

МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

***«МИРЭА – Российский технологический университет»***

**РТУ МИРЭА**



Отчет по выполнению практического задания № 7

**Тема:**

«Рекурсивные алгоритмы и их реализация»

Дисциплина: «Структуры и алгоритмы обработки данных»

| Выполнил студент группы: ИКБО-73-23 |  |  | Полянский. М. Д |
| --- | --- | --- | --- |
| Принял ассистент |  |  | Золотухин. С. А |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 3](#_30j0zll)

[2.1 Формулировка задачи 3](#_1fob9te)

[2.2 Итерационный алгоритм 3](#_2et92p0)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 6](#_17dp8vu)

[2.5 Объединение программ 9](#_1ci93xb)

[3 ЗАДАНИЕ №2 12](#_2bn6wsx)

[3.1 Формулировка задачи 12](#_qsh70q)

[3.2 Рекурсивная функция 12](#_u290pxmzmczz)

[4 ВЫВОДЫ 14](#_ihv636)

[5 ЛИТЕРАТУРА 15](#_32hioqz)

# **1 ЦЕЛЬ**

Получить знания и практические навыки по разработке и реализации рекурсивных процессов.

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи**

Разработать и протестировать рекурсивные функции в соответствии с задачами варианта

Требования к выполнению первой задачи варианта:

• приведите итерационный алгоритм решения задачи

• реализуйте алгоритм в виде функции и отладьте его

• определите теоретическую сложность алгоритма

• опишите рекуррентную зависимость в решении задачи

• реализуйте и отладьте рекурсивную функцию решения задачи

• определите глубину рекурсии, изменяя исходные данные

• определите сложность рекурсивного алгоритма, используя метод подстановки и дерево рекурсии

• приведите для одного из значений схему рекурсивных вызовов

• разработайте программу, демонстрирующую выполнение обеих функций и покажите результаты тестирования.

Задание: Прохождение лабиринта

## **2.2 Итерационный алгоритм**

Начинаем с начальной точки лабиринта, координаты которой обозначаются как (0, 0).

Помечаем текущую позицию как посещенную и переходим к следующей клетке, используя один из возможных ходов: вправо, вниз, влево или вверх.

Повторяем шаг 2 до тех пор, пока не достигнем конечной точки лабиринта или не окажется, что путь невозможен. Если мы достигли конечной точки, объявляем, что путь найден, и выводим найденный путь.

Для каждого шага алгоритм проверяет, является ли следующая клетка проходимой и не посещалась ли она ранее. Если возможные ходы заблокированы или не приводят к достижению конечной точки, алгоритм завершается с сообщением о том, что путь не найден.

Итерационный алгоритм последовательно двигается по клеткам лабиринта, чтобы найти путь, и в случае успеха выводит координаты точек, через которые проходит путь.

Реализация данного алгоритма представленная в функции solveMazeIterative (блок кода 1)

| *// Прохождение лабиринта с использованием итерации* bool solveMazeIterative(int N, char maze[][4]) {  if (N == 0) return false;   int start\_x = 0, start\_y = 0;  int end\_x = N - 1, end\_y = N - 1;   int x = start\_x, y = start\_y;  while (x != end\_x || y != end\_y) {  *// Если достигли конечной точки, возвращаем успех*  if (x == end\_x && y == end\_y) {  cout << "Путь найден!" << endl;  return true;  }   maze[x][y] = '0'; *// Помечаем текущую клетку как посещенную*   *// Проверяем соседние клетки вверх, вниз, влево и вправо*  if (isSafe(x + 1, y, N, maze)) x++; *// Вниз*  else if (isSafe(x, y + 1, N, maze)) y++; *// Вправо*  else if (isSafe(x - 1, y, N, maze)) x--; *// Вверх*  else if (isSafe(x, y - 1, N, maze)) y--; *// Влево*  else {  cout << "Путь не найден!" << endl;  return false;  }  }   *// Выводим найденный путь*  cout << "Way: ";  for (int i = 0; i <= end\_x; ++i) {  for (int j = 0; j <= end\_y; ++j) {  if (maze[i][j] == '0') cout << "(" << i << "," << j << ") ";  }  }  cout << "(" << end\_x << "," << end\_y << ")" << endl;   return true; } |
| --- |

Блок схема 1 - Реализация итерационного алгоритма для задачи 1

Проведем отладку данной функции на значениях '1', '0', '1', '1', '1', '1', '1', '0', '0', '1', '0', '1', '1', '1', '1', '1' и продемонстрируем проделанный путь(рис. 1).



