ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировка массива»

Выполнил работу

Маркелова Софья

Академическая группа № С3100

Принял

Ментор, Владислав Вершинин

Санкт-Петербург

2024

Введение:

Цель лабораторной работы – реализовать 3 алгоритма сортировки с разной временной и пространственной сложностью. Необходимо оценить полученную сложность алгоритмов и сопоставить время исполнения алгоритма с расчетным временем исполнения алгоритма в зависимости от количества элементов в исходном массиве. Также необходимо проанализировать полученные результаты, чтобы понять, в каких случаях стоит использовать тот или иной алгоритм сортировки.

Для достижения цели были установлены следущие задачи: реализовать алогиритм шейкерной сортировки с лучшей сложностью O(N^2) и пространственной O(1); алгоритм быстрой сортировки со средней сложностью до O(N^2) и пространственной до O(N); алгоритм блочной сортировки со средней сложностью до O(N\*k) и пространственной до O(N\*k), где k значительно меньше N.

1. Теоретическа подготовка:

Для выполнения работы необходимо обладать теоретической подготовкой в части понимания работы с генератором случайных чисел, рекурсией. Также потребуется понимание метода преобразования числа десятичной системы счисления в другую систему (двоичную, троичную, и.т.д). Необходимо знать принцип оценки времени исполнения программы и расчет оценки сложности алгоритма.

1. Реализация

В процессе реализации были использованы стандартная библиотека iostream для вывода результата, chrono для фиксации времени работы, vector для работы со структурой данных типа вектор, cstdlib для инициализации генератора случайных чисел, algorithm для работы со стандартным набором алгоритмов.

Реализация шейкерной сортировки:

Реализация функции shakerSort. Каждая итерация алгоритма разбивается на два этапа. Первый этап перебирает массив слева направо. Во время цикла сравниваются соседние элементы, и если значение слева больше значения справа, значения меняются местами. В конце первой итерации наибольшее число будет находиться в конце массива.

do {

        // прямой проход: перемещение больших элементов вправо

        for (i = left; i < right; i++)

        // если текущий элемент больше последующего, меняем их местами

        if (arr[i] > arr[i + 1]) {

            t = arr[i];

            arr[i] = arr[i + 1];

            arr[i + 1] = t;

        }

        // уменьшаем правую границу, т. к. последний элемент найден

        -- right;

Второй этап проходит по массиву в обратном направлении — начиная с элемента, непосредственно предшествующего последнему отсортированному элементу, и возвращаясь к началу массива. Здесь также сравниваются соседние элементы и при необходимости меняются местами.

// обратный проход: перемещение меньших элементов влево

        for (i = right; i > left; --i)

        // если текущий элемент меньше предыдущего, меняем их местами

        if (arr[i] < arr[i - 1]) {

            t = arr[i];

            arr[i] = arr[i - 1];

            arr[i - 1] = t;

        }

        // увеличиваем левую границу, т. к. первый элемент найден

        ++left;

    } while (left < right);

}

Реализация быстрой сортировки:

Реализация функции qsortRecursive. Изначально рассматривается базовый случай, если размер массива меньше или равен 1, массив уже отсортирован,функция возвращает управление. Затем инициализируются два указателя i и j для обхода массива с двух сторон, в качестве опорного элемента выбирается центральный элемент массива mid.

// базовый случай: если размер массива меньше или равен 1, массив отсортирован

    if (size <= 1) return;

    int i = 0; // указатель для перебора слева

    int j = size - 1; // указатель для перебора справа

    int mid = arr[size / 2];  // центральный элемент массива (опорный элемент)

В следующем шаге происходит разделение массива на 2 части. Цикл do...while используется для перемещения указателей i и j. Внутренние циклы while увеличивают i, пока не найдется элемент, который больше или равен опорному элементу, и уменьшают j, пока не найдется элемент, который меньше или равен опорному. Если i меньше или равен j, элементы на позициях i и j меняются местами с помощью функции swap.

