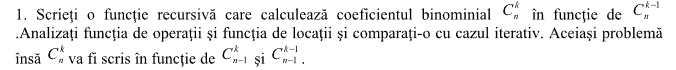
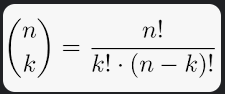
Structuri de Date

Laboratorul 1 – Tema 1 - Nicoleta Radu

# 1. Coeficientul Binomial





// Complexitate: O(n\*max(k,n-k))

// C(n,k)

int CoeficientBinomial\_1(int n, int k)

{

if (k > n)

{

return 0;

}

if (k == 0 || k == n)

{

return 1;

}

return CoeficientBinomial\_1(n - 1, k - 1)

+ CoeficientBinomial\_1(n - 1, k);

}

// C(n-1,k)

int CoeficientBinomial\_2(int n, int k)

{

if (k > n)

{

return 0;

}

if (k == 0 || k == n)

{

return 1;

}

return CoeficientBinomial\_1((n - 1) - 1, k - 1)

+ CoeficientBinomial\_1((n - 1) - 1, k);

}

// C(n,k - 1)

int CoeficientBinomial\_3(int n, int k)

{

if (k > n)

{

return 0;

}

if (k == 0 || k == n)

{

return 1;

}

return CoeficientBinomial\_3(n - 1, (k - 1) - 1)

+ CoeficientBinomial\_3(n - 1, k);

}

// C(n - 1,k - 1)

int CoeficientBinomial\_4(int n, int k)

{

if (k > n)

{

return 0;

}

if (k == 0 || k == n)

{

return 1;

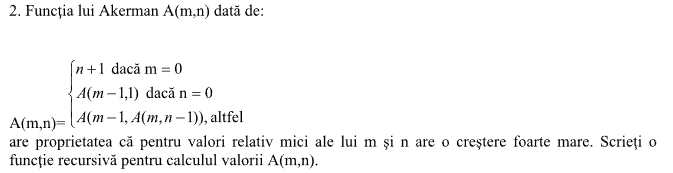
}

return CoeficientBinomial\_4((n - 1) - 1, (k - 1) - 1)

+ CoeficientBinomial\_4((n - 1) - 1, k);

}

# 2. Functia Akerman



int Akerman(int m, int n)

{

if (m == 0)

{

return n + 1;

}

else if (n == 0)

{

return Akerman(m - 1, 1);

}

else

{

return Akerman(m - 1, Akerman(m, n - 1));

}

}

# 3. Functia Bijectiva



int\* submultime(int yourArray[], int yourArraySize, int& newSize)

{

// newSize va fi dimensiunea submultimii gasite

int maximum{ 1 }, count{1}, maximumIndex{0};

for (size\_t i = 1; i < yourArraySize; i++)

{

if (yourArray[i] > yourArray[i - 1])

{

/\*

numaram cate secvente consecutive de numere

crescatoare gasim

\*/

count++;

}

else

{

if (maximum < count)

{

/\*

salvam cea mai mare secventa de numere

crescatoare in "maximum"

\*/

maximum = count;

/\*

salvam indexul de unde incepe secventa de

numere crescatoare in "maximumIndex"

\*/

maximumIndex = i - maximum;

}

/\*

resetam variabila "count"

\*/

count = 1;

}

}

if (maximum < count)

{

maximum = count;

maximumIndex = yourArraySize - maximum;

}

newSize = maximum;

int\* result = new int[maximum];

/\*

salvam submultimea in noul array "rezultat"

\*/

int index{ 0 };

for (size\_t i = maximumIndex; i < static\_cast<unsigned int>(maximum) + maximumIndex; i++)

{

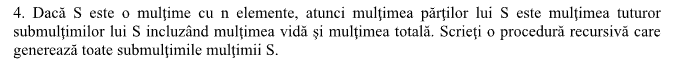
result[index++] = yourArray[i];

}

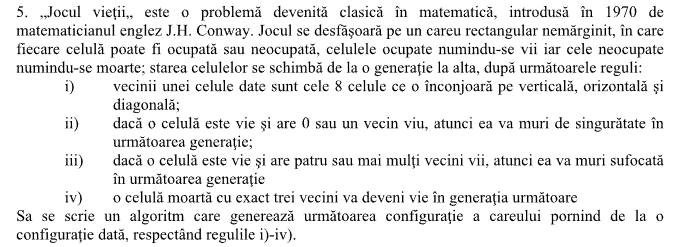
return result;