Рисунок 1 - Тестирование алгоритма

Теоретическая сложность итеративного алгоритма прохождения лабиринта зависит от размеров лабиринта и его формы. Предположим, что лабиринт имеет размерность N х N, где N - количество клеток в строке и столбце.

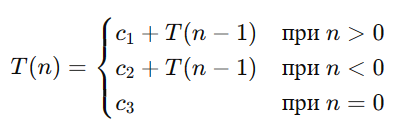
В наихудшем случае алгоритм может посетить каждую клетку лабиринта один раз. Таким образом, общее количество операций в итеративном алгоритме пропорционально количеству клеток в лабиринте, то есть O(N2).

В реализации данного алгоритма используется цикл while, который выполняется, пока не будет найден путь или не закончится клеток для исследования. В худшем случае (когда путь проходит через каждую клетку лабиринта), алгоритм посещает каждую клетку один раз, что приводит к временной сложности O(N2).

Таким образом, теоретическая временная сложность итеративного алгоритма прохождения лабиринта составляет O(N2), где N - размерность лабиринта.

## **2.3 Рекуррентная зависимость**

Рекуррентная зависимость в решении задачи прохождения лабиринта может быть описана следующим образом:



Где:

T(n) - время выполнения функции для лабиринта размером n×n.

𝑐1 и 𝑐2- константы времени на выполнение базовых операций внутри рекурсивной функции для случаев n>0 и n<0 соответственно.

𝑐3 - константное время на выполнение базовых операций при выходе из рекурсии при n=0.

Рекурсивная функция вызывает себя для каждой соседней проходимой клетки, чтобы найти путь от нее до конечной. Таким образом, время выполнения функции для лабиринта размером n×n зависит от времени выполнения функций для соседних клеток и константного времени на выполнение базовых операций.

## **2.4 Рекурсивная функция**

Реализуем рекурсивную функцию для данной задачи (блок кода 2).

| *// Рекурсивная функция для прохождения лабиринта* bool solveMazeRecursive(int maze[][4], int x, int y, int N, pair<int, int>\* path, int& path\_index) {  *// Если достигли конечной точки, возвращаем успех*  if (x == N - 1 && y == N - 1) {  path[path\_index++] = { x, y };  return true;  }   *// Помечаем текущую клетку как посещенную*  maze[x][y] = 0;  path[path\_index++] = { x, y };   *// Проверяем соседние клетки вверх, вниз, влево и вправо*  if (isSafe(maze, x + 1, y, N) && solveMazeRecursive(maze, x + 1, y, N, path, path\_index)) return true; *// Вправо*  if (isSafe(maze, x - 1, y, N) && solveMazeRecursive(maze, x - 1, y, N, path, path\_index)) return true; *// Влево*  if (isSafe(maze, x, y + 1, N) && solveMazeRecursive(maze, x, y + 1, N, path, path\_index)) return true; *// Вниз*  if (isSafe(maze, x, y - 1, N) && solveMazeRecursive(maze, x, y - 1, N, path, path\_index)) return true; *// Вверх*   *// Если ни один из соседей не привел к решению, удаляем текущую клетку из пути*  path\_index--;  return false; } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем отладку данной функции на значениях '1', '0', '1', '1', '1', '1', '1', '0', '0', '1', '0', '1', '1', '1', '1', '1' и продемонстрируем проделанный путь(рис. 2).



Рисунок 2 - Тестирование адаптированной программы

Глубина рекурсии - это количество рекурсивных вызовов, которые произошли во время выполнения программы. Глубина рекурсии зависит от размера лабиринта и структуры самого лабиринта. В худшем случае, если лабиринт представляет собой поле, где каждая ячейка является проходимой и путь продолжается до самого конца, глубина рекурсии будет равна длине пути от начальной до конечной точки. В данном коде глубина рекурсии зависит от сложности лабиринта и его размера. Для данного конкретного лабиринта размером 4x4 и простого пути от начальной до конечной точки, глубина рекурсии будет не более 16 (4x4), так как каждая ячейка в пути добавляется в массив path, и path\_index увеличивается с каждым рекурсивным вызовом, пока не будет достигнута конечная точка.