// делим массив на 2 части

    do {

        // перемещаем указатель i вправо, пока не найдем элемент, который больше или равен mid

        while (arr[i] < mid) {

            i++;

        }

        // перемещаем указатель j влево, пока не найдем элемент, который меньше или равен mid

        while (arr[j] > mid) {

            j--;

        }

        // если указатели i и j не пересеклись, меняем элементы местами

        if (i <= j) {

            swap(arr[i], arr[j]); // Используем std::swap для удобства

            i++;

            j--;

        }

После разделения массива на две части (левая и правая), функция вызывает саму себя рекурсивно для сортировки этих частей. Первый вызов сортирует левую часть массива от начала до позиции j. Второй вызов сортирует правую часть массива начиная с позиции i.

// Рекурсивные вызовы для сортировки левой и правой частей массива

    if (j >= 0) {

        // сортируем левую часть до j (включительно)

        qsortRecursive(arr, j + 1);

    }

    if (i < size) {

        // сортируем правую часть начиная с i

        qsortRecursive(&arr[i], size - i);

    }

Реализация вёдерной сортировки:

Реализация функции bucketSort. На превом шаге происходит поиск минимального и максимального элементов в массиве с помощью цикла for, который проходит по всем элементам массива и обновляет значения, если находит большие изначальных.

int maxVal = arr[0];

    int minVal = arr[0];

    for (int i = 1; i < size; i++) {

        if (arr[i] > maxVal) { // O(n)

            maxVal = arr[i];

        }

        if (arr[i] < minVal) { // O(n)

            minVal = arr[i];

        }

    }

Затем определяется количество «вёдер» как максимум между 1 и размером массива, делённым на 5. Это позволяет создать достаточное количество ведер для распределения элементов, но не перегружать их.

/ количество ведер

    int bucketCount = max(1, size / 5); //  1 ведро на каждые 5 элементов

Затем создается вектор векторов (или массив ведер), где каждое ведро будет представлено как отдельный вектор целых чисел.

vector<vector<int>> buckets(bucketCount);

bucketRange вычисляет диапазон значений, который будет помещен в каждое ведро. Используется приведение типа для получения точного значения типа double.

// размер ведра

    double bucketRange = static\_cast<double>(maxVal - minVal + 1) / bucketCount;

Следующим шагом происходит распределение элементов по вёдрам. Цикл for проходит по всем элементам массива. Для каждого элемента вычисляется индекс ведра (bucketIndex), в которое он будет помещен. Если индекс выходит за пределы, он корректируется, чтобы не превышать количество ведер. Элемент добавляется в соответствующее ведро с помощью метода push\_back.

// распределяем элементы по ведрам

    // O(n)

    for (int i = 0; i < size; i++) {

        int bucketIndex = static\_cast<int>((arr[i] - minVal) / bucketRange);

        if (bucketIndex >= bucketCount) { // в случае, если индекс выходит за пределы

            bucketIndex = bucketCount - 1;

        }

        buckets[bucketIndex].push\_back(arr[i]);

    }

Последни этапом идёт сортировка вёдер и сбор результатов. Внешний цикл проходит по всем ведрам. Если ведро не пустое, оно сортируется с использованием стандартной функции sort, что обычно является сортировкой вставками или быстрой сортировкой. После сортировки элементы из ведра собираются обратно в исходный массив arr, начиная с позиции index.

// сортируем каждое ведро и собираем результаты

    int index = 0;

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

        // используем сортировку вставками для небольших ведер

        if (!buckets[i].empty()) {

            sort(buckets[i].begin(), buckets[i].end());

            for (int j = 0; j < buckets[i].size(); j++) {

                arr[index++] = buckets[i][j]; // собираем отсортированные элементы обратно в массив

            }

        }

    }

Общее для трех алгоритмов сортировки:

Для каждого алгоритма сортировки были написаны тесты для лучшего, среднего и худшего случаев, чтобы проверить работу алгоритмов.

