

Introduction to Programming (in C++)

Multi-dimensional vectors

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

Matrices

- A matrix can be considered a two-dimensional vector, i.e. a vector of vectors.

my_matrix:

3	8	1	0
5	0	6	3
7	2	9	4

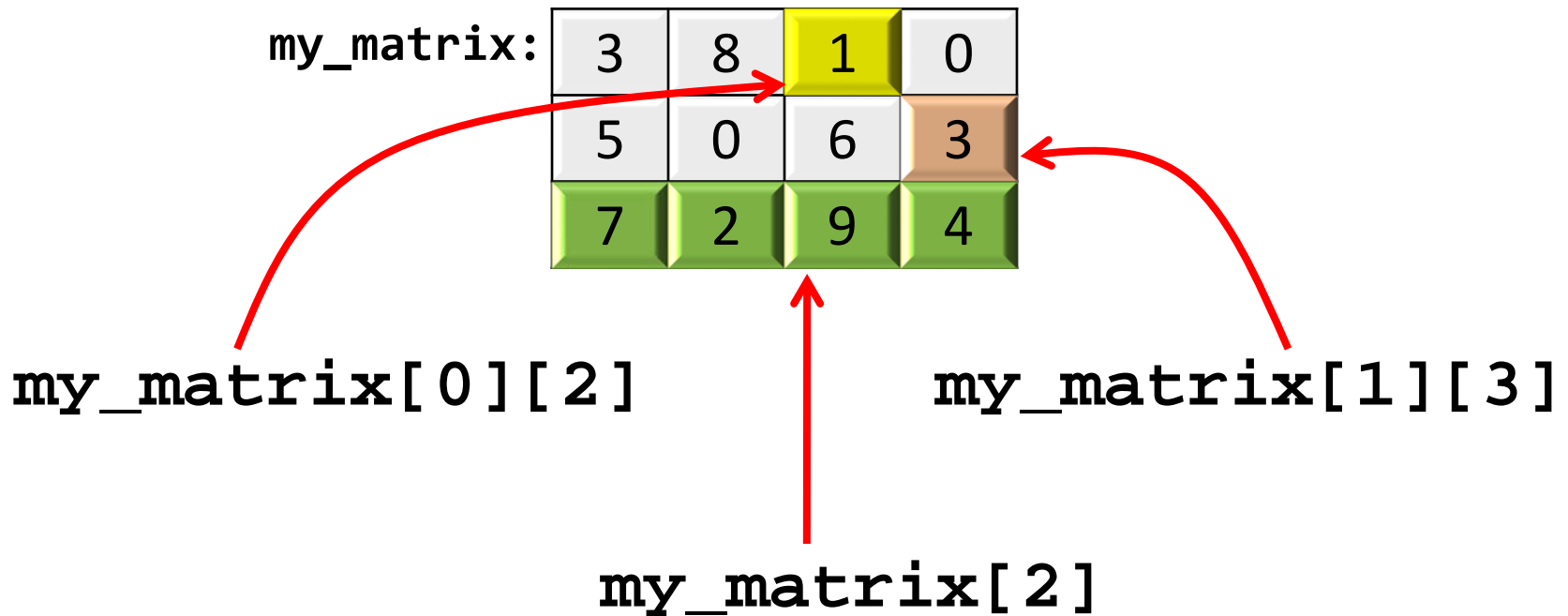
```
// Declaration of a matrix with 3 rows and 4 columns  
vector< vector<int> > my_matrix(3,vector<int>(4));
```

```
// A more elegant declaration  
typedef vector<int> Row; // One row of the matrix  
typedef vector<Row> Matrix; // Matrix: a vector of rows
```

```
Matrix my_matrix(3,Row(4)); // The same matrix as above
```

Matrices

- A matrix can be considered as a 2-dimensional vector, i.e., a vector of vectors.



n -dimensional vectors

- Vectors with any number of dimensions can be declared:

```
typedef vector<int> Dim1;  
typedef vector<Dim1> Dim2;  
typedef vector<Dim2> Dim3;  
typedef vector<Dim3> Matrix4D;  
  
Matrix4D my_matrix(5,Dim3(i+1,Dim2(n,Dim1(9))));
```

Sum of matrices

- Design a function that calculates the sum of two $n \times m$ matrices.

