|  |
| --- |
| 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: Чахуриди К.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи (Вариант 12, в списке 28) 4](#_1fob9te)

[2.2 Итерационный алгоритм 4](#_2et92p0)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 6](#_17dp8vu)

[2.5 Программа с итерационным алгоритмом и рекурсивной функцией 9](#_1ci93xb)

[3 ЗАДАНИЕ №2 13](#_2bn6wsx)

[3.1 Формулировка задачи 13](#_qsh70q)

[3.2 Рекурсивная функция 13](#_u290pxmzmczz)

[4 ВЫВОДЫ 16](#_ihv636)

[5 ЛИТЕРАТУРА 17](#_32hioqz)

# **1 ЦЕЛЬ**

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

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи (Вариант 12, в списке 28)**

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: Сортировка массива по возрастанию

## **2.2 Итерационный алгоритм**

Повторять для каждого элемента массива:

Повторять для всех пар элементов:

Если текущий элемент больше следующего, поменять их местами.

Если за весь проход по массиву ни одного обмена не произошло, завершить сортировку.

Реализуем алгоритм в блок кода 1.

| #include <iostream> using namespace std;  // Итерационная функция сортировки (пузырьковая сортировка) void bubbleSort(int arr[], int n) {  for (int i = 0; i < n - 1; ++i) {  bool swapped = false;  for (int j = 0; j < n - i - 1; ++j) {  if (arr[j] > arr[j + 1]) {  swap(arr[j], arr[j + 1]);  swapped = true;  }  }  // Если элементы не были поменяны местами, массив уже отсортирован  if (!swapped)  break;  } }  void printArray(int arr[], int size) {  for (int i = 0; i < size; i++) {  cout << arr[i] << " ";  }  cout << endl; }  int main() {  int arr[] = {64, 34, 25, 12, 22, 11, 90};  int n = sizeof(arr)/sizeof(arr[0]);  cout << "Изначальный массив: ";  printArray(arr, n);   bubbleSort(arr, n);  cout << "Отсортированный массив (итеративно): ";  printArray(arr, n);  return 0; } |
| --- |

Блок код 1 - Реализация итерационного алгоритма для задачи 1

Проведем отладку данной функции при значениях 64, 34, 25, 12, 22, 11, 90 (рис. 1).



Рисунок 1 - Тестирование алгоритма

**Худший случай:** O(n2)

**Средний случай:** O(n2)

**Лучший случай:** O(n)

**Пространственная сложность:** O(1)

## **2.3 Рекуррентная зависимость**

T(n)=T(k)+T(n−k−1)+O(n), где:

* k — количество элементов в первой подчасти после разделения.
* T(k) — время, затраченное на сортировку первой подчасти.
* T(n−k−1) — время, затраченное на сортировку второй подчасти.
* O(n) — время, затраченное на разделение массива на подчасти (так как разделение требует одного прохода по массиву из nnn элементов).

## **2.4 Рекурсивная функция**

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

| #include <iostream> using namespace std;  // Функция для разделения массива и возвращения индекса опорного элемента int partition(int arr[], int low, int high) {  int pivot = arr[high]; // выбираем последний элемент как опорный  int i = low - 1; // индекс меньшего элемента   for (int j = low; j < high; j++) {  if (arr[j] <= pivot) {  i++;  swap(arr[i], arr[j]);  }  }  swap(arr[i + 1], arr[high]);  return i + 1; }  // Рекурсивная функция быстрой сортировки void quickSort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);   // Сортируем элементы до и после разделения  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  } }  void printArray(int arr[], int size) {  for (int i = 0; i < size; i++) {  cout << arr[i] << " ";  }  cout << endl; }  int main() {  int arr[] = {64, 34, 25, 12, 22, 11, 90};  int n = sizeof(arr)/sizeof(arr[0]);  cout << "Изначальный массив: ";  printArray(arr, n);   quickSort(arr, 0, n - 1);  cout << "Отсортированный массив (рекурсивно): ";  printArray(arr, n);  return 0; } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем отладку данной функции на тех же значениях, что и при отладке итерационного алгоритма (рис. 2).