Для шейкерной сортировки:

int best\_case[] = {1, 2, 4, 4, 6, 8, 12};

int averege\_case[] = {1, 4, 2, 8, 6, 12, 4};

int worst\_case[] = {12, 8, 6, 4, 2, 1};

Для быстрой сортировки:

int best\_case[] = {1, 2, 4, 4, 6, 8, 12};

int averege\_case[] = {1, 4, 2, 8, 6, 12, 4};

int worst\_case[] = {12, 8, 6, 4, 2, 1};

Для вёдерной сортировки:

int best\_case[] = {1, 2, 4, 4, 6, 8, 12};

int averege\_case[] = {1, 4, 2, 8, 6, 12, 4};

int worst\_case[] = {12, 12, 12, 12, 12, 12};

1. Эксперементальная часть

Для каждого алгоритма сортировки существует разная сложность. Так, для лучшего случай, когда массив уже отсортирован сложность шейкерной сортировки состовляет O(N), для быстрой сортировки – O(n\*log(n)), для вёдерной сортировки O(n + k), где k – это максимальное количество элементов в люом ведре (при равномерном распределении). Для среднего случая, когда элементы в массиве генерируются случайным образом, для шейкерной сортировки сложность составляет O(N^2), для быстрой сортировки - O(n\*log(n)), для вёдерной - O(n + k\*log(k)), где k – количество вёдер. Худший случай для шейкерной и быстрой сортировок совпадают, все элементы массива расположены в обратном порядке, таким образом сложность шейкерной сортировки и быстрой сортировки составляет O(N^2). Худший случай для вёдерной сортировки отличается, в массиве все элементы равные, таким образом они все попадают в одно ведро, сложность данной сортировки возрастает до O(N^2). Итоговая память, которая выделяется в функции shakerSort равняется 16 байт. Итоговая память, которая выделяется в функции qsortRecursive равняется 12 байт. Итоговая память, которая выделяется в функции shakerSort равняется 16 байт. Итоговая память, которая выделяется в функции bucketSort равняется 8 байт.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| Размер входного массива | время, микросек  shaker\_sort | время, микросек  quick\_sort | время, микросек  bucket\_sort |
| 1000 | 1739 | 105 | 155 |
| 2000 | 6281 | 201 | 276 |
| 3000 | 14344 | 337 | 574 |
| 4000 | 32442 | 650 | 711 |
| 5000 | 37727 | 670 | 801 |
| 6000 | 51326 | 721 | 971 |
| 7000 | 88420 | 799 | 898 |
| 8000 | 105751 | 1001 | 1348 |

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

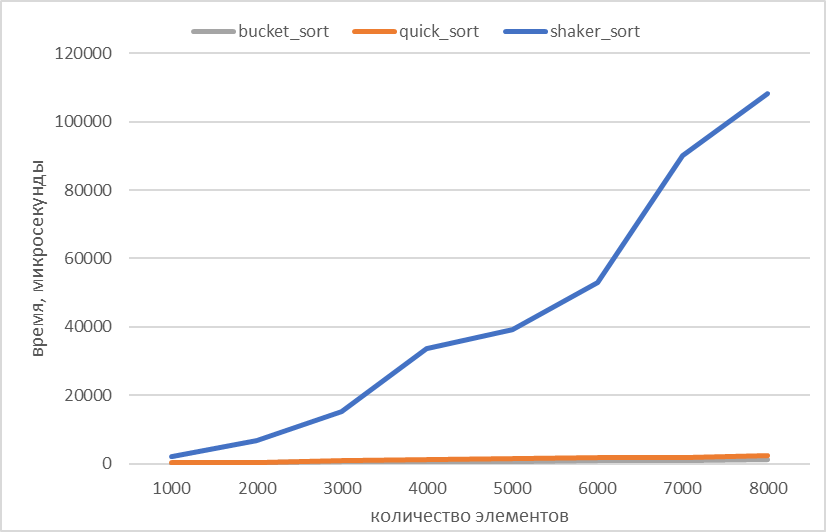


Рисунок №1 - График работы алгоритма

Боксплот, представляющий визальную диаграмму для «среднего» времени, которое требуется для выполнения сортировки, изображение №2.

