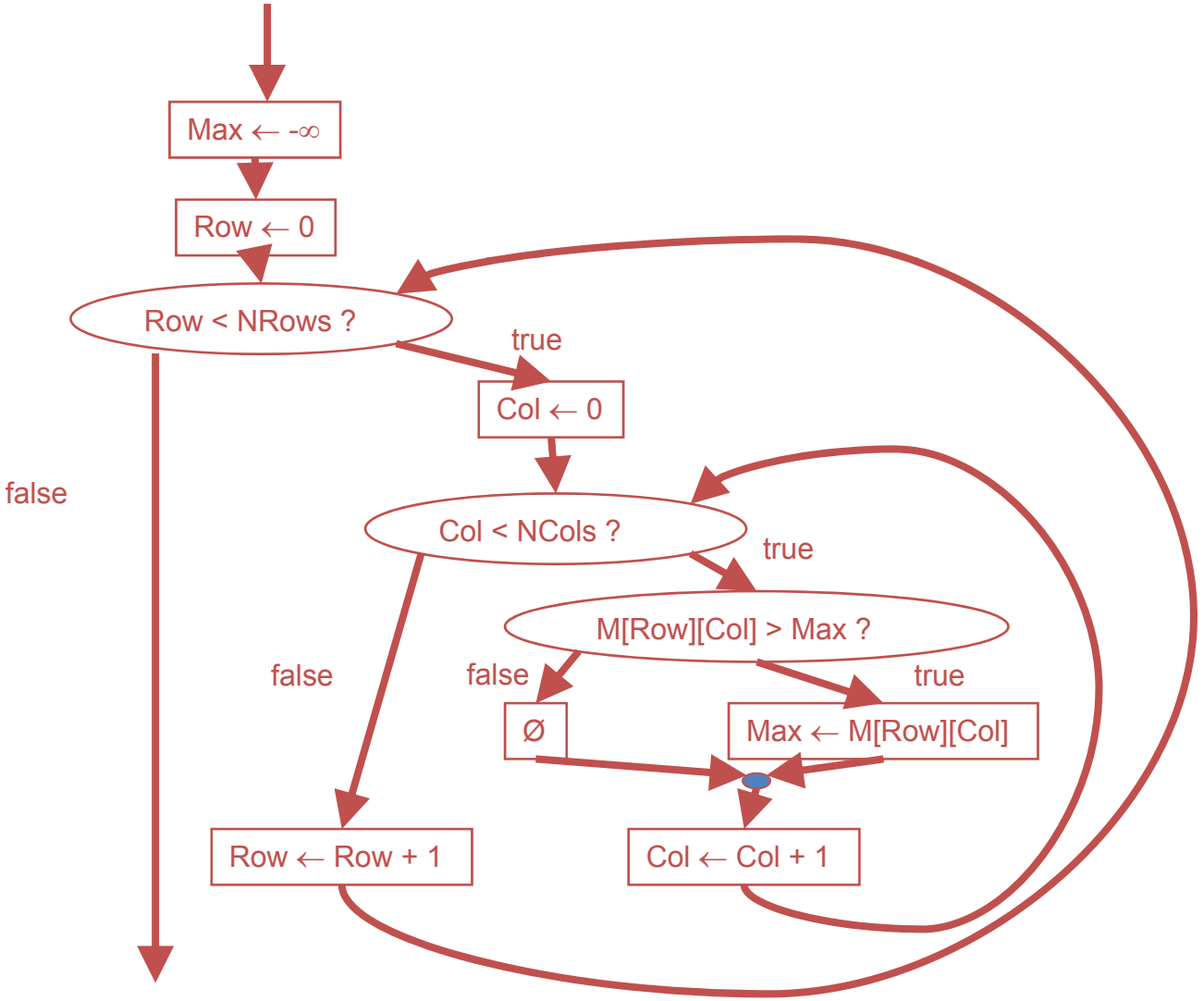
Max value in a matrix (p. 1)

- Write an algorithm to find the maximum value in a matrix.