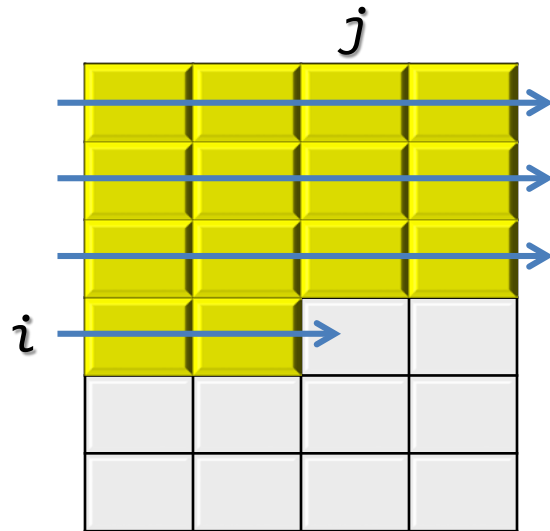
$$\begin{bmatrix} 2 & -1 \\ 0 & 1 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 2 & -1 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 2 & 0 \\ 1 & 1 \end{bmatrix}$$

```
typedef vector< vector<int> > Matrix;
```

```
Matrix matrix_sum(const Matrix& a,  
                  const Matrix& b);
```

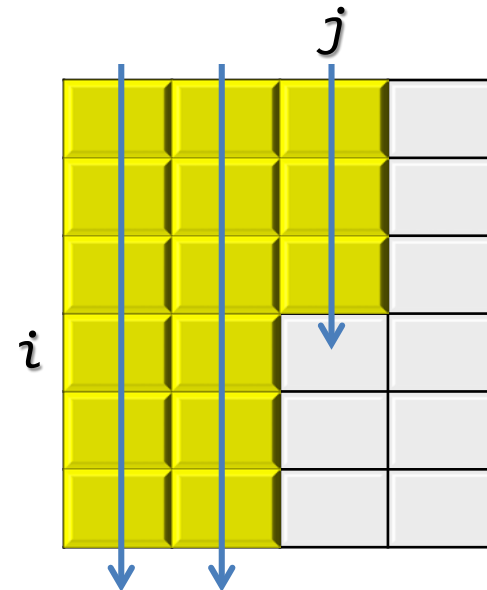
How are the elements of a matrix visited?

- By rows



For every row i
For every column j
Visit `Matrix[i][j]`

- By columns



For every column j
For every row i
Visit `Matrix[i][j]`

Sum of matrices (by rows)

```
typedef vector< vector<int> > Matrix;

// Pre: a and b are non-empty matrices and have the same size.
// Returns a+b (sum of matrices).

Matrix matrix_sum(const Matrix& a, const Matrix& b) {

    int nrows = a.size();
    int ncols = a[0].size();
    Matrix c(nrows, vector<int>(ncols));

    for (int i = 0; i < nrows; ++i) {
        for (int j = 0; j < ncols; ++j) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    return c;
}
```

Sum of matrices (by columns)

```
typedef vector< vector<int> > Matrix;

// Pre: a and b are non-empty matrices and have the same size.
// Returns a+b (sum of matrices).

Matrix matrix_sum(const Matrix& a, const Matrix& b) {

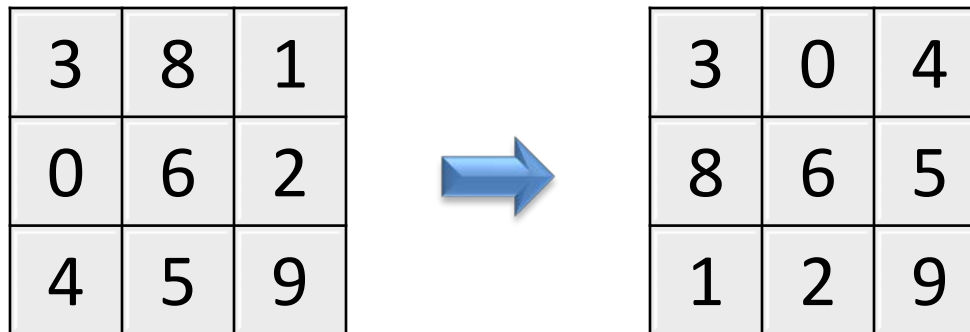
    int nrows = a.size();
    int ncols = a[0].size();
    Matrix c(nrows, vector<int>(ncols));

    for (int j = 0; j < ncols; ++j) {
        for (int i = 0; i < nrows; ++i) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    return c;
}
```


Transpose a matrix

- Design a procedure that transposes a square matrix in place:

```
void Transpose (Matrix& m);
```



- Observation: we need to swap the upper with the lower triangular matrix. The diagonal remains intact.

