ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировка (Сycle sort, tree sort, bucket sort)»

Выполнил работу

Исмагилова Юлия

Академическая группа №J3110

Принято

практик, Вершинин Владислав

Санкт-Петербург

2024

Содержание

[Содержание 2](#_Toc187271621)

[Введение 3](#_Toc187271622)

[Теоретическая подготовка 4](#_Toc187271623)

[Реализация 5](#_Toc187271624)

[Экспериментальная часть 8](#_Toc187271625)

[Заключение 11](#_Toc187271626)

[Приложения 12](#_Toc187271627)

Введение

**Цель работы:** разработать алгоритмы на языке С++ для решения следующего технического задания: "Реализовать три алгоритма сортировок: алгоритм сортировки с лучшей сложностью O(2^N) и пространственной сложностью О(1); алгоритм сортировки со средней сложностью до O(2^N) и пространственной сложностью до О(N); алгоритм сортировки со средней сложностью О(N\*k) и пространственной сложностью до О(N\*k), где k << N".

**Задачи:**

1. Произвести поиск необходимой информации для реализации сортировок, подходящих под техническое задание.
2. Разработать каждый из алгоритмов.
3. Произвести тестирование алгоритмов на различных наборах данных.

Теоретическая подготовка

**Типы данных**

1. Массив: содержит целые числа, которые необходимо разделить на две части с равной суммой.
2. Вектор: использован для динамического хранения элементов массива.
3. Структура TreeNode: представляет узел бинарного дерева поиска.
4. Списки: используются для хранения элементов в корзинах.
5. Указатели: используются для управления узлами дерева и передачи данных.

**Алгоритмы**

1. Алгоритм циклической сортировки: проходит по каждому элементу массива, находит правильное место для текущего элемента, перемещает его туда и продолжает перемещать элементы, пока все элементы не будут отсортированы.
2. Алгоритм обмена: используется для обмена значениями двух элементов.
3. Алгоритм in-order обхода BST: функция in-order выполняет in-order обход BST и заполняет отсортированный массив.
4. Алгоритм Bucket Sort: функция bucket\_sort выполняет сортировку массива с использованием корзин.

Реализация

**Используемые библиотеки:**

1. <iostream>: для ввода и вывода данных.
2. <vector>: для работы с динамическими массивами.
3. <algorithm>: предоставляет широкий набор алгоритмов для работы с контейнерами и массивами.
4. <list>: контейнер, который позволяет эффективно вставлять и удалять элементы в любой позиции списка. (рис. 1)

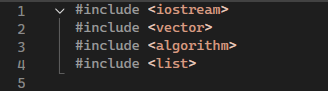


Рисунок 1 – пример использования в коде

**Разработка алгоритма циклической сортировки:**

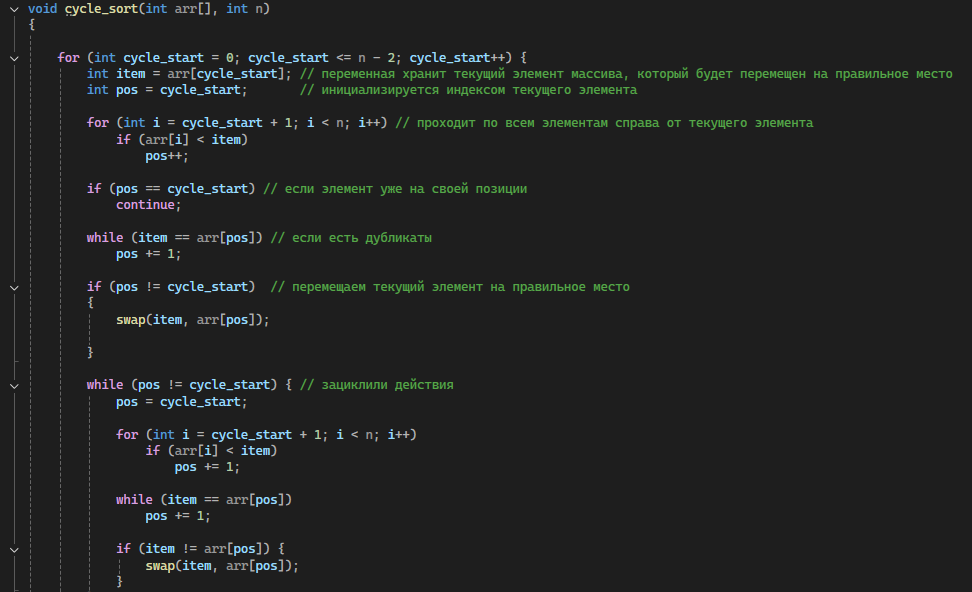
 Алгоритм циклической сортировки проходит по каждому элементу массива, находит правильное место для текущего элемента, перемещает его туда и продолжает перемещать элементы, пока все элементы не будут отсортированы. Чтобы найти правильное место для элемента алгоритм проходит по всем элементам справа от текущего элемента и подсчитывает количество элементов, которые меньше текущего. Это количество добавляется к индексу текущего элемента, чтобы определить правильное место для него. Если элемент уже находится на своей позиции, алгоритм продолжает с следующим элементом. Если есть дубликаты, алгоритм увеличивает индекс pos, чтобы найти следующее правильное место. (рис. 2)