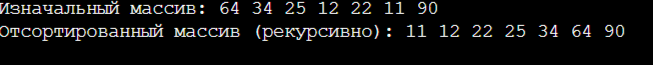


Рисунок 2 - Тестирование рекурсивной программы

Глубина рекурсии в быстрой сортировке зависит от опорного элемента и его позиции после каждого разделения. В худшем случае (если массив уже отсортирован или наоборот, в обратном порядке), глубина рекурсии будет максимальной и равна n, где n — длина массива. В среднем случае глубина рекурсии оценивается как log⁡n.

Быстрая сортировка (QuickSort) имеет среднюю временную сложность O(nlog⁡n). Однако в худшем случае, когда на каждом шаге выбирается худший опорный элемент, сложность может быть O(n2).

Метод подстановки

Для определения временной сложности используем рекуррентное соотношение: T(n)=T(k)+T(n−k−1)+O(n) где k — количество элементов в одной из частей после разбиения.

В среднем случае предполагаем, что массив делится на две равные части: T(n)=2T(n/2)+O(n)

Используем метод подстановки:

1. Подставляем в рекуррентное соотношение: T(n)=2T(n/2)+cn
2. Разворачиваем несколько уровней рекурсии: T(n)=2(2T(n/4)+cn2)+cn=4T(n/4)+2cn

T(n)=4(2T(n/8)+cn4)+2cn=8T(n/8)+3cn

1. Продолжаем разворачивать до конца: T(n)=2kT(n/2k)+kcn, где k=log⁡n

T(n)=nT(1)+cnlog⁡n=O(nlog⁡n)

Таким образом, средняя временная сложность быстрой сортировки O(nlog⁡n).

Схема рекурсивных вызовов

Каждый вызов функции quickSort будет помечен как Q(low, high), где low и high - границы подмассива.

| Q(0, 6) // Сортируем весь массив |\_ Q(0, 5) // Сортируем левый подмассив {64, 34, 25, 12, 22, 11}  |\_ Q(0, 0) // Левый подмассив одного элемента {11}  |\_ Q(1, 5) // Сортируем правый подмассив {64, 34, 25, 12, 22}  |\_ Q(1, 4) // Сортируем левый подмассив {22, 34, 25, 12}  |\_ Q(1, 0) // Левый подмассив пустой  |\_ Q(2, 4) // Сортируем правый подмассив {34, 25, 12}  |\_ Q(2, 3) // Сортируем левый подмассив {12, 25}  |\_ Q(2, 1) // Левый подмассив пустой  |\_ Q(3, 3) // Правый подмассив {25}  |\_ Q(5, 4) // Правый подмассив пустой  |\_ Q(6, 5) // Правый подмассив пустой |\_ Q(7, 6) // Правый подмассив пустой |
| --- |

Рисунок 3 - Вызов рекурсии для массива значений

Это дерево рекурсии демонстрирует последовательные вызовы рекурсивной функции.

## **2.5 Программа с итерационным алгоритмом и рекурсивной функцией**

Реализуем программу с двумя алгоритмами в блок коде 3. Продемонстрируем результаты работы программы на рисунке 4.