Transpose a matrix

```
// Interchanges two values
```

```
void swap(int& a, int& b) {  
    int c = a;  
    a = b;  
    b = c;  
}
```

```
// Pre: m is a square matrix
```

```
// Post: m contains the transpose of the input matrix
```

```
void Transpose(Matrix& m) {  
    int n = m.size();  
    for (int i = 0; i < n - 1; ++i) {  
        for (int j = i + 1; j < n; ++j) {  
            swap(m[i][j], m[j][i]);  
        }  
    }  
}
```

Is a matrix symmetric?

- Design a procedure that indicates whether a matrix is symmetric:

```
bool is_symmetric(const Matrix& m);
```

3	0	4
0	6	5
4	5	9

symmetric

3	0	4
0	6	5
4	2	9

not symmetric

- Observation: we only need to compare the upper with the lower triangular matrix.

Is a matrix symmetric?

```
// Pre: m is a square matrix  
// Returns true if m is symmetric, and false otherwise
```

```
bool is_symmetric(const Matrix& m) {  
    int n = m.size();  
    for (int i = 0; i < n - 1; ++i) {  
        for (int j = i + 1; j < n; ++j) {  
            if (m[i][j] != m[j][i]) return false;  
        }  
    }  
    return true;  
}
```

Search in a matrix

- Design a procedure that finds a value in a matrix. If the value belongs to the matrix, the procedure will return the location (i, j) at which the value has been found.

```
// Pre:  m is a non-empty matrix  
// Post: i and j define the location of a cell  
//       that contains the value x in m.  
//       In case x is not in m, then i = j = -1.
```

```
void search(const Matrix& m, int x, int& i, int& j);
```

Search in a matrix

```
// Pre: m is a non-empty matrix
// Post: i and j define the location of a cell
//       that contains the value x in M.
//       In case x is not in m, then i = j = -1

void search(const Matrix& m, int x, int& i, int& j) {
    int nrows = m.size();
    int ncols = m[0].size();
    bool found = false;
    int i = 0;
    while (not found and i < nrows) {
        int j = 0;
        while (not found and j < ncols) {
            if (m[i][j] == x) found = true;
            ++j;
        }
        ++i;
    }
    if (not found) {
        i = -1;
        j = -1;
    }
}
```