GIVENS:	M	(a matrix)
	NRows	(the number of rows in A)
	NCols	(the number of columns in A)
INTERMEDIATES:	Row	(index of current row)
	Col	(index of current column)
RESULT:	Max	(the maximum value in the matrix)
HEADER:	Max ← FindMatrixMax(M, NRows, NCols)	

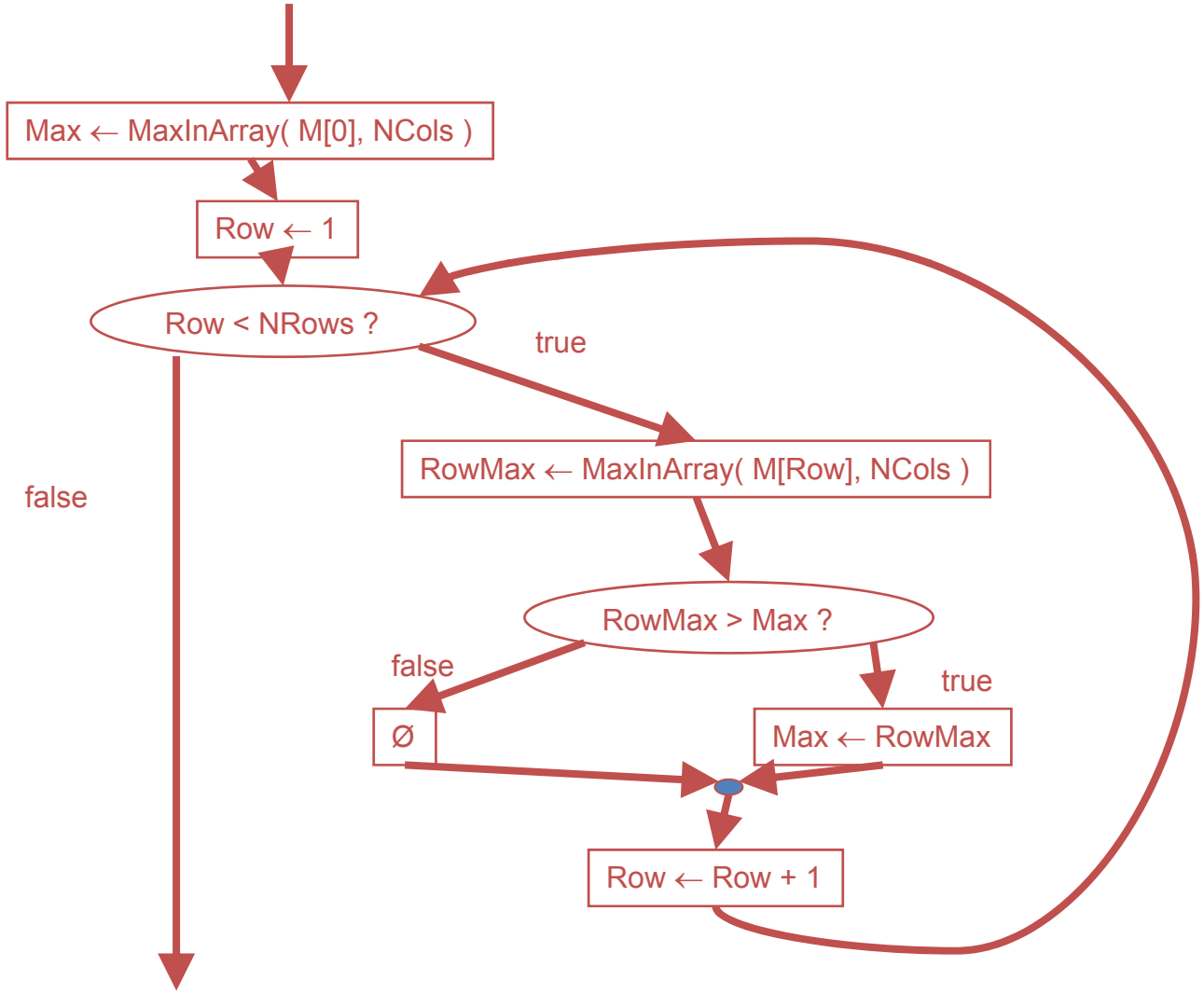
Max value in a matrix (p. 2)