Определим сложность рекурсивной функции.

Если рассматривать дерево рекурсии, то на вершине дерева находится исходный вызов функции с начальными координатами (0, 0). Каждый узел в дереве представляет собой один рекурсивный вызов функции с новыми координатами. Узлы дерева соответствуют всем возможным путям в лабиринте.

Метод подстановки. Предположим, что функция solveMazeRecursive вызывается T(N) раз, где N - размер лабиринта. Для каждого вызова функции выполняются четыре рекурсивных вызова, один для каждого соседнего направления. При каждом рекурсивном вызове размер лабиринта уменьшается на 1 (так как мы перемещаемся на одну клетку). Базовый случай достигается, когда мы достигаем конечной точки, что происходит после N\*N шагов.

Сложность рекурсивного алгоритма. На каждом уровне дерева рекурсии происходит 4 рекурсивных вызова. Глубина дерева рекурсии равна длине пути от начальной до конечной точки, то есть N\*N. Таким образом, общее число узлов в дереве рекурсии составляет O(4N).

Изобразим схему рекурсии для данной задачи так (рисунок 3)

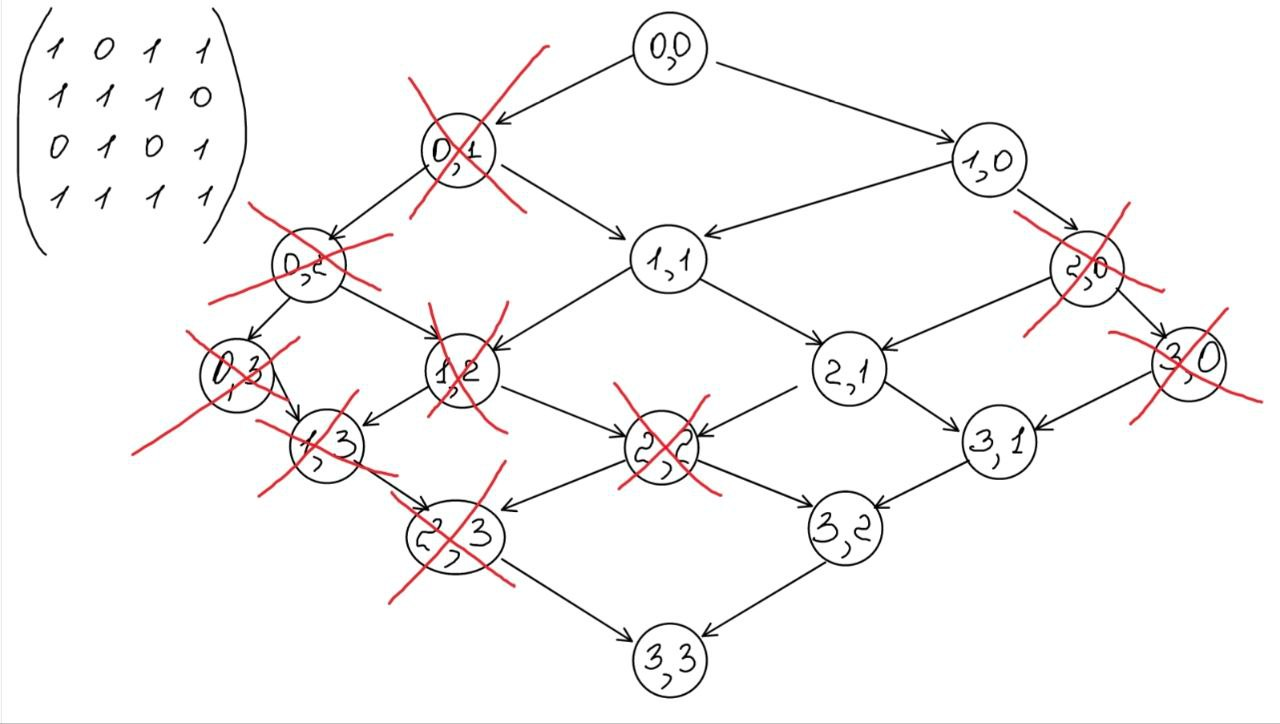


Рисунок 3 - Изображение схемы рекурсивных вызовов

## **2.5 Объединение программ**

Объединим программы итерационного алгоритма и рекурсивной функции в блок коде 3 с выполнением обоих алгоритмов на заданном числе. Продемонстрируем результаты работы программы на рисунке 4.