Search in a sorted matrix

- A sorted matrix m is one in which

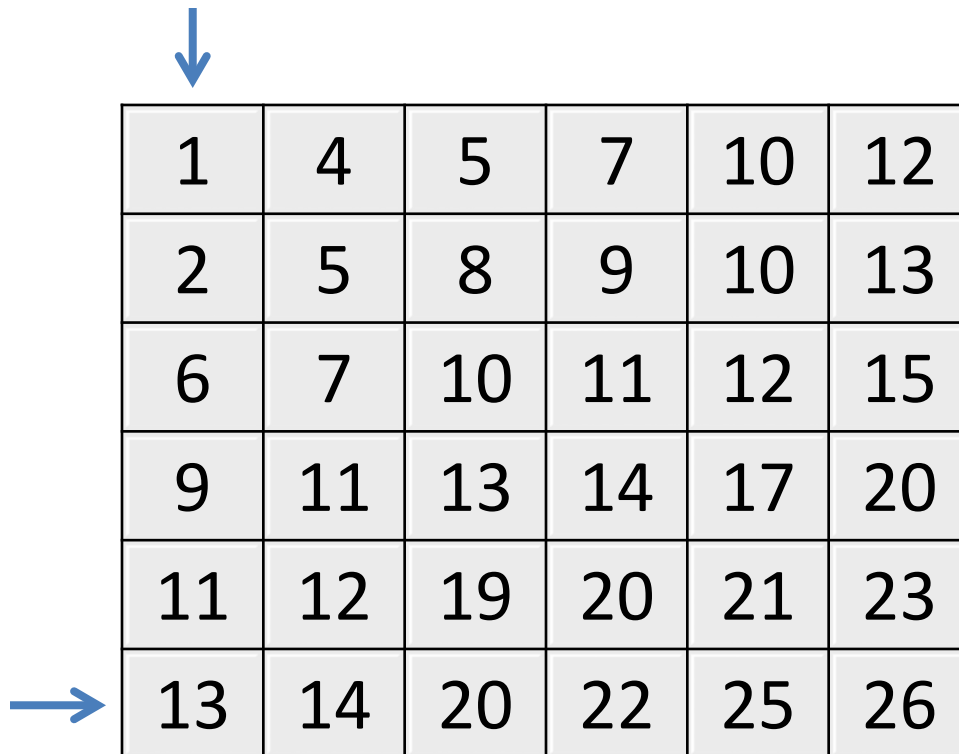
$$m[i][j] \leq m[i][j+1]$$

$$m[i][j] \leq m[i+1][j]$$

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- Example: let us find 10 in the matrix. We look at the lower left corner of the matrix.
- Since $13 > 10$, the value cannot be found in the last row.

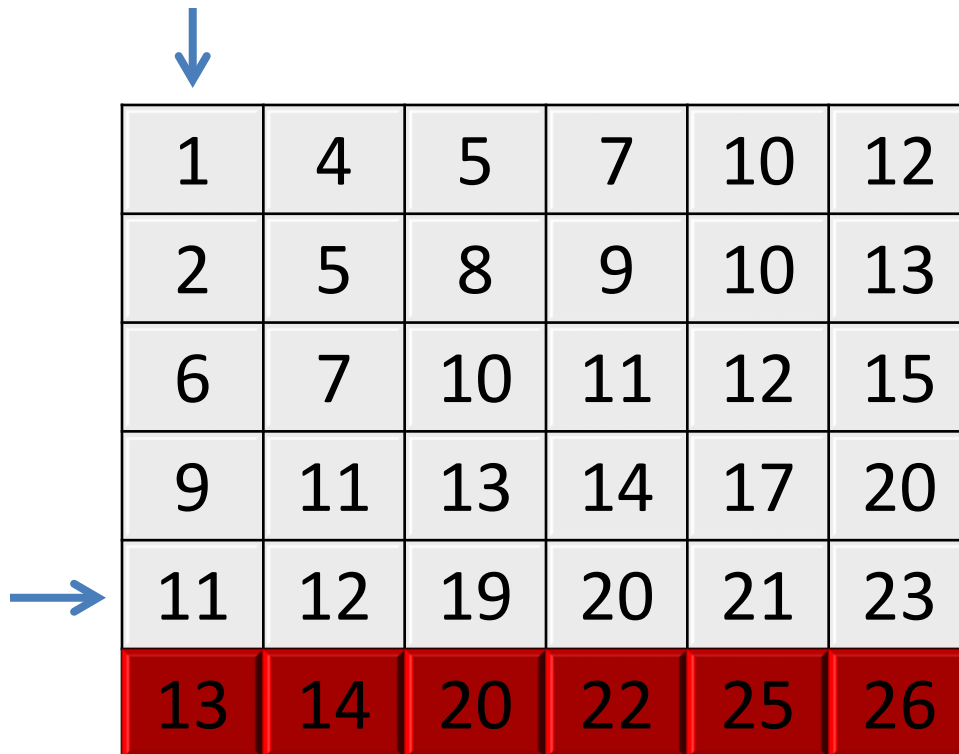


A 6x6 matrix of sorted numbers. The values increase from top-left to bottom-right. A blue arrow points down to the first row, and another blue arrow points right to the last row.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- We look again at the lower left corner of the remaining matrix.
- Since $11 > 10$, the value cannot be found in the row.