}

# 4. Submultimile lui S



# 5. Jocul Vietii



#include <iostream>

using namespace std;

typedef int\*\* Matrix;

int CountLivingCells(Matrix yourMatrix, int Value ,int Row, int Column, unsigned int matrixRows, unsigned int matrixColumns)

{

int count = 0;

// Inside Matrix > 0

if ((Row > 0 && Row < matrixRows - 1) && (Column > 0 && Column < matrixColumns - 1))

{

// Upper left corner

if (yourMatrix[Row - 1][Column - 1] == Value)

{

count++;

}

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Lower left corner

if (yourMatrix[Row + 1][Column - 1] == Value)

{

count++;

}

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

// Lower right corner

if (yourMatrix[Row + 1][Column + 1] == Value)

{

count++;

}

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

// Upper right corner

if (yourMatrix[Row - 1][Column + 1] == Value)

{

count++;

}

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

}

// Upper Row

else if (Row == 0 && ((Column > 0) && (Column < matrixColumns - 1)))

{

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Lower left corner

if (yourMatrix[Row + 1][Column - 1] == Value)

{

count++;

}

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

// Lower right corner

if (yourMatrix[Row + 1][Column + 1] == Value)

{

count++;

}

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

}

// Left Column

else if (Column == 0 && ((Row > 0) && (Row < matrixRows - 1)))

{

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

// Upper right corner

if (yourMatrix[Row - 1][Column + 1] == Value)

{

count++;

}

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

// Lower right corner

if (yourMatrix[Row + 1][Column + 1] == Value)

{

count++;

}

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

}

// Lower Row

else if (Row == matrixRows - 1 && ((Column > 0) && (Column < matrixColumns - 1)))

{

// Upper left corner

if (yourMatrix[Row - 1][Column - 1] == Value)

{

count++;

}

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

// Upper right corner

if (yourMatrix[Row - 1][Column + 1] == Value)

{

count++;

}

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

}

// Right Column

else if (Column == matrixColumns - 1 && ((Row > 0) && (Row < matrixColumns - 1)))

{

// Upper left corner

if (yourMatrix[Row - 1][Column - 1] == Value)

{

count++;

}

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Lower left corner

if (yourMatrix[Row + 1][Column - 1] == Value)

{

count++;

}

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

}

// Upper Left Corner

else if (Row == 0 && Column == 0)

{

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

// Lower right corner

if (yourMatrix[Row + 1][Column + 1] == Value)

{

count++;

}

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

}

// Lower Left Corner

else if (Row == matrixRows - 1 && Column == 0)

{

// Right

if (yourMatrix[Row][Column + 1] == Value)

{

count++;

}

// Upper right corner

if (yourMatrix[Row - 1][Column + 1] == Value)

{

count++;

}

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

}

// Lower Right Corner

else if (Row == matrixRows - 1 && Column == matrixColumns - 1)

{

// Upper left corner

if (yourMatrix[Row - 1][Column - 1] == Value)

{

count++;

}

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Up

if (yourMatrix[Row - 1][Column] == Value)

{

count++;

}

}

// Upper Right Corner

else if (Row == 0 && Column == matrixColumns - 1)

{

// Left

if (yourMatrix[Row][Column - 1] == Value)

{

count++;

}

// Lower left corner

if (yourMatrix[Row + 1][Column - 1] == Value)

{

count++;

}

// Down

if (yourMatrix[Row + 1][Column] == Value)

{

count++;

}

}

return count;

}

bool IsALivingCell(Matrix yourMatrix, unsigned int Row, unsigned int Column)

{

if (yourMatrix[Row][Column] == 1)

{

return true;

}

else

{

return false;

}

}

bool IsADeadCell(Matrix yourMatrix, unsigned int Row, unsigned int Column)

{

if (yourMatrix[Row][Column] == 0)

{

return true;

}

else

{

return false;

}

}

Matrix InstantiateMatrix(unsigned int& Rows, unsigned int& Columns)

{

Matrix TEMP;