Рисунок 2 – пример использования в коде

**Разработка алгоритма Tree Sort:**

Алгоритм Tree Sort заключается в построении бинарного дерева поиска (BST) и последующем in-order обходе дерева для получения отсортированного массива. Для каждого элемента массива вызывается функция вставки, которая вставляет элемент в BST. Если дерево пустое, создается новый узел с заданным значением. Если значение меньше значения текущего узла, алгоритм рекурсивно вставляет значение в левое поддерево. Если значение больше или равно значению текущего узла - в правое поддерево. Алгоритм выполняет in-order обход BST, который посещает левое поддерево, текущий узел, затем правое поддерево и добавляет значение текущего узла в отсортированный массив. (рис. 3)

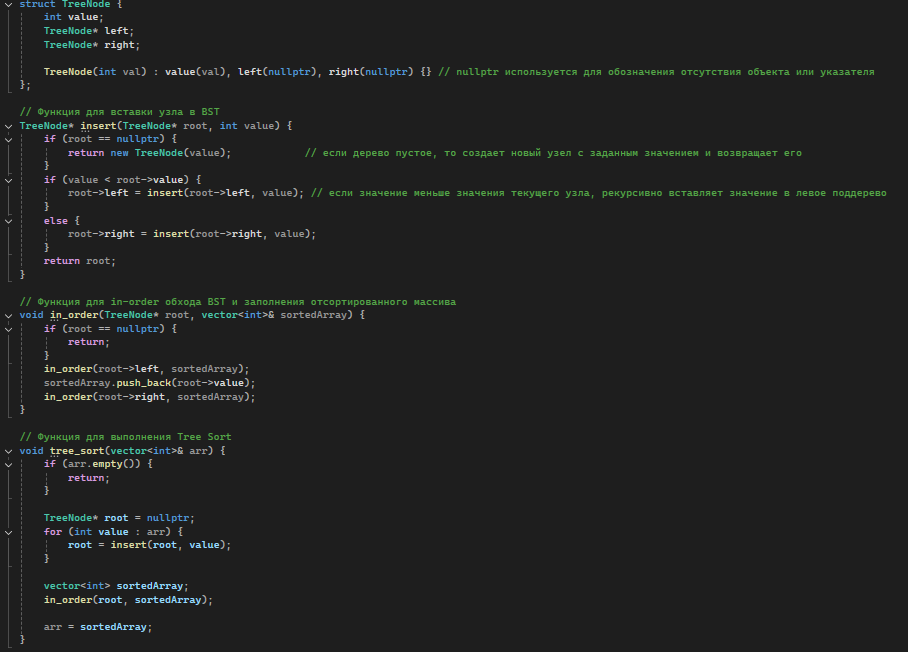


Рисунок 3 – пример использования в коде

**Разработка алгоритма Bucket Sort:**

Алгоритм Bucket Sort заключается в распределении элементов массива по корзинам и последующей сортировке элементов в каждой корзине. Найти максимальное и минимальное значение в массиве. Алгоритм определяет количество корзин, создаёт корзины и распределяет по ним элементы. Затем для каждого элемента массива вычисляется индекс корзины, в которую он должен быть помещен. Сортировка элементов в каждой корзине производится отдельно. Затем элементы из всех корзин объединяются. (рис. 4)