BODY:

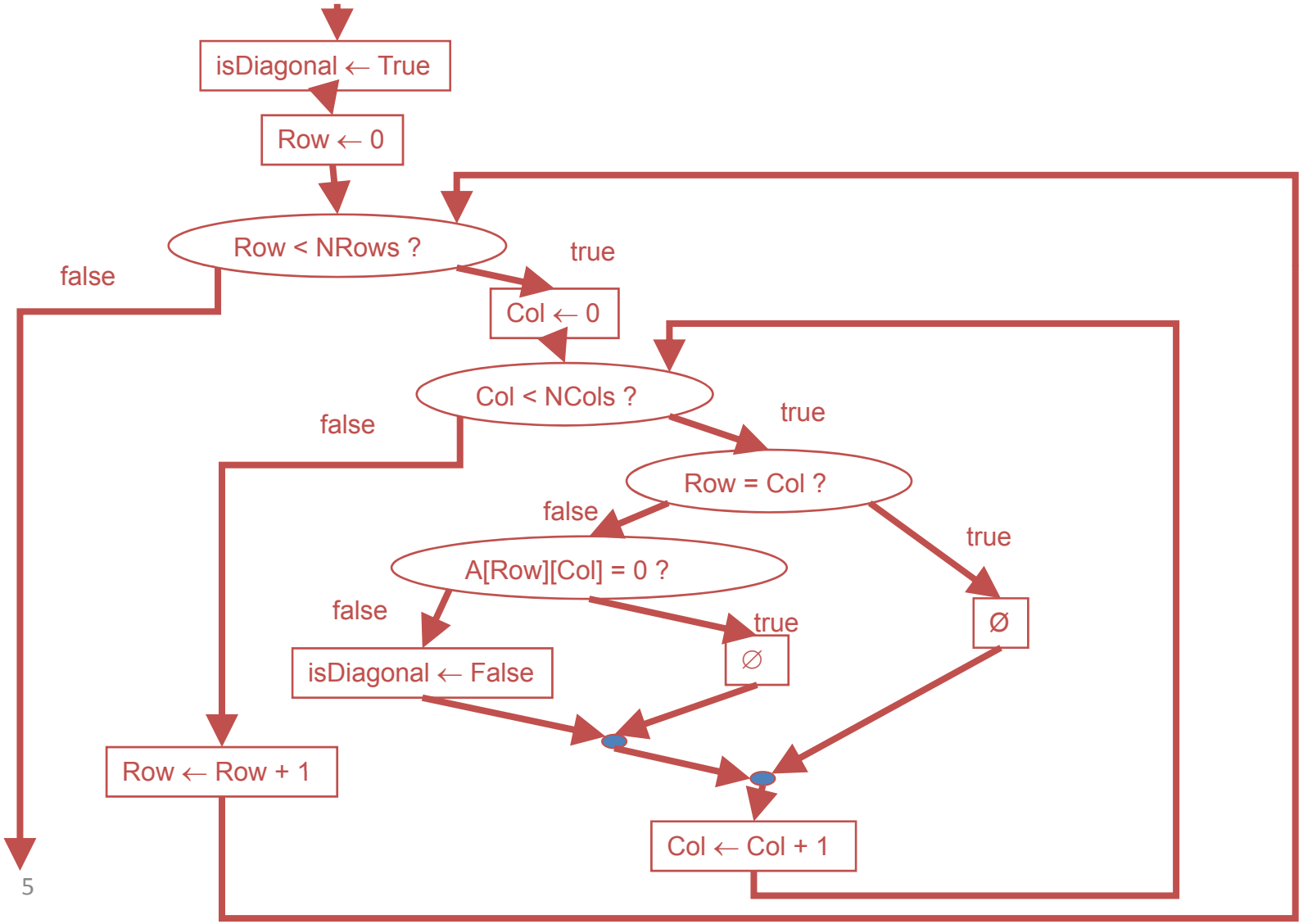


Alternative