

#include <Windows.h>

#include <iostream>

using namespace std;

double recursion1(int n, int x)

{

// выходим из функции когда дошли до конца

if (n == x)

{

return sqrt(1 + n + n);

}

// Иначе продалжаем вызывать ее с увеличенным параметром

return sqrt(1 + (n + x) \* recursion1(n, x + 1));

}

// Функция для вычисления бинома

long long recursion2(int m, int n) {

// Базовые случаи выхода из функции

if (m == 0 || m == n) {

return 1;

}

else {

// Рекурсивное вычисление по формуле Бинома Ньютона

return recursion2(m, n - 1) + recursion2(m - 1, n - 1);

}

}

int main() {

SetConsoleOutputCP(1251);

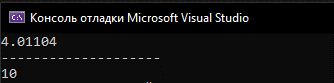
cout << recursion1(3, 1);

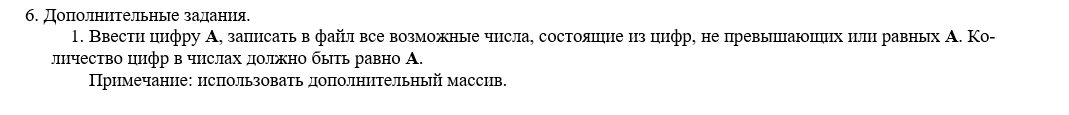
cout << "\n--------------------\n";

cout << recursion2(3, 5);

return 0;

}





#include <iostream>

#include <fstream>

#include <vector>

#include <Windows.h>

// Функцыя для запісу ліку ў файл

void write(std::ofstream& outputFile, const std::vector<int>& number) {

for (int digit : number) {

outputFile << digit;

}

outputFile << std::endl;

}

// Рэкурсіўная функцыя для генерацыі ўсіх лікаў

void generateNumbers(int A, std::vector<int>& currentNumber, int currentDigit, std::ofstream& outputFile) {

if (currentDigit == A) {

// Калі мы дасягнулі патрэбнай колькасці лічбаў, запісваем лік у файл

write(outputFile, currentNumber);

return;

}

for (int digit = 1; digit <= A; ++digit) {

// Дадаем лічбу да бягучага ліку

currentNumber.push\_back(digit);

// Рэкурсіўны выклік для наступнай лічбы

generateNumbers(A, currentNumber, currentDigit + 1, outputFile);

// Прыбіраем апошнюю дабаўленую лічбу перад пераходам да наступнай ітэрацыі

currentNumber.pop\_back();

}

}

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int A;

std::cout << "Увядзіце лічбу A: ";

std::cin >> A;

if (A < 1 || A > 9) {

std::cout << "Некарэктнае значэнне A. Увядзіце лічбу ад 1 да 9." << std::endl;

return 1;

}

// Адкрываем файл для запісу

std::ofstream outputFile("output.txt");

if (!outputFile.is\_open()) {

std::cerr << "Немагчыма адкрыць файл для запісу." << std::endl;

return 1;

}

std::vector<int> currentNumber; // Вектар для захоўвання бягучага ліку

generateNumbers(A, currentNumber, 0, outputFile);

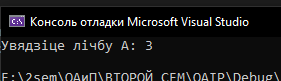
// Закрываем файл пасля завяршэння запісу

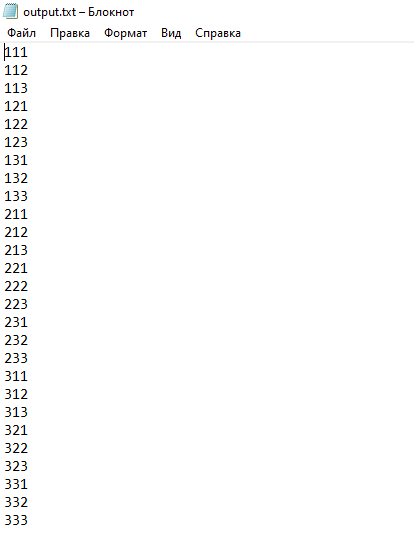
outputFile.close();

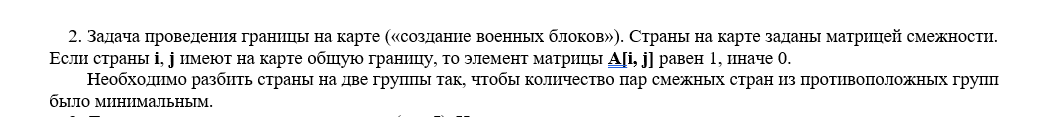
std::cout << "Лікі паспяхова запісаны ў файл 'output.txt'." << std::endl;

return 0;

}







Пачытаўшы ў інтэрнэце, здаецца, гэтая задача ў сутнасці з'яўляецца задачай аб двухдольным графе, дзе краіны ўяўляюць вяршыні графа, а рэбры паміж краінамі - наяўнасць агульнай мяжы. Задача заключаецца ў разбіцці вяршыняў (краін) на дзве групы так, каб мінімізаваць колькасць рэбраў (межаў), якія злучаюць вяршыні з розных груп.