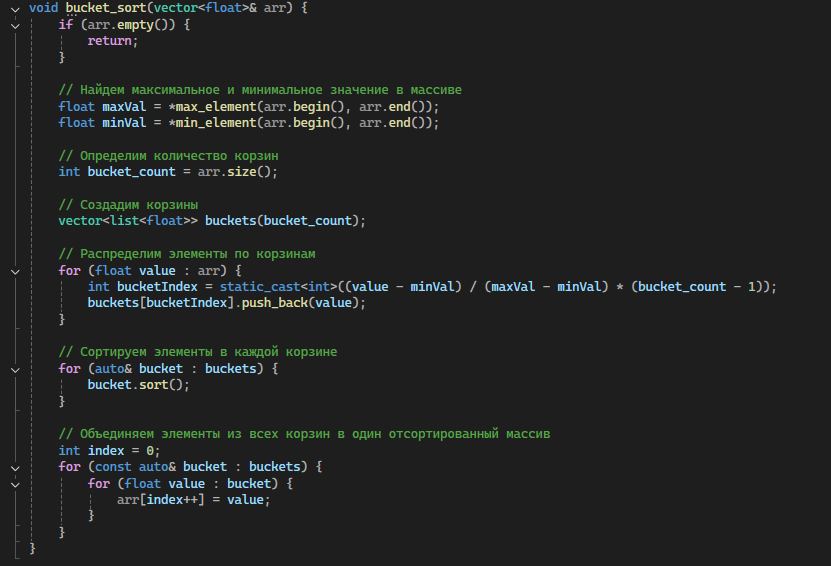


Рисунок 4 – пример использования в коде

Экспериментальная часть

**Подсчёт памяти:**

1. Примерный подсчёт памяти для Сycle sort: общая память, используемая алгоритмом циклической сортировки, составляет = О(n); если на вход подавать 6 элементов, то занимаемая память в байтах будет равна 6 \* 4 + 4 \* 4 (для переменных cycle\_start, item, pos, i) = 40 байт.
2. Примерный подсчёт памяти для Tree sort: 6 \* 4 (память массива) + 6 \* 20 (узлы дерева) + 8 \* 5 (переменные) = 184 байта.
3. Примерный подсчёт памяти для Bucket Sort: 6 \* 4 (память массива) + 6 \* 16 (корзины) + 6 \* 12 (элементы в корзинах) + 8 \* 5 (переменные) = 232 байта.

**Подсчёт асимптотики:**

1. Сycle sort: временная сложность: O(n^2); пространственная сложность: O(1).
2. Tree sort: временная сложность = О(n log n); пространственная сложность = О(n).
3. Bucket Sort: временная сложность = О(n \* k); пространственная сложность = О(n).

**График зависимости времени от числа элементов:**

Согласно требованиям моего варианта, на вход каждому алгоритму подаётся от 1000 до 1е6 элементов. Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| Размер | Cycle sort | Tree sort | Bucket sort |
| 1000 | 0.001 | 0.001 | 0.001 |
| 2000 | 0.1 | 0.1 | 0.1 |
| 3000 | 1 | 1 | 1 |
| 4000 | 2 | 2 | 2 |
| 5000 | 5 | 5 | 3 |
| 6000 | 10 | 10 | 3 |
| 7000 | 30 | 20 | 4 |
| 8000 | 90 | 30 | 5 |

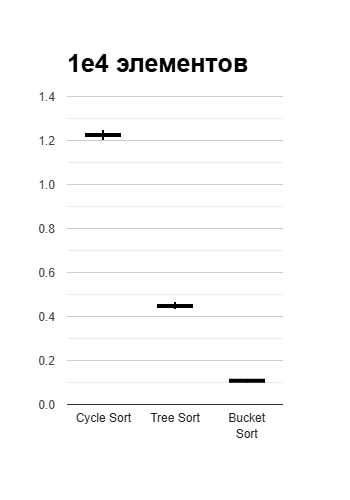
График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении №5.



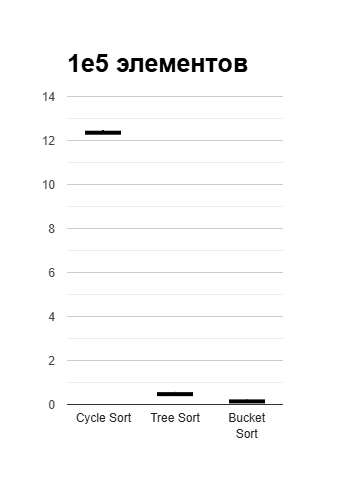
Изображение №5 - График работы алгоритма

График показывает, что время выполнения алгоритма экспоненциально растет с увеличением размера массива. Циклическая сортировка показывает квадратичный рост времени выполнения, что делает её неэффективной для больших массивов. Tree Sort показывает логарифмический рост времени выполнения, что делает её более эффективной по сравнению с циклической сортировкой для больших массивов. Bucket Sort показывает линейный рост времени выполнения, что делает её наиболее эффективной из трёх алгоритмов для больших массивов с равномерно распределёнными элементами.