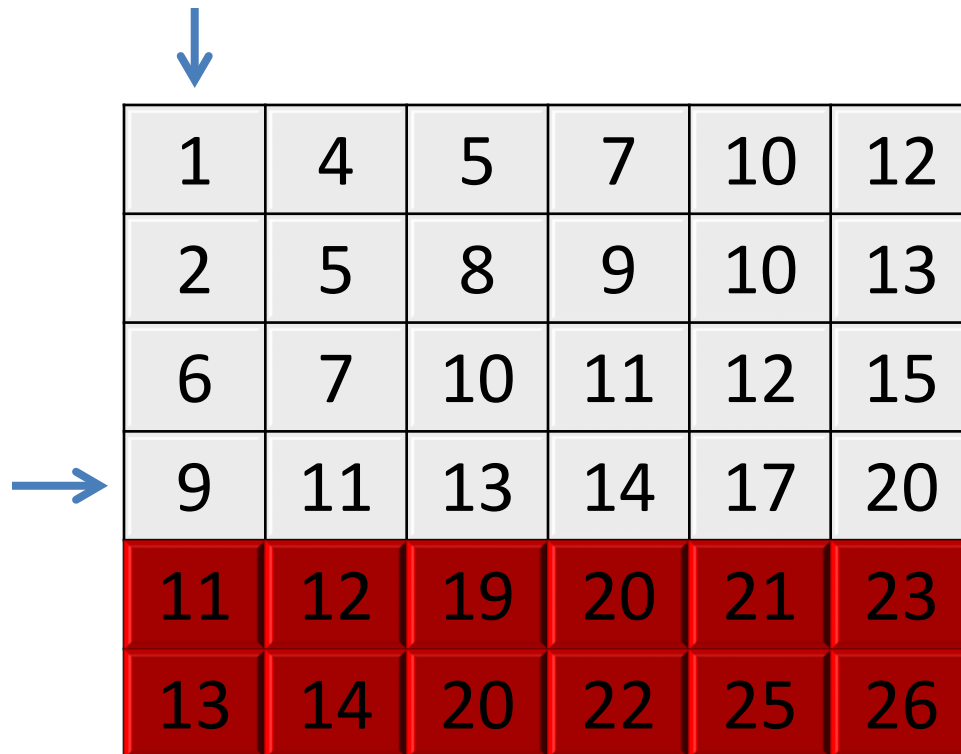


A 6x6 matrix of sorted numbers. The first row is [1, 4, 5, 7, 10, 12], the second row is [2, 5, 8, 9, 10, 13], the third row is [6, 7, 10, 11, 12, 15], the fourth row is [9, 11, 13, 14, 17, 20], the fifth row is [11, 12, 19, 20, 21, 23], and the sixth row is [13, 14, 20, 22, 25, 26]. A blue arrow points down to the first row, and another blue arrow points right to the fifth row. The sixth row is highlighted in red.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- Since $9 < 10$, the value cannot be found in the column.



A 6x6 matrix of sorted numbers. The first column is highlighted with a blue arrow pointing down, and the fourth row is highlighted with a blue arrow pointing right. The bottom two rows (rows 5 and 6) are highlighted in red.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- Since $11 > 10$, the value cannot be found in the row.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

