

# Многомерни масиви и алгоритми

УП практикум, 2ра група  
Богомил Стоянов  
Виолета Кастрева

# Многомерни масиви - основна концепция

Спомняте ли си аналогията, че масивите приличат на рафт, на който нареждаме еднотипни неща?

Сега нека си представим, че имаме нещо такова ->

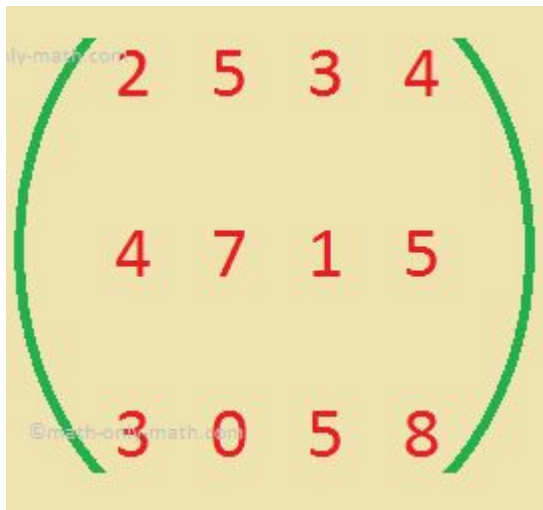
Имаме нещо като рафт от рафтове.



# Многомерни масиви - основна концепция

Именно това са двумерните масиви - масив от масиви.

Можем също така да направим аналогия с алгебрата, тъй като структурата им е като на матрици (или таблици). Всъщност точно така ще ги наричаме и тук, матрици.



A 3x4 matrix of red numbers enclosed in large green parentheses. The numbers are arranged as follows:

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

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	0	1	1	1

# Матрици в C++



```
1 int main() {  
2     //деклариране, първият индекс е брой редове  
3     //вторият - по колко елементи има на ред (стълбове)  
4     int matrix[3][3];  
5  
6     //деклариране и инициализация (един начин)  
7     int secondMatrix[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
8     //това ще ни отговаря на  
9     // 1 2 3  
10    // 4 5 6  
11    // 7 8 9  
12  
13    return 0;  
14 }
```

# Достъп до елементите в матрица

Всеки елемент достъпваме с два индекса - за реда и за стълба.

Например, ако имаме  $x[3][4]$ , то достъпът до елементите ще бъде съответно:

	Col 1	Col 2	Col 3	Col 4
Row 1	$x[0][0]$	$x[0][1]$	$x[0][2]$	$x[0][3]$
Row 2	$x[1][0]$	$x[1][1]$	$x[1][2]$	$x[1][3]$
Row 3	$x[2][0]$	$x[2][1]$	$x[2][2]$	$x[2][3]$

# Достъп до елементите в матрица - пример

```
1 #include <iostream>
2
3 int main() {
4     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
5
6     //достъп до елемента
7     std::cout << "Accessed Element: " << matrix[0][1] << std::endl; // изход: 2
8
9     //промяна на стойността на елемент
10    matrix[0][1] = 10; // променяме я от 2 на 10
11
12    std::cout << "Modified Element: " << matrix[0][1] << std::endl; // изход: 10
13
14    return 0;
15 }
16
```

# Обхождане на елементите на матрица

Използваме вложени цикли, за да обходим всички елементи, обикновено външният цикъл е за редове, вътрешният - за стълбове.

Изход:

1 2 3

4 5 6

7 8 9

```
1 #include <iostream>
2
3 int main() {
4     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
5
6     // Iterating over the matrix with nested loops
7     for(int i = 0; i < 3; i++) {      // Outer loop for rows
8         for(int j = 0; j < 3; j++) { // Inner loop for columns
9             std::cout << matrix[i][j] << " "; // Access and print each element
10        }
11        std::cout << std::endl; // Newline at the end of each row
12    }
13
14    return 0;
15 }
16
```