| #include <iostream> using namespace std;  *// Функция для проверки, может ли точка быть частью пути в лабиринте (рекурсивный метод)* bool isSafeRecursive(int x, int y, int N, char maze[][4]) {  return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == '1'); }  *// Рекурсивная функция для прохождения лабиринта* bool solveMazeRecursive(int x, int y, int N, char maze[][4], pair<int, int>\* path, int& path\_index) {  *// Если достигли конечной точки, возвращаем успех*  if (x == N - 1 && y == N - 1) {  path[path\_index++] = { x, y };  return true;  }   *// Помечаем текущую клетку как посещенную*  maze[x][y] = '0';  path[path\_index++] = { x, y };   *// Проверяем соседние клетки вверх, вниз, влево и вправо*  if (isSafeRecursive(x + 1, y, N, maze) && solveMazeRecursive(x + 1, y, N, maze, path, path\_index)) return true; *// Вниз*  if (isSafeRecursive(x, y + 1, N, maze) && solveMazeRecursive(x, y + 1, N, maze, path, path\_index)) return true; *// Вправо*  if (isSafeRecursive(x - 1, y, N, maze) && solveMazeRecursive(x - 1, y, N, maze, path, path\_index)) return true; *// Вверх*  if (isSafeRecursive(x, y - 1, N, maze) && solveMazeRecursive(x, y - 1, N, maze, path, path\_index)) return true; *// Влево*   *// Если ни один из соседей не привел к решению, удаляем текущую клетку из пути*  path\_index--;  return false; }  *// Функция для проверки, может ли точка быть частью пути в лабиринте (итеративный метод)* bool isSafeIterative(int x, int y, int N, char maze[][4]) {  return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == '1'); }  *// Прохождение лабиринта с использованием итерации* bool solveMazeIterative(int n, char m[][4]) {  if (n == 0) return false;   int start\_x = 0, start\_y = 0;  int end\_x = n - 1, end\_y = n - 1;   int x = start\_x, y = start\_y;  while (x != end\_x || y != end\_y) {  *// Если достигли конечной точки, возвращаем успех*  if (x == end\_x && y == end\_y) {  cout << "The path has been found!" << endl;  return true;  }   m[x][y] = '0'; *// Помечаем текущую клетку как посещенную*   *// Проверяем соседние клетки вверх, вниз, влево и вправо*  if (isSafeIterative(x + 1, y, n, m)) x++; *// Вниз*  else if (isSafeIterative(x, y + 1, n, m)) y++; *// Вправо*  else if (isSafeIterative(x - 1, y, n, m)) x--; *// Вверх*  else if (isSafeIterative(x, y - 1, n, m)) y--; *// Влево*  else {  cout << "The path has not been found" << endl;  return false;  }  }   *// Выводим найденный путь*  cout << "The path has been found:";  for (int i = 0; i <= end\_x; ++i) {  for (int j = 0; j <= end\_y; ++j) {  if (m[i][j] == '0') cout << "(" << i << "," << j << ") ";  }  }  cout << "(" << end\_x << "," << end\_y << ")" << endl;   return true; }  int main() {  *// Пример лабиринта: '1' - проходимая ячейка, '0' - непроходимая ячейка*  char maze[4][4] = {  {'1', '0', '1', '1'},  {'1', '1', '1', '0'},  {'0', '1', '0', '1'},  {'1', '1', '1', '1'}  };   int N = 4; *// Размер лабиринта*  pair<int, int> path[16]; *// Для хранения пути*   cout << "Recursive Maze Solver:" << endl;  int path\_index = 0;  if (solveMazeRecursive(0, 0, N, maze, path, path\_index)) {  cout << "Path found:" << endl;  for (int i = 0; i < path\_index; ++i) {  cout << "(" << path[i].first << "," << path[i].second << ") ";  }  cout << endl;  } else {  cout << "Path not found" << endl;  }   cout << endl;  char m[4][4] = {  {'1', '0', '1', '1'},  {'1', '1', '1', '0'},  {'0', '1', '0', '1'},  {'1', '1', '1', '1'}  };   int n = 4; *// Размер лабиринта*   cout << "Iterative Maze Solver:" << endl;  solveMazeIterative(n, m);   return 0; } |
| --- |