| #include <iostream> using namespace std;  // Функция для печати массива void printArray(int arr[], int size) {  for (int i = 0; i < size; i++) {  cout << arr[i] << " ";  }  cout << endl; }  // Итерационная функция сортировки (пузырьковая сортировка) void bubbleSort(int arr[], int n) {  for (int i = 0; i < n - 1; ++i) {  bool swapped = false;  for (int j = 0; j < n - i - 1; ++j) {  if (arr[j] > arr[j + 1]) {  swap(arr[j], arr[j + 1]);  swapped = true;  }  }  // Если элементы не были поменяны местами, массив уже отсортирован  if (!swapped)  break;  } }  // Функция для разделения массива и возвращения индекса опорного элемента int partition(int arr[], int low, int high) {  int pivot = arr[high]; // выбираем последний элемент как опорный  int i = low - 1; // индекс меньшего элемента   for (int j = low; j < high; j++) {  if (arr[j] <= pivot) {  i++;  swap(arr[i], arr[j]);  }  }  swap(arr[i + 1], arr[high]);  return i + 1; }  // Рекурсивная функция быстрой сортировки void quickSort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);   // Сортируем элементы до и после разделения  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  } }  int main() {  int arr1[] = {64, 34, 25, 12, 22, 11, 90};  int n1 = sizeof(arr1)/sizeof(arr1[0]);  cout << "Изначальный массив для пузырьковой сортировки: ";  printArray(arr1, n1);   bubbleSort(arr1, n1);  cout << "Отсортированный массив (итеративно): ";  printArray(arr1, n1);   int arr2[] = {64, 34, 25, 12, 22, 11, 90};  int n2 = sizeof(arr2)/sizeof(arr2[0]);  cout << "Изначальный массив для быстрой сортировки: ";  printArray(arr2, n2);   quickSort(arr2, 0, n2 - 1);  cout << "Отсортированный массив (рекурсивно): ";  printArray(arr2, n2);   return 0; } |
| --- |

Блок кода 3 - Объединение программ

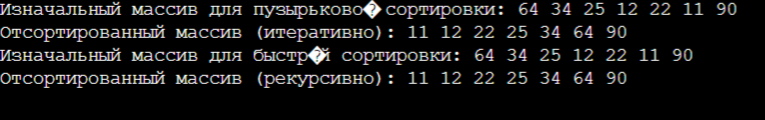


Рисунок 4 - Тестирование программы

# **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: Создать новый однонаправленный список из исходного однонаправленного списка, записав его элементы наоборот

## **3.2 Рекурсивная функция**

Реализуем задачу с помощью рекурсивной функции(блок кода 4).

| #include <iostream> using namespace std;  // Определение структуры узла списка struct Node {  int data;  Node\* next; };  // Функция для добавления элемента в начало списка void push(Node\*\* head\_ref, int new\_data) {  Node\* new\_node = new Node();  new\_node->data = new\_data;  new\_node->next = (\*head\_ref);  (\*head\_ref) = new\_node; }  // Рекурсивная функция для создания нового списка в обратном порядке Node\* reverseList(Node\* head, Node\* new\_head = nullptr) {  if (head == nullptr) {  return new\_head;  }  Node\* new\_node = new Node();  new\_node->data = head->data;  new\_node->next = new\_head;  return reverseList(head->next, new\_node); }  // Функция для печати списка void printList(Node\* node) {  while (node != nullptr) {  cout << node->data << " ";  node = node->next;  }  cout << endl; }  // Главная функция int main() {  // Создание исходного списка  Node\* head = nullptr;  push(&head, 20);  push(&head, 4);  push(&head, 15);  push(&head, 85);   cout << "Изначальный список: ";  printList(head);   // Реверсирование списка  Node\* reversedHead = reverseList(head);  cout << "Список в обратном порядке: ";  printList(reversedHead);   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

Продемонстрируем результаты работы программы на рисунке 4.



Рисунок 4 - Тестирование программы

Глубина рекурсии и теоретическая сложность

Глубина рекурсии в нашем случае будет равна количеству элементов в списке, так как каждый рекурсивный вызов обрабатывает один элемент.

Теоретическая временная сложность алгоритма — O(n), где n — количество элементов в списке. Пространственная сложность также будет O(n) из-за рекурсивных вызовов, которые сохраняются в стеке вызовов.

Глубина рекурсии

Глубина рекурсии равна количеству элементов в списке n.

# **4 ВЫВОДЫ**

Оба задания продемонстрировали использование итерационных и рекурсивных подходов для решения задач сортировки массива и обработки однонаправленного списка. Итерационный алгоритм был реализован для сортировки массива, а рекурсивный — для создания нового списка из исходного. Результаты тестирования подтвердили правильность и эффективность разработанных функций.

# 

# **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).