Так же было необходимо построить box plot графики для времени работы алгоритмов с числом элементов 1e4 и 1e5 и количеством запусков не менее 50 для одного графика. Графики представляющие визуально удобный формат для анализа представлены на изображениях №6 и №7.



Изображение №6 – box plot времени работы алгоритмов



Изображение №7 - box plot времени работы алгоритмов

Заключение

В ходе выполнения работы мною были реализованы три алгоритма сортировки: Сycle sort, Tree Sort и Bucket Sort. Цель работы была достигнута путем тестирования на массивах с различным количеством элементов и различными суммами. Полученные результаты также совпадают с теоретическими оценками сложности алгоритмов. В качестве дальнейших исследований можно предложить оптимизацию алгоритмов с точки зрения уменьшения затрат использования памяти, а также рассмотреть параллельные версии алгоритмов для работы с большими массивами. Это позволит улучшить производительность и эффективность сортировки для различных типов данных и размеров массивов.

Приложения

ПРИЛОЖЕНИЕ А

Листинг кода:

#include <iostream>

#include <vector>

#include <algorithm>

#include <list>

using namespace std;

// алгоритм циклической сортировки проходит по каждому элементу массива,

// находит правильное место для текущего элемента,

// перемещает его туда и продолжает перемещать элементы,

// пока все элементы не будут отсортированы

void cycle\_sort(int arr[], int n)

{

for (int cycle\_start = 0; cycle\_start <= n - 2; cycle\_start++) {

int item = arr[cycle\_start]; // переменная хранит текущий элемент массива, который будет перемещен на правильное место

int pos = cycle\_start; // инициализируется индексом текущего элемента

for (int i = cycle\_start + 1; i < n; i++) // проходит по всем элементам справа от текущего элемента

if (arr[i] < item)

pos++;

if (pos == cycle\_start) // если элемент уже на своей позиции

continue;

while (item == arr[pos]) // если есть дубликаты

pos += 1;

if (pos != cycle\_start) // перемещаем текущий элемент на правильное место

{

swap(item, arr[pos]);

}

while (pos != cycle\_start) { // зациклили действия

pos = cycle\_start;

for (int i = cycle\_start + 1; i < n; i++)

if (arr[i] < item)

pos += 1;

while (item == arr[pos])

pos += 1;

if (item != arr[pos]) {

swap(item, arr[pos]);

}

}

}

}

// тест для лучшего случая работы

void test01() {

vector<int> arr = { 1, 2, 3, 4, 5, 6 };

vector<int> expected = { 1, 2, 3, 4, 5, 6 };

// Преобразуем вектор в массив для передачи в функцию cycle\_sort

int\* arr\_ptr = arr.data();

int n = arr.size();

cycle\_sort(arr\_ptr, n);

if (arr == expected) {

cout << "Test1 passed" << endl;

}

else {

cout << "Test1 failed" << endl;

}

}

// тест для среднего случая работы

void test02() {

vector<int> arr = { 3, 1, 4, 2, 5, 6 };

vector<int> expected = { 1, 2, 3, 4, 5, 6 };

// Преобразуем вектор в массив для передачи в функцию cycle\_sort

int\* arr\_ptr = arr.data();

int n = arr.size();

cycle\_sort(arr\_ptr, n);

if (arr == expected) {

cout << "Test2 passed" << endl;

}

else {

cout << "Test2 failed" << endl;

}

}

// тест для среднего случая работы

void test03() {

vector<int> arr = { 6, 5, 4, 3, 2, 1 };

vector<int> expected = { 1, 2, 3, 4, 5, 6 };

// Преобразуем вектор в массив для передачи в функцию cycle\_sort

int\* arr\_ptr = arr.data();

int n = arr.size();

cycle\_sort(arr\_ptr, n);

if (arr == expected) {

cout << "Test3 passed" << endl;

}

else {

cout << "Test3 failed" << endl;