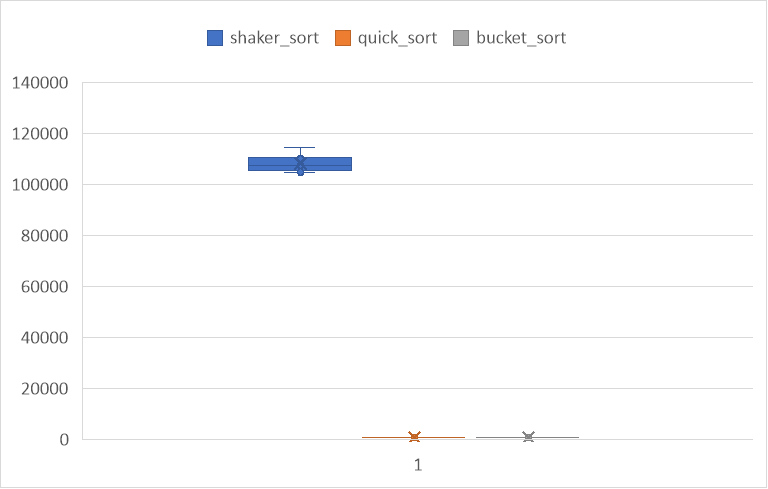


Рисунок №2

Время выполнения реализованных алгоритмов для средних случаев совпадает с заявленной сложностью.

Заключение

В ходе выполнения работы мною были реализованы алгоритмы шейкерной сортировки, быстрой сортировки, вёдерной сортировки. Цель работы была достигнута путём тестирования на массивах с различным числом и комбинациями входных значений. Полученные результаты также совпадают с теоретическими оценками сложности алгоритма. Каждый алгоритм сортировки имеет свои особенности, а также условия, когда стоит их использовать. Так, мы убедились, что использовать шейкерную сортировку при больших входных данных будет неэффективно, данная сортировка подходит для небольших или почти отсортированных массивов. Быстрая сортировка напротив очень эффективна для больших массивов, а также в работе со случайными данными. Вёдерная сортировка эффективна для распределённых данных, когда значения элементов известны и ограничены, данная сортировка может работать очень быстро при условии, что данные равномерно распределенв и вёдра не переполнены.

ПРИЛОЖЕНИЕ А

Листинг кода файла quick\_sort.cpp

Шейкерная сорировка:

#include <iostream>

#include <chrono>

#include <cstdlib>

#include <algorithm>

#include <vector>

using namespace std;

void shakerSort(int\* arr, int array\_size) {

    int i; // переменная для итерации; 4 байта

    int t; // переменная для временного хранения значения при обмене элементов; 4 байта

    int left = 0; // переменная для отслеживания индекса текущей левой границы; 4 байта

    int right = array\_size - 1; // переменная для отслеживания индекса текущей правой границы; 4 байта

    // основной цикл, выполняется, пока левая граница меньше правой

    do {

        // прямой проход: перемещение больших элементов вправо

        for (i = left; i < right; i++)

        // если текущий элемент больше последующего, меняем их местами

        if (arr[i] > arr[i + 1]) {

            t = arr[i];

            arr[i] = arr[i + 1];

            arr[i + 1] = t;

        }

        // уменьшаем правую границу, т. к. последний элемент найден

        -- right;

        // обратный проход: перемещение меньших элементов влево

        for (i = right; i > left; --i)