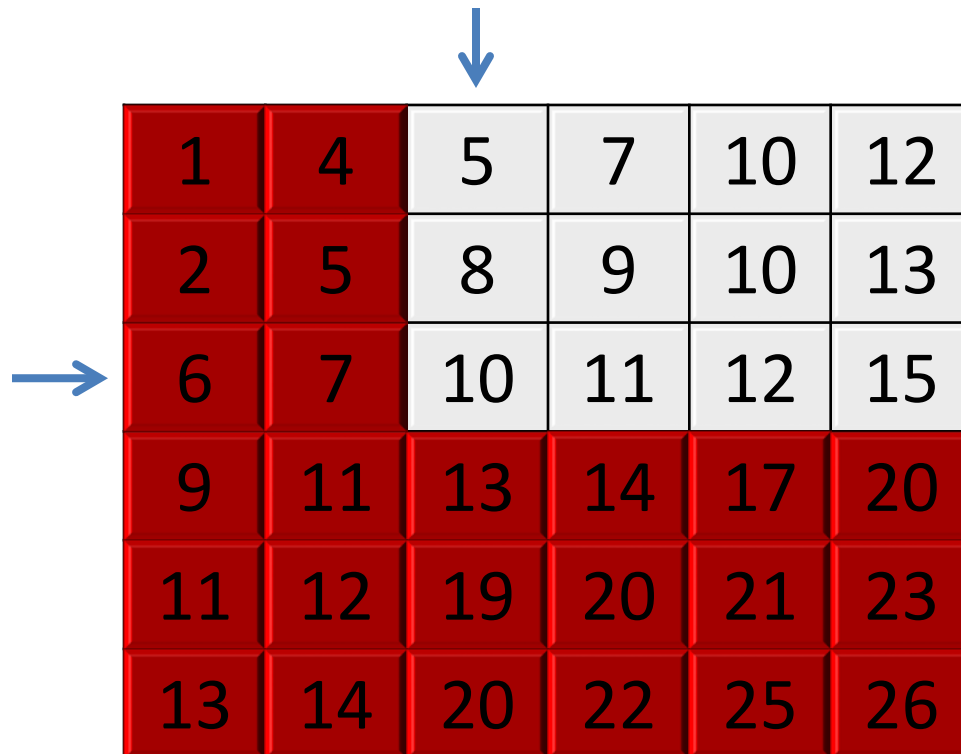
Search in a sorted matrix

- Since $7 < 10$, the value cannot be found in the column.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- The element has been found!

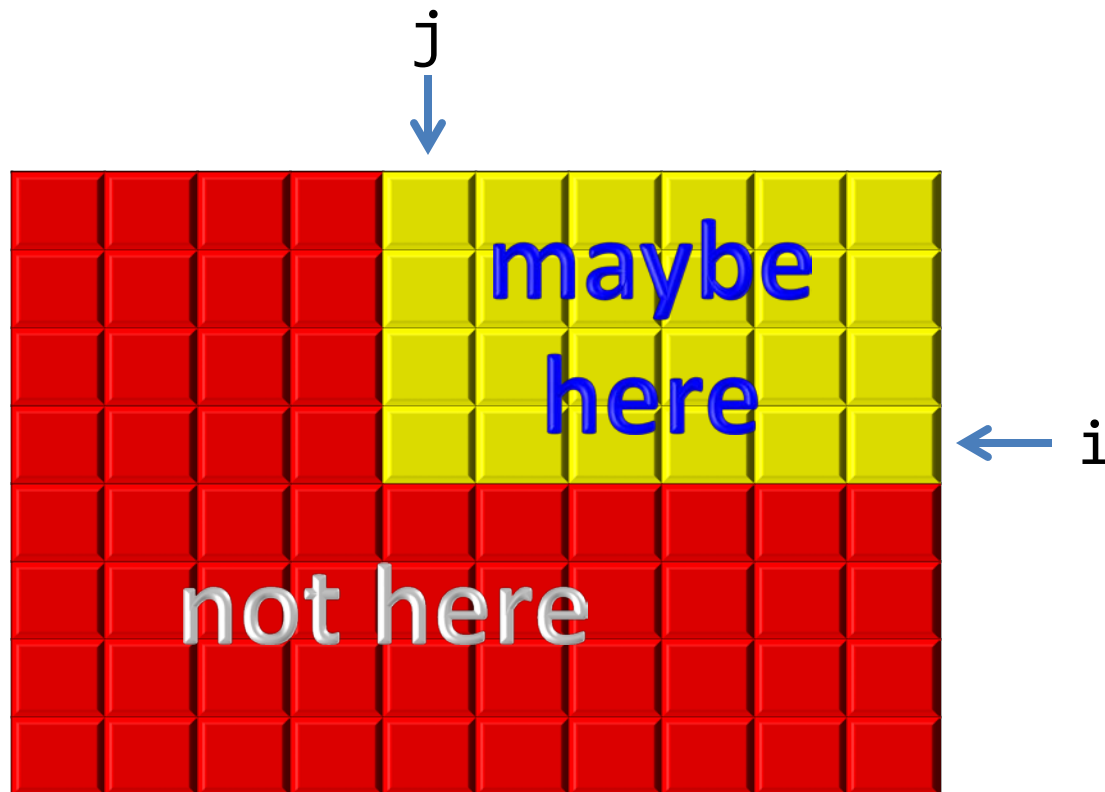


A 6x6 matrix of numbers, sorted in ascending order both row-wise and column-wise. The matrix is shown with a search path highlighted in red. The path starts at the top-left corner (1, 1) and moves right to (1, 2), then down to (2, 2), then down to (3, 2), then right to (3, 3), then right to (3, 4), then right to (3, 5), then right to (3, 6), then down to (4, 6), then down to (5, 6), and finally down to (6, 6). The element '10' in the third row, second column is highlighted in white, indicating it has been found. A blue arrow points to this element from the left, and another blue arrow points to it from above.