TEMP = new int\* [Rows];

for (size\_t i = 0; i < Rows; i++)

{

TEMP[i] = new int[Columns];

}

return TEMP;

}

void ManualMatrixGeneration(Matrix yourMatrix, unsigned int Rows, unsigned int Columns)

{

for (size\_t i = 0; i < Rows; i++)

{

for (size\_t j = 0; j < Columns; j++)

{

// user input

cout << "[" << i << "," << j << "] = ";

cin >> yourMatrix[i][j];

}

}

}

void RandomMatrixGeneration(Matrix yourMatrix, unsigned int Rows, unsigned int Columns)

{

for (size\_t i = 0; i < Rows; i++)

{

for (size\_t j = 0; j < Columns; j++)

{

// 0 or 1 random generation

yourMatrix[i][j] = rand() % 2;

}

}

}

void PrintMatrix(Matrix yourMatrix, unsigned int Rows, unsigned int Columns)

{

cout << endl;

cout << "PRINTING MATRIX" << endl;

for (size\_t i = 0; i < Rows; i++)

{

for (size\_t j = 0; j < Columns; j++)

{

cout << yourMatrix[i][j] << " ";

}

cout << endl;

}

}

int main()

{

unsigned int matrixRows, matrixColumns, generations, distanceBetweenCells;

int neighbours = 0;

cout << "matrix columns: ";

cin >> matrixColumns;

cout << "matrix rows: ";

cin >> matrixRows;

cout << "Number of generations: ";

cin >> generations;

cout << "Distance between cells: ";

cin >> distanceBetweenCells;

Matrix matrix = InstantiateMatrix(matrixRows, matrixColumns);

RandomMatrixGeneration(matrix, matrixRows, matrixColumns);

//ManualMatrixGeneration(matrix, matrixRows, matrixColumns);

cout << endl;

cout << "\*\*\*Original Matrix\*\*\*";

PrintMatrix(matrix, matrixRows, matrixColumns);

while (generations != 0)

{

for (size\_t i = 0; i < matrixRows; i = i + distanceBetweenCells)

{

for (size\_t j = 0; j < matrixColumns; j = j + distanceBetweenCells)

{

neighbours = CountLivingCells(matrix, 1, i, j, matrixRows, matrixColumns);

if (IsALivingCell(matrix, i, j))

{

if (neighbours <= 1)

{

matrix[i][j] = 0;

cout << "[" << i << "," << j << "] = died" << endl;

}

else if (neighbours >= 4)

{

matrix[i][j] = 0;

cout << "[" << i << "," << j << "] = died" << endl;

}

}

else if (IsADeadCell(matrix, i, j))

{

if (neighbours == 3)

{

matrix[i][j] = 1;

cout << "[" << i << "," << j << "] = returned to life" << endl;

}

}

}

}

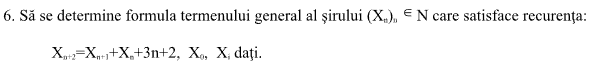
generations--;

PrintMatrix(matrix, matrixRows, matrixColumns);

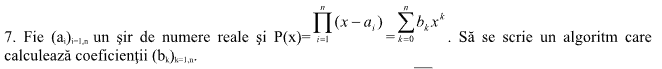
}

}

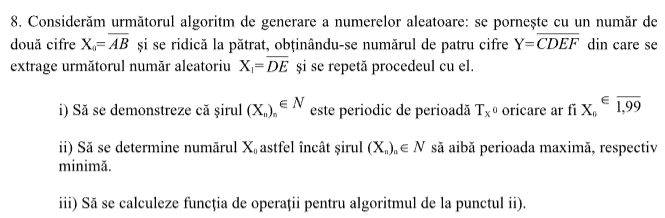
# 6. Formula termenului General



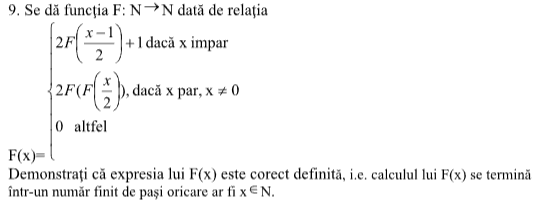
# 7. Coeficientii



# 8. Generare Numere Aleatoare



# 9. Functie



int functieRandom(int x)