Algorithm

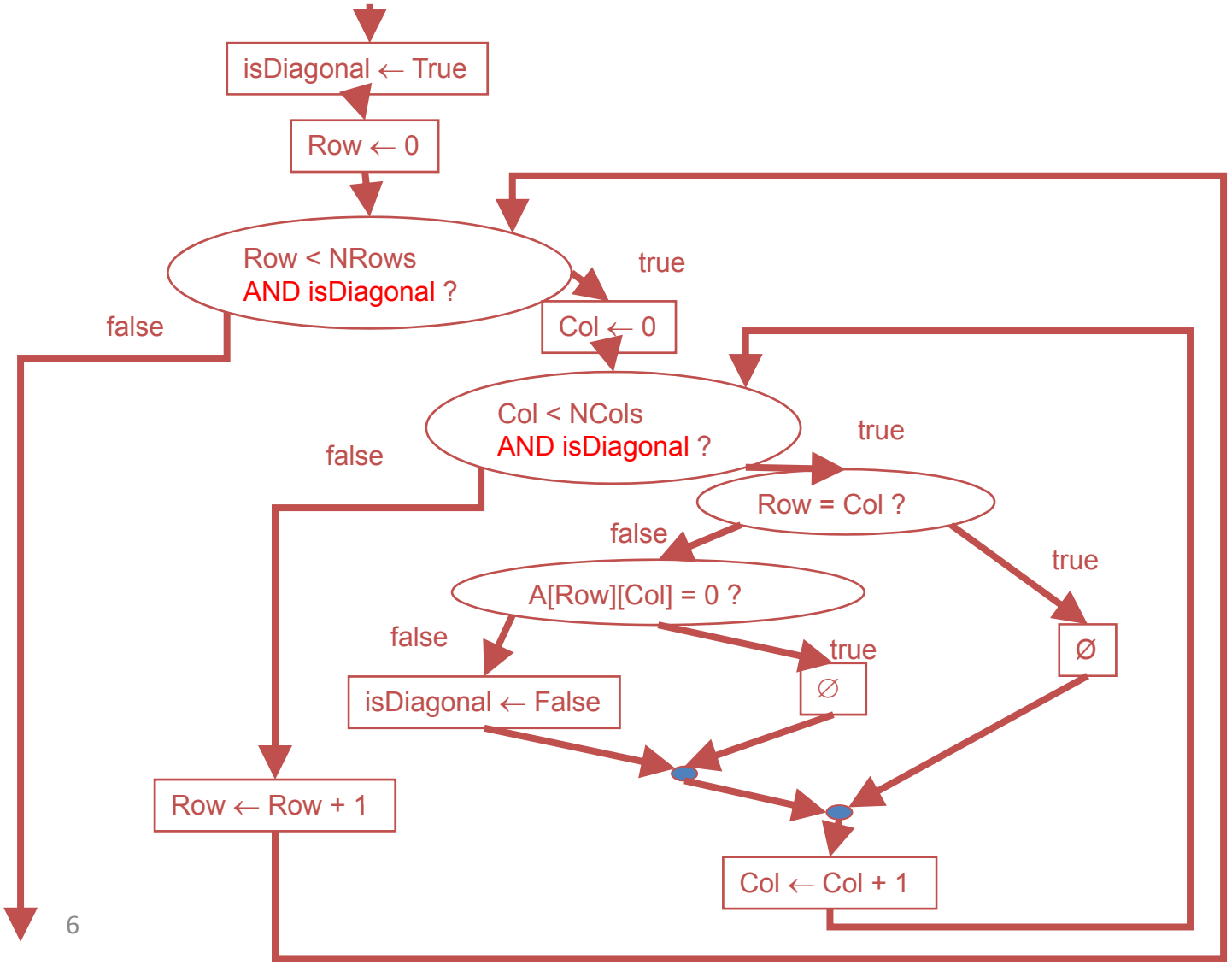
BODY:



Diagonal-check algorithm



Efficient Version



Max value in a matrix

- Translate the algorithm for the maximum value in a matrix to Java:
  - Note: **`Integer.MIN_VALUE`** is the most negative allowable integer for a Java **`int`**, and can be used for –∞.

```
public static int matrixMax (int[][] m, int nRows, int nCols)
{
  int max;
  int row;
  int col;
  max = Integer.MIN_VALUE;        // m[0][0] is an alternate choice
  for ( row = 0; row < nRows; row = row + 1 )
  {
    for ( col = 0; col < nCols; col = col + 1 )
    {
      if ( m[row][col] > max )

          max = m[row][col];
    }
  }
  return max;
}
```

Reading a Matrix

- Write Java code to read in a matrix row by row (first it reads in the number of rows and columns, then it asks for the values in row 0, then it asks for the values in row 1, etc.). All values are read one per line using **`ITI1120.readInt().`**

```
int row, col;
int nRows, nCols;
int[][] m ;
System.out.println("Enter number of rows and number of columns: ") ;
nRows = ITI1120.readInt() ;
nCols = ITI1120.readInt() ;
m = new int[nRows][nCols] ;
for ( row = 0 ; row < nRows ; row = row+1 )
{
  System.out.println("Enter the values for row " + row );
  for ( col=0; col < nCols ; col = col+1)
  {
    m[row][col] = ITI1120.readInt() ;
  }
}
```

Find Cheap Direct Flights (p. 1)

- Suppose you live in one of Escape's cities and have $D to spend. Write an algorithm that returns an array of the cities you can afford to fly to directly.

GIVENS:

Home	(the number of the city you live)
Cost	(the cost matrix)
D	(the amount you afford)
N	(the total number of cities)

RESULTS:

Cities	(an array of cities to which you can go)
NumCities	(the number of cities to which you can go)