Блок кода 3 - Объединение программ

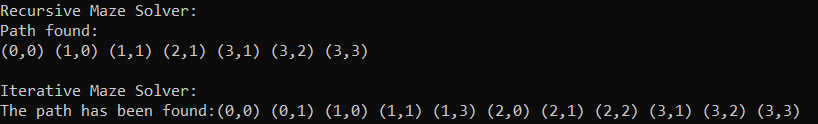


Рисунок 4 - Тестирование программы

# **3 ЗАДАНИЕ №2**

## **3.1 Формулировка задачи**

Требования к выполнению второй задачи варианта:

• рекурсивную функцию для обработки списковой структуры согласно варианту. Информационная часть узла – простого типа – целого;

• для создания списка может быть разработана простая или рекурсивная функция по желанию (в тех вариантах, где не требуется рекурсивное создание списка);

• определите глубину рекурсии

• определите теоретическую сложность алгоритма

• разработайте программу, демонстрирующую работу функций и покажите результаты тестов.

Задание: Определить симметрично ли число, цифры которого последовательно записаны в узлах двунаправленного списка.

## **3.2 Рекурсивная функция**

Реализуем задачу с помощью рекурсивной функции(блок кода 3).

| #include <iostream> using namespace std;  struct Node {  int data;  Node\* prev;  Node\* next;  Node(int d) : data(d), prev(nullptr), next(nullptr) {} };  Node\* createLinkedList(int\* numbers, int size, int index = 0) {  if (index == size)  return nullptr;  Node\* newNode = new Node(numbers[index]);  newNode->next = createLinkedList(numbers, size, index + 1);  if (newNode->next)  newNode->next->prev = newNode;  return newNode; } bool isSymmetric(Node\* left, Node\* right) {  if (!left || !right)  return true;  if (left->data != right->data)  return false;  return isSymmetric(left->next, right->prev); }  bool isListSymmetric(Node\* head) {  Node\* current = head;  while (current->next)  current = current->next;  return isSymmetric(head, current); }  void printList(Node\* head) {  Node\* current = head;  while (current) {  cout << current->data << " ";  current = current->next;  }  cout << std::endl; }  int main() {  int numbers1[] = { 1, 2, 3, 4, 3, 2, 1 };  Node\* list1 = createLinkedList(numbers1, 7);  cout << "List1: ";  printList(list1);  cout << "Is list1 symmetric: " << boolalpha << isListSymmetric(list1) << endl;   int numbers2[] = { 1, 2, 3, 4, 5, 6, 7 };  Node\* list2 = createLinkedList(numbers2, 7);  cout << "List2: ";  printList(list2);  cout << "Is list2 symmetric: " << boolalpha << isListSymmetric(list2) << endl;   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

В данном случае глубина рекурсии определяется как глубина списка, так как каждый рекурсивный вызов функции приводит к движению вдоль списка на один уровень глубже. Таким образом, глубина рекурсии равна длине списка.

Сложность проверки симметричности списка составляет O(n), где n - это длина списка, так как в худшем случае мы должны пройти по всему списку один раз для проверки симметричности.

Продемонстрируем результаты работы программы на рисунке 5.

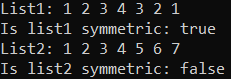


Рисунок 5 - Тестирование программы

# **4 ВЫВОДЫ**

В процессе выполнения данной практической работы были разработаны программы на языке C++, демонстрирующие работу рекурсивных функций.

Эта практическая работа позволила ознакомиться с разработкой и реализаций рекурсивных процессов на языке программирования C++ и приобрести практические навыки работы с данными процессами. Полученный опыт будет полезен при решении задач, связанных с решением задач, где структура данных или алгоритмы могут быть естественно выражены в рекурсивной форме.

# **5 ЛИТЕРАТУРА**

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).

12. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).