# Въвеждане на матрица от конзолата

```
1 #include <iostream>
2 using namespace std;
3
4 const int MAX_ROWS = 100; // Maximum number of rows
5 const int MAX_COLS = 100; // Maximum number of columns
6
7 int main() {
8     int matrix[MAX_ROWS][MAX_COLS];
9     int rows, cols;
10
11     cout << "Enter the number of rows: ";
12     cin >> rows;
13     cout << "Enter the number of columns: ";
14     cin >> cols;
15
16     cout << "Enter matrix elements:" << endl;
17     for(int i = 0; i < rows; i++) {
18         for(int j = 0; j < cols; j++) {
19             cout << "Enter element [" << i << "][" << j << "]: ";
20             cin >> matrix[i][j];
21         }
22     }
23
24     return 0;
25 }
26
```



# Основни операции с матрици

Ще разгледаме няколко основни задачи, за да придобием интуиция за обхождането и работата с матрици, а именно:

- сума на елементите
- сума на елементите по редове, и по стълбове
- транспониране (на квадратни матрици) (не е много основно, но пък е забавно)

# Сума на всички елементи в матрица



```
1 #include <iostream>
2
3 int main() {
4     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
5     int sum = 0;
6
7     for(int i = 0; i < 3; i++) {
8         for(int j = 0; j < 3; j++) {
9             sum += matrix[i][j];
10        }
11    }
12
13    std::cout << "Sum of all elements: " << sum << std::endl;
14
15    return 0;
16 }
17
```

# Суми по редове

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
6
7     for(int i = 0; i < 3; i++) {
8         int rowSum = 0;
9         for(int j = 0; j < 3; j++) {
10             rowSum += matrix[i][j];
11         }
12         cout << "Sum of row " << i << ": " << rowSum << endl;
13     }
14
15
16     return 0;
17 }
18
```

# Суми по стълбове (нека по принцип избягваме std)