#include <iostream>

#include <vector>

#include <Windows.h>

// Функцыя для вызначэння двухдольнасці графа з выкарыстаннем рэкурсіі

// Параметры:

// - adjacencyMatrix: матрыца сумежнасці графа

// - current: бягучая вяршыня, якую мы правяраем

// - colors: вектар колераў вяршыняў (0 або 1, -1 азначае неафарбаваную вяршыню)

// - color: колер бягучай вяршыні

// Вяртае true, калі граф двудолен, і false у адваротным выпадку

bool isBipartiteRecursive(const std::vector<std::vector<int>>& adjacencyMatrix, int current, std::vector<int>& colors, int color) {

colors[current] = color; // Афарбоўваем бягучую вяршыню

// Перабіраем суседнія вяршыні

for (int neighbor = 0; neighbor < adjacencyMatrix.size(); ++neighbor) {

if (adjacencyMatrix[current][neighbor] == 1) {

// Калі суседняя вяршыня не афарбаваная, выклікаем рэкурсіўна з іншым колерам

if (colors[neighbor] == -1 && !isBipartiteRecursive(adjacencyMatrix, neighbor, colors, 1 - color)) {

return false; // Граф не двудольны

}

// Калі суседняя вяршыня афарбавана ў той жа колер, граф не двудолен

else if (colors[neighbor] == color) {

return false;

}

}

}

return true; // Усё дакладна афарбавана, граф двудолен

}

// Функцыя для вызначэння двухдольнасці графа з выкарыстаннем рэкурсіі

// Параметры:

// - adjacencyMatrix: матрыца сумежнасці графа

// - start: вяршыня, з якой пачынаецца праверка

// - colors: вектар колераў вяршыняў (0 або 1, -1 азначае неафарбаваную вяршыню)

// Вяртае true, калі граф двудолен, і false у адваротным выпадку

bool isBipartite(const std::vector<std::vector<int>>& adjacencyMatrix, int start, std::vector<int>& colors) {

return isBipartiteRecursive(adjacencyMatrix, start, colors, 0);

}

// Функцыя для вываду краін з групы

// Параметры:

// - colors: вектар кветак вяршыняў

// - group: нумар групы

void printGroup(const std::vector<int>& colors, int group) {

std::cout << "Група " << group << ": ";

for (int i = 0; i < colors.size(); ++i) {

if (colors[i] == group) {

std::cout << i + 1 << " "; // Краіны нумаруюцца з 1

}

}

std::cout << std::endl;

}

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

// Прыклад матрыцы сумежнасці для краін

std::vector<std::vector<int>> adjacencyMatrix = {

{0, 1, 1, 0},

{1, 0, 0, 1},

{1, 0, 0, 1},

{0, 1, 1, 0}

};

int numCountries = adjacencyMatrix.size();

std::vector<int> colors(numCountries, -1); // -1 азначае, што вяршыня яшчэ не афарбаваная

// Правяраем двудольнасць графа, пачынаючы з кожнай вяршыні

for (int i = 0; i < numCountries; ++i) {

if (colors[i] == -1 && !isBipartite(adjacencyMatrix, i, colors)) {

std::cout << "Мяжы нельга разбіць на дзве групы з мінімальнай колькасцю сумежных пар" << std::endl;

return 1;

}

}

std::cout << "Мяжы можна разбіць на дзве групы з мінімальнай колькасцю сумежных пар" << std::endl;

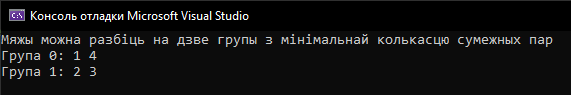
// Выводзім краіны ў адпаведныя групы

printGroup(colors, 0);

printGroup(colors, 1);

return 0;

}



#include <iostream>

#include <vector>

// Функцыя для абмену значэннямі паміж двума зменнымі, проста для зручнасці

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

// Рэкурсіўная функцыя для генерацыі ўсіх перастановак

void generatePermutations(std::vector<int>& numbers, int startIndex, int n) {

// Калі дасягнуты канец масіва, выводзім бягучую перастаноўку

if (startIndex == n - 1) {

for (int i = 0; i < n; ++i) {

std::cout << numbers[i];

}

std::cout << std::endl;

return;

}

// генеруецца ўсе магчымыя перастаноўкі, змяняючы бягучы элемент з кожным з наступных

for (int i = startIndex; i < n; ++i) {

// Змяняем бягучы элемент з кожным з наступных

swap(numbers[startIndex], numbers[i]);

// Рэкурсіўна генеруем перастаноўкі для пакінутай часткі масіва

generatePermutations(numbers, startIndex + 1, n);

// Вяртаем масіў у зыходны стан для далейшых перастановак

swap(numbers[startIndex], numbers[i]);

}

}

int main() {

// Зыходныя натуральныя лікі

std::vector<int> numbers = { 1, 2, 3 };

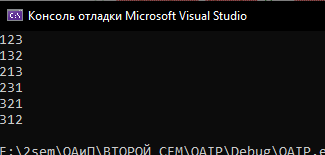
int n = numbers.size();

// Генераваны і выводзім ўсе перастаноўкі

generatePermutations(numbers, 0, n);

return 0;

}



Для 5 занадта шмат