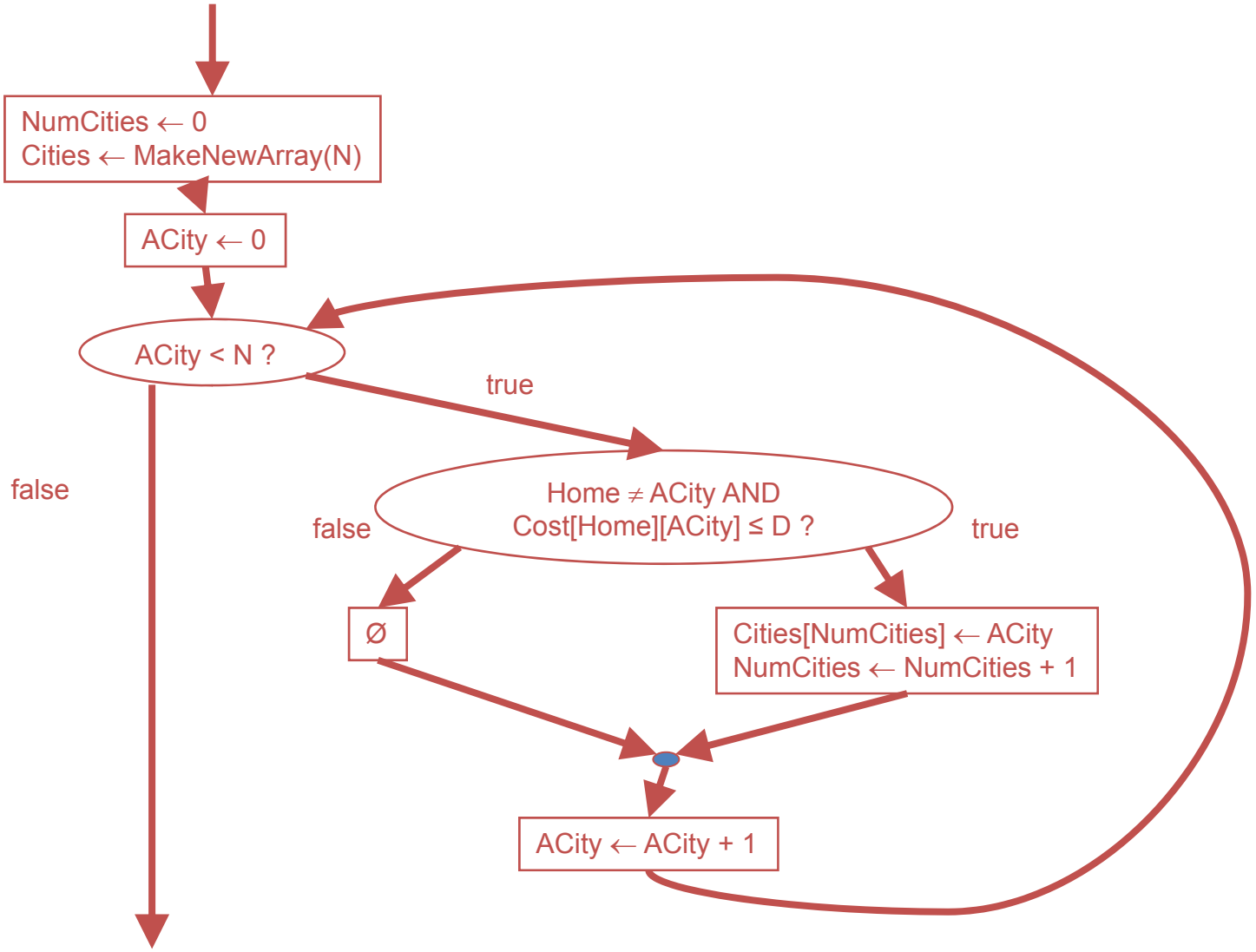
INTERMEDIATE:

ACity	(the city we are currently checking)

HEADER

(NumCities, Cities) ← CheapDirectFlights (Home, Cost, D, N)

Find Cheap Direct Flights (p. 2)



Translate into Java

```
// GIVENS:
//
// home – the city from where we want to fly
// cost – the matrix containing the costs to fly between each city
// d    - the amount of money we have to spend
// n    - the number of rows and columns in matrix 'cost'


public static int[] cheapDirectFlights ( int home, int[][] cost,
                                         int d, int n )
{
   int[] cities;          // RESULT:  an array of cities we can afford
                          //          to visit

   // INTERMEDIATES:

   int aCity ;            // The city we are currently checking (which
                          // is the column number in matrix 'cost')
   int numCities ;        // Number of cities we can afford to visit
   int[] tempCities;      // Temporary array to hold cities we can afford
                          // to visit:  it is large enough to hold all
                          // possible cities
```

Translate into Java

```
tempCities = new int[n];
numCities = 0 ;
for (aCity = 0; aCity < n ; aCity = aCity + 1 )
{
   if ( ( aCity != home ) && ( cost[home][aCity ] <= d ) )
   {
      tempCities[numCities] = aCity;
      numCities = numCities + 1;
   }
   else
   {
      ; // do nothing
   }
}
```

Translate into Java

```
   // At this point we have to get around Java's inability to return
   // more than one  value.  Create a new array of the correct length.
   // The caller can obtain the number of cities by the length of the
   // array.

   cities = new int[numCities];
   for ( aCity = 0; aCity < numCities; aCity = aCity + 1 )
   {
      cities[aCity] = tempCities[aCity];
   }

   // Now we can return the array of cities with the correct length

   return cities;
}
```

Algorithm DeleteRow (1)

GIVENS:

M	(a square matrix)
N	(number of rows and columns in M)
R	(row number to be removed)

RESULTS: (none)

MODIFIEDS:

M	(the original matrix with row R removed, and all rows moved up by one)

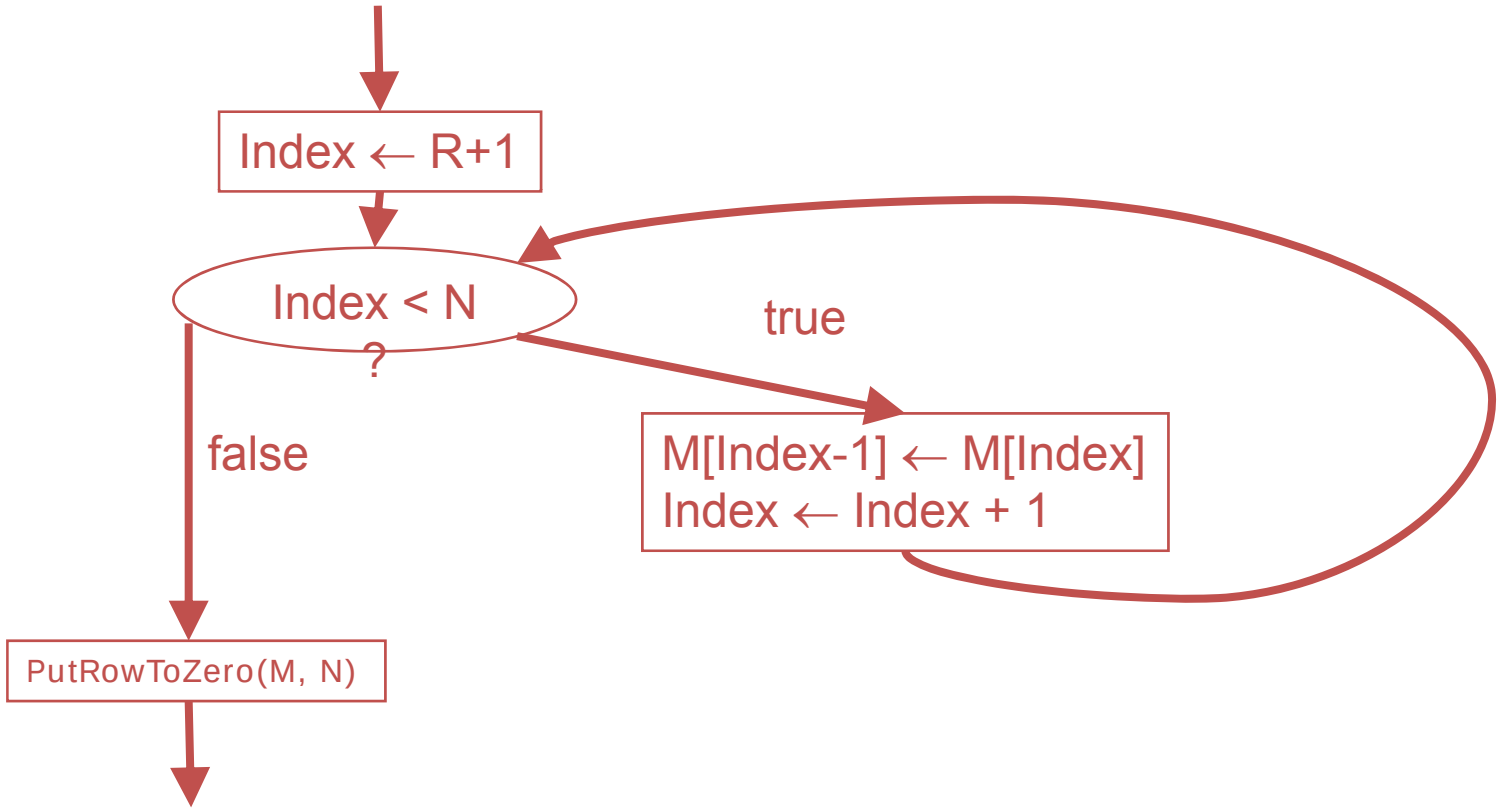
INTERMEDIATES:

Index	(index of row being moved)

HEADER:

DeleteRow(M, N, R)

Algorithm DeleteRow (2)



Algorithm PutRowToZero (1)

GIVENS:

M *(a square matrix)*

N *(size of M)*

RESULTS: (none)

MODIFIED:

M *(last row put to 0)*

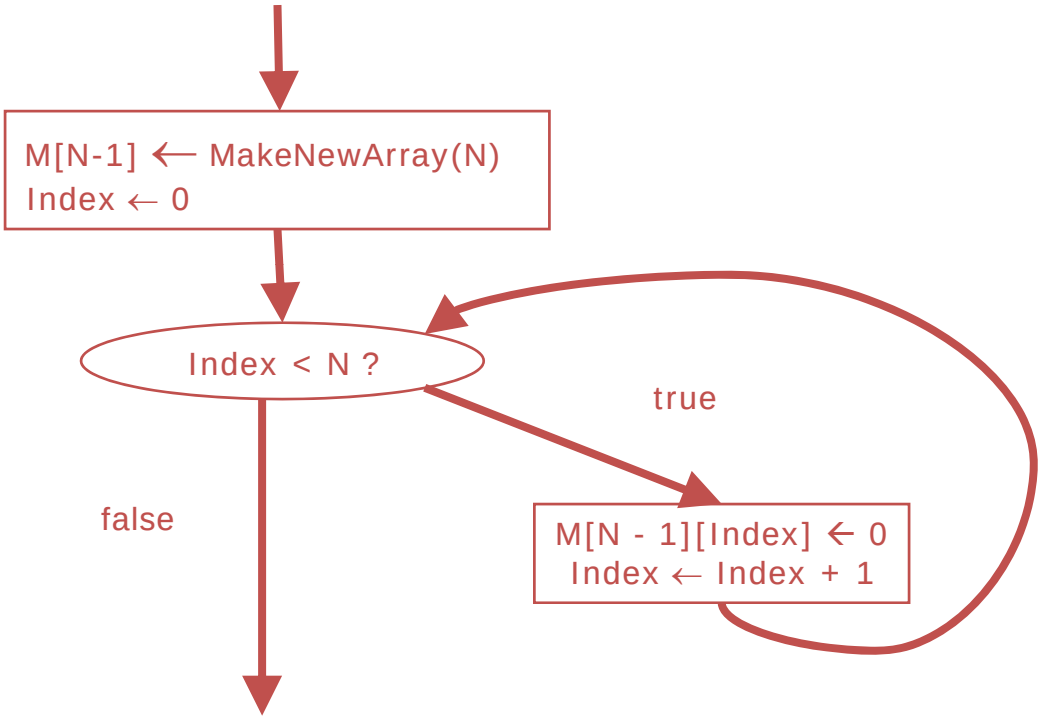
INTERMEDIATE:

Index *(index of the column)*

HEADER:

PutRowToZero(M, N)

Algorithm PutRowToZero (2)



Translation into Java

```
public static void deleteRow(int [ ][ ]m, int n, int r)
{
  int index; // INTERMEDIATE
  for (index = r + 1; index < n; index = index + 1)
  {
     m[index - 1] = m[index];
  }
  putRowToZero(m, n);
}
private static void putRowToZero(int [ ][ ]m, int n)
{
  int index; // INTERMEDIATE
  m[n-1] = new int[n];
  for (index = 0; index < n; index = index + 1)
  {
   m[n - 1][index] = 0;
  }
}
```

Algorithm DeleteColumn (1)

GIVENS:

M	(a square matrix)
N	(number of rows and columns in M)
C	(column number to be removed)

RESULTS: (none)

MODIFIEDS:

M	(the original matrix with column C removed, and all columns moved to the left by one position)

INTERMEDIATES:

RIndex	(index of current row)
CIndex	(index of column being moved)

HEADER:

DeleteColumn(M, N, C)

Algorithm DeleteColumn (2)