1	4	5	7	10	12
2	5	8	9	10	13
6	7	10	11	12	15
9	11	13	14	17	20
11	12	19	20	21	23
13	14	20	22	25	26

Search in a sorted matrix

- *Invariant*: if the element is in the matrix, then it is located in the sub-matrix $[0\dots i, j\dots n\text{cols}-1]$



Search in a sorted matrix

```
// Pre: m is non-empty and sorted by rows and columns
//       in ascending order.
// Post: i and j define the location of a cell that contains the value
//       x in m. In case x is not in m, then i=j=-1.
```

```
void search(const Matrix& m, int x, int& i, int& j) {
    int nrows = m.size();
    int ncols = m[0].size();
    i = nrows - 1;
    j = 0;
    bool found = false;
    // Invariant: x can only be found in M[0..i,j..ncols-1]
    while (not found and i >= 0 and j < ncols) {
        if (m[i][j] < x) ++j;
        else if (m[i][j] > x) --i;
        else found = true;
    }

    if (not found) {
        i = -1;
        j = -1;
    }
}
```

Search in a sorted matrix

- What is the largest number of iterations of a search algorithm in a matrix?

Unsorted matrix	$nrows \times ncols$
Sorted matrix	$nrows + ncols$

- The search algorithm in a sorted matrix cannot start in all of the corners of the matrix. Which corners are suitable?

Matrix multiplication

- Design a function that returns the multiplication of two matrices.

2	-1	0	1
1	3	2	0

 ×

1	2	-1
3	0	2
-1	1	3
2	-1	4

 =

1	3	0
8	4	11

```
// Pre: a is a non-empty n×m matrix,  
//      b is a non-empty m×p matrix  
// Returns a×b (an n×p matrix)
```

```
Matrix multiply(const Matrix& a, const Matrix& b);
```

Matrix multiplication

```
// Pre: a is a non-empty n×m matrix, b is a non-empty m×p matrix.  
// Returns a×b (an n×p matrix).
```

```
Matrix multiply(const Matrix& a, const Matrix& b) {  
    int n = a.size();  
    int m = a[0].size();  
    int p = b[0].size();  
    Matrix c(n, vector<int>(p));  
  
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < p; ++j) {  
            int sum = 0;  
            for (int k = 0; k < m; ++k) {  
                sum = sum + a[i][k]*b[k][j];  
            }  
            c[i][j] = sum;  
        }  
    }  
    return c;  
}
```

Matrix multiplication

```
// Pre: a is a non-empty n×m matrix, b is a non-empty m×p matrix.  
// Returns a×b (an n×p matrix).
```

```
Matrix multiply(const Matrix& a, const Matrix& b) {  
    int n = a.size();  
    int m = a[0].size();  
    int p = b[0].size();  
    Matrix c(n, vector<int>(p, 0));
```

Initialized
to zero

```
    for (int i = 0; i < n; ++i) {  
        for (int j = 0; j < p; ++j) {  
            for (int k = 0; k < m; ++k) {  
                c[i][j] += a[i][k]*b[k][j];  
            }  
        }  
    }  
    return c;  
}
```

The loops can
be in any order

Accumulation

Matrix multiplication

```
// Pre: a is a non-empty n×m matrix, b is a non-empty m×p matrix.  
// Returns a×b (an n×p matrix).
```

```
Matrix multiply(const Matrix& a, const Matrix& b) {  
    int n = a.size();  
    int m = a[0].size();  
    int p = b[0].size();  
    Matrix c(n, vector<int>(p, 0));  
  
    for (int j = 0; j < p; ++j) {  
        for (int k = 0; k < m; ++k) {  
            for (int i = 0; i < n; ++i) {  
                c[i][j] += a[i][k]*b[k][j];  
            }  
        }  
    }  
    return c;  
}
```