{

if (x % 2 == 1)

{

return 2 \* functieRandom((x - 1) / 2);

}

if (x % 2 == 0 && x != 0)

{

return 2 \* functieRandom(functieRandom(x / 2));

}

else

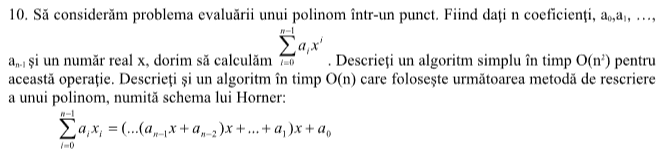
{

return 0;

}

}

# 10. Polinom



// Schelet clasa polinom cu functia calcularii unui polinom intr-un punct

class Polinom

{

private:

int grad;

vector <float> coeficient;

public:

Polinom(int);

~Polinom()

{

}

void citire();

void afisare();

void dealocare();

Polinom operator+(Polinom);

float calculPunct(float);

};

// Functie de calcul cu un algoritm in timp de O(n)

float Polinom::calculPunct(float x)

{

float s = 0;

for (int i = 0; i < grad; i++)

{

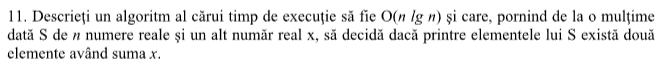
s = s + coeficient[i] \* pow(x, (float)i);

}

return s;

}

# 11. Suma X



// O(n log n)

bool findSum(int arr[], unsigned int size ,int x)

{

unsigned index{ 0 };

bool foundSum = false;

int currentNum{0};

while (foundSum == false)

{

if (index == size - 1)

{

break;

}

else

{

currentNum = arr[index++];

}

for (size\_t i = 1; i < size; i++)

{

if (currentNum + arr[i] == x)

{

std::cout << currentNum << " + " << arr[i] << " = " << x << std::endl;

foundSum = true;

return foundSum;

}

else

{

foundSum = false;

}

}

}

return foundSum;

}

int main()

{

int a[] = { 12,3,45,12,678,3 };

unsigned int size = sizeof(a) / sizeof(a[0]);

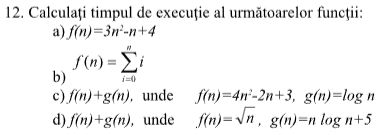
int x = 25;

std::cout << "Is there a sum of A equal to X? : " << findSum(a,size,x);

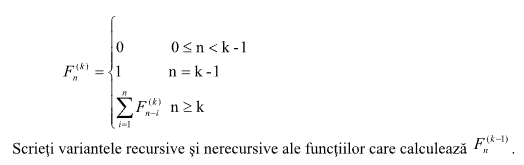
return 0;

}

# 12. Calculul Timpului Executiei



# 13. Fibonacci



int fibonacci\_iterativ(int n)

{

int a = 0, b = 1, c, i;

if (n == 0)

return a;

for (i = 2; i <= n; i++)

{

c = a + b;

a = b;

b = c;

}

return b;

}

// fibonacci recursiv - normal

int fibonacci\_recursiv(int n)

{

if (n <= 1)

return n;

return fibonacci\_recursiv(n - 1) + fibonacci\_recursiv(n - 2);

}

// fibonnaci pentru F(n - 1, k) ??

int fibonacci\_weird(int n, int k)

{

if (n >= 0 && n < k - 1)

{

return 0;

}

else if (n == k - 1)

{

return 1;

}

else

{

return fibonacci\_weird(n - 1,k) + fibonacci\_weird(n - 2,k);

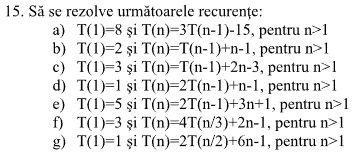
}

}

# 14. Metodele coardei



# 15. Recurente



# 16. Factorial recursiv si iterativ



// Recursiv

int Factorial(int value)

{

if (value == 1)

{

return 1;

}

else

{

return value \* Factorial(value - 1);

}

}

// Iterativ

int factorial(int value)

{

int result{ 1 };

for (size\_t i = 1; i <= value; i++)

{

result = result \* i;

}

return result;

}