```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
6
7     for(int j = 0; j < 3; j++) {
8         int colSum = 0;
9         for(int i = 0; i < 3; i++) {
10             colSum += matrix[i][j];
11         }
12         cout << "Sum of column " << j << ": " << colSum << endl;
13     }
14
15     return 0;
16 }
17
```

# Транспониране



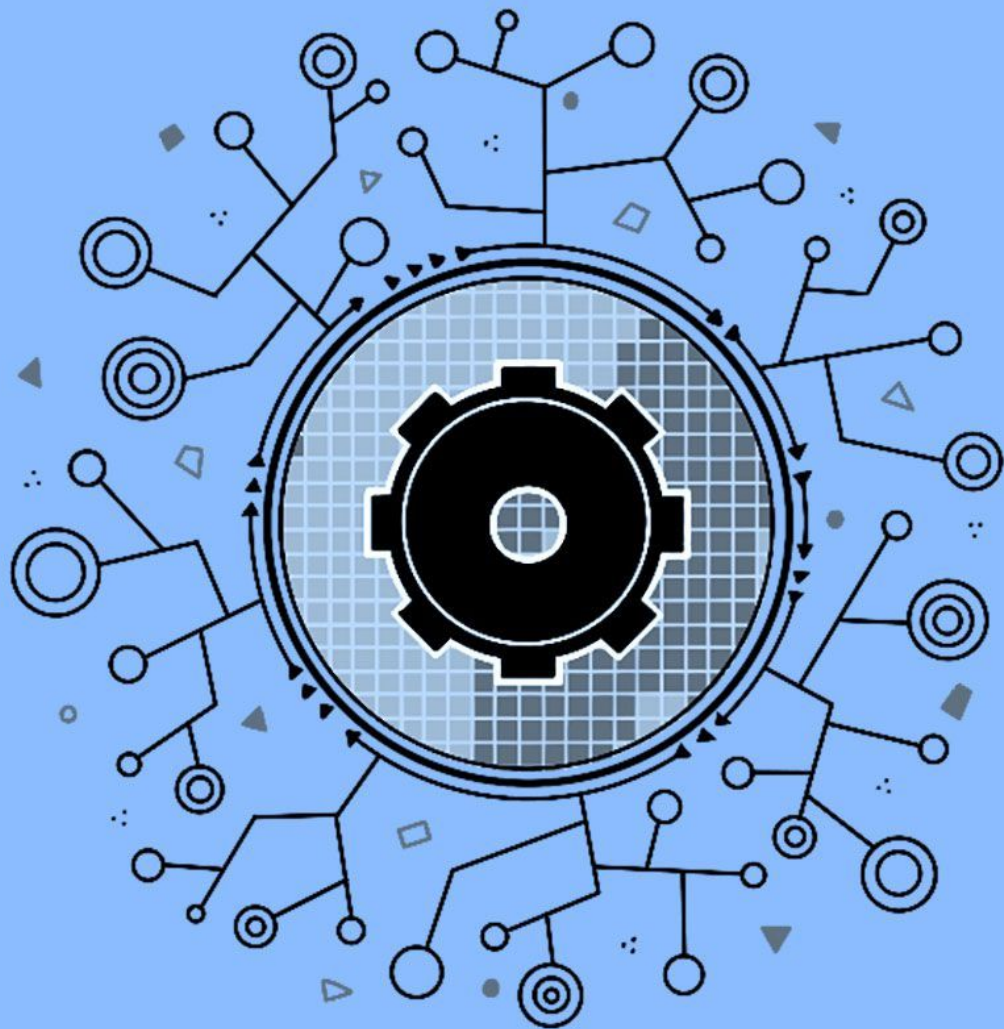
```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
6     int transpose[3][3];
7
8     for(int i = 0; i < 3; i++) {
9         for(int j = 0; j < 3; j++) {
10             transpose[j][i] = matrix[i][j];
11         }
12     }
13
14     return 0;
15 }
16
```

# Ами за тримерни масиви?

По индукция, всички следващи размерности третираме по аналогичен начин.

Например за тримерен масив ще имаме `int 3D[3][3][3]` и обхождането ще бъде с 3 вложени цикъла.

Повече от тримерни масиви рядко се ползват в практиката.



## Алгоритъм?

A set of instructions  
for solving a problem

## Алгоритми - увод

Ще разгледаме основни  
алгоритми за търсене на  
елементи в масиви,  
сортиране на елементи и т.н.

# Линейно търсене

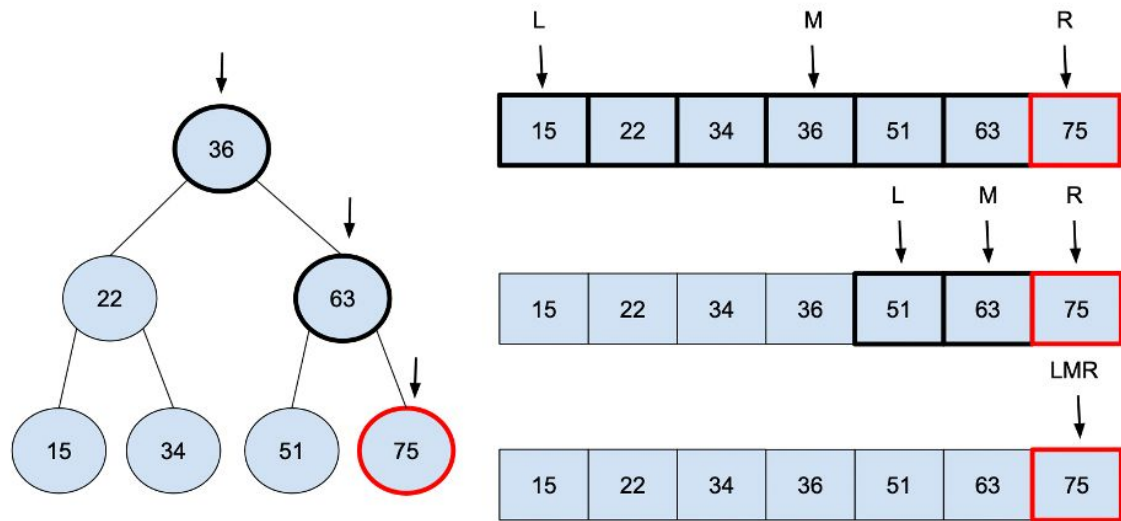
Вече е интуитивно ясно, как най-лесно да намерим някакъв елемент в масив:

```
1 #include <iostream>
2
3 int linearSearch(int arr[], int n, int x) {
4     for(int i = 0; i < n; i++) {
5         if(arr[i] == x)
6             return i; // Found x, return the index
7     }
8     return -1; // x not found
9 }
10 int main() {
11     int arr[] = {2, 3, 4, 10, 40};
12     int x = 10;
13     int n = 5; // Manually specifying the number of elements in arr
14     int result = linearSearch(arr, n, x);
15     if (result == -1)
16         std::cout << "Element is not present in array";
17     else
18         std::cout << "Element is present at index " << result;
19     std::cout << std::endl;
20     return 0;
21 }
22
```



# Двоично търсене

Сега да разгледаме нещо, работещо доста по-бързо, алгоритъм, с който търсим елемент  $X$  в масив, който обаче трябва да е сортиран



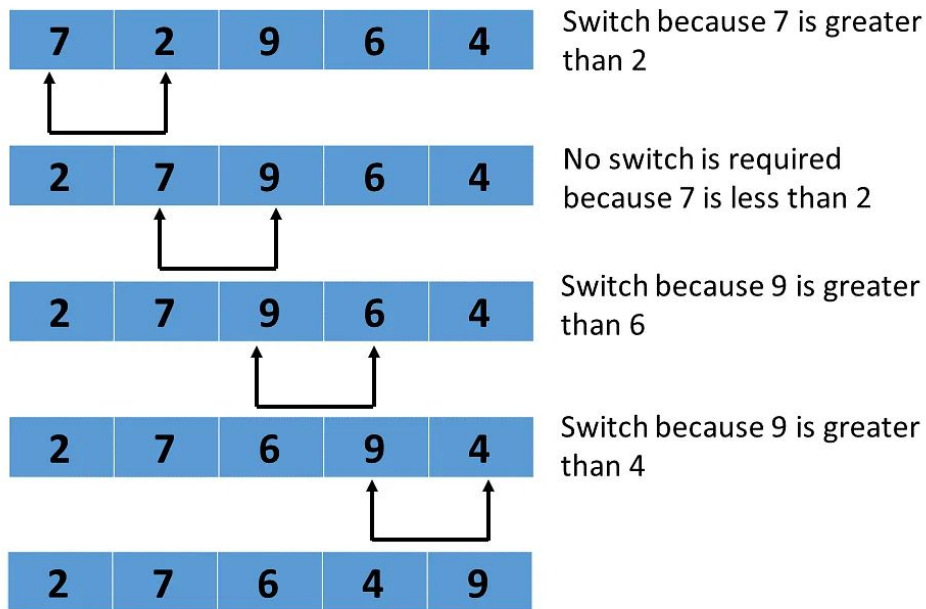
# Двоично търсене - имплементация



```
1 #include <iostream>
2
3 int binarySearch(const int arr[], int size, int target) {
4     int left = 0, right = size - 1;
5
6     while (left <= right) {
7         int mid = left + (right - left) / 2;
8
9         if (arr[mid] == target) {
10             return mid; // Target value found
11         } else if (arr[mid] < target) {
12             left = mid + 1; // Target is in the right half
13         } else {
14             right = mid - 1; // Target is in the left half
15         }
16     }
17
18     return -1; // Target not found
19 }
```

# Сортиране - Bubble Sort

Ще видим един стандартен, сравнително прост, метод за сортиране на масив от елементи



# Bubble Sort - имплементация



```
1 void bubbleSort(int arr[], int n) {  
2     for (int i = 0; i < n-1; i++) {  
3         for (int j = 0; j < n-i-1; j++) {  
4             if (arr[j] > arr[j+1]) {  
5                 // Swap arr[j] and arr[j+1]  
6                 int temp = arr[j];  
7                 arr[j] = arr[j+1];  
8                 arr[j+1] = temp;  
9             }  
10        }  
11    }  
12 }
```

Край