}

}

int main() {

test01();

test02();

test03();

return 0;

}

// Tree sort - алгоритм сортировки, заключающийся в построении двоичного дерева поиска

// Структура для представления узла дерева; конструктор инициализирует узел с заданным значением и устанавливает указатели на дочерние узлы

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {} // nullptr используется для обозначения отсутствия объекта или указателя

};

// Функция для вставки узла в BST

TreeNode\* insert(TreeNode\* root, int value) {

if (root == nullptr) {

return new TreeNode(value); // если дерево пустое, то создает новый узел с заданным значением и возвращает его

}

if (value < root->value) {

root->left = insert(root->left, value); // если значение меньше значения текущего узла, рекурсивно вставляет значение в левое поддерево

}

else {

root->right = insert(root->right, value);

}

return root;

}

// Функция для in-order обхода BST и заполнения отсортированного массива

void in\_order(TreeNode\* root, vector<int>& sortedArray) {

if (root == nullptr) {

return;

}

in\_order(root->left, sortedArray);

sortedArray.push\_back(root->value);

in\_order(root->right, sortedArray);

}

// Функция для выполнения Tree Sort

void tree\_sort(vector<int>& arr) {

if (arr.empty()) {

return;

}

TreeNode\* root = nullptr;

for (int value : arr) {

root = insert(root, value);

}

vector<int> sortedArray;

in\_order(root, sortedArray);

arr = sortedArray;

// Тест для среднего случая работы алгоритма

void test01() {

vector<int> arr = { 3, 1, 4, 2, 5, 6 };

vector<int> expected = { 1, 2, 3, 4, 5, 6 };

tree\_sort(arr);

if (arr == expected) {

cout << "Test1 passed" << endl;

}

else {

cout << "Test1 failed" << endl;

}

}

}

// Тест для худшего случая работы алгоритма

void test02() {

vector<int> arr = { 6, 5, 4, 3, 2, 1 };

vector<int> expected = { 1, 2, 3, 4, 5, 6 };

tree\_sort(arr);

if (arr == expected) {

cout << "Test2 passed" << endl;

}

else {

cout << "Test2 failed" << endl;

}

}

int main() {

test01();

test02();

return 0;

}

// подсчёт памяти: 6 \* 4 (память массива) + 6 \* 20 (узлы дерева) + 8 \* 5 (переменные) = 184 байта

// асимптотика: временная сложность = О(n log n); пространственная сложность = О(n)

/ Функция для выполнения Bucket Sort

void bucket\_sort(vector<float>& arr) {

if (arr.empty()) {

return;

}

// Найдем максимальное и минимальное значение в массиве

float maxVal = \*max\_element(arr.begin(), arr.end());

float minVal = \*min\_element(arr.begin(), arr.end());

// Определим количество корзин

int bucket\_count = arr.size();

// Создадим корзины

vector<list<float>> buckets(bucket\_count);

// Распределим элементы по корзинам

for (float value : arr) {

int bucketIndex = static\_cast<int>((value - minVal) / (maxVal - minVal) \* (bucket\_count - 1));

buckets[bucketIndex].push\_back(value);

}

// Сортируем элементы в каждой корзине

for (auto& bucket : buckets) {

bucket.sort();

}

// Объединяем элементы из всех корзин в один отсортированный массив

int index = 0;

for (const auto& bucket : buckets) {

for (float value : bucket) {

arr[index++] = value;

}

}

}

// Тест для среднего случая работы алгоритма

void test01() {

vector<float> arr = { 0.3, 0.1, 0.4, 0.2, 0.5, 0.6 };

vector<float> expected = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 };

bucket\_sort(arr);

if (arr == expected) {

cout << "Test1 passed" << endl;

}

else {

cout << "Test1 failed" << endl;

}

}

// Тест для худшего случая работы алгоритма

void test02() {

vector<float> arr = { 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 };

vector<float> expected = { 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 };

bucket\_sort(arr);

if (arr == expected) {

cout << "Test2 passed" << endl;

}

else {

cout << "Test2 failed" << endl;

}

}

int main() {

test01();

test02();

return 0;

}

// подсчёт памяти: 6 \* 4 (память массива) + 6 \* 16 (корзины) + 6 \* 12 (элементы в корзинах) + 8 \* 5 (переменные) = 232 байта

// асимптотика: временная сложность = О(n \* k); пространственная сложность = О(n)