        // если текущий элемент меньше предыдущего, меняем их местами

        if (arr[i] < arr[i - 1]) {

            t = arr[i];

            arr[i] = arr[i - 1];

            arr[i - 1] = t;

        }

        // увеличиваем левую границу, т. к. первый элемент найден

        ++left;

    } while (left < right);

}

// Функция для проверки результата теста и вывода соответствующего сообщения

void assertTest(bool ans) {

    if (ans) {

        cout << "The test is passed" << '\n';

    } else {

        cout << "The test is not passed" << '\n';

    }

}

// тесты

void runTest() {

// лучший случай

// O(n)

int best\_case[] = {1, 2, 4, 6, 8, 12};

int expected\_best\_case[] = {1, 2, 4, 6, 8, 12};

int size\_best\_case = sizeof(best\_case) / sizeof(best\_case[0]);

shakerSort(best\_case, size\_best\_case);

bool result\_best\_case = equal(begin(best\_case), end(best\_case), begin(expected\_best\_case));

assertTest(result\_best\_case);

// средний случай

// O(n^2)

int averege\_case[] = {1, 4, 2, 8, 6, 12};

int expected\_averege\_case[] = {1, 2, 4, 6, 8, 12};

int size\_averege\_case = sizeof(averege\_case) / sizeof(averege\_case[0]);

shakerSort(averege\_case, size\_averege\_case);

bool result\_averege\_case = equal(begin(averege\_case), end(averege\_case), begin(averege\_case));

assertTest(result\_averege\_case);

// худший случай

// O(n^2)

int worst\_case[] = {12, 8, 6, 4, 2, 1};

int expected\_worst\_case[] = {1, 2, 4, 6, 8, 12};

int size\_worst\_case = sizeof(worst\_case) / sizeof(worst\_case[0]);

shakerSort(worst\_case, size\_worst\_case);

bool result\_worst\_case = equal(begin(worst\_case), end(worst\_case), begin(expected\_worst\_case));

assertTest(result\_worst\_case);

}

void generateRandomNumbers(vector<int>& numbers, int count) {

    // Инициализация генератора случайных чисел

    srand(static\_cast<unsigned int>(std::time(nullptr)));

    for (int i = 0; i < count; ++i) {

        numbers.push\_back(rand() % 100); // Генерация случайного числа от 0 до 99

    }

}

int main() {

    int arr[] = {2, 78, 56, 34, 2, 12, 67, 90, 42, 11};

    int size =sizeof(arr) / sizeof(arr[0]);

    shakerSort(arr, size);

    for (int i = 0; i < size; ++i) {

        cout << arr[i] << " ";

    }

    runTest();

    const int count = 10000; // Количество случайных чисел

    vector<int> randomNumbers;

    // Генерация случайных чисел

    generateRandomNumbers(randomNumbers, count);

    // Преобразование std::vector в массив для сортировки

    int\* randomArr = randomNumbers.data();

    // Записываем текущее время работы

    auto start = chrono::high\_resolution\_clock::now();

    shakerSort(randomArr, count);

    // Текущее время окончания выполнения работы

    auto end = chrono::high\_resolution\_clock::now();

    // Продолжительность выполнения работы

    auto duration = chrono::duration\_cast< chrono::microseconds>(end - start);

    cout << "Sorting completed." << endl;

    cout << "Running time: " << duration.count() << " milliseconds" << endl;

    return 0;

}

Быстрая сортировка:

#include <iostream>

#include <chrono>

#include <cstdlib>

#include <algorithm>

#include <vector>

using namespace std;

void qsortRecursive(int \*arr, int size) {

    // базовый случай: если размер массива меньше или равен 1, массив отсортирован

    if (size <= 1) return;

    int i = 0; // указатель для перебора слева; 4 байта

    int j = size - 1; // указатель для перебора справа; 4 байта

    int mid = arr[size / 2];  // центральный элемент массива (опорный элемент); 4 байта

    // делим массив на 2 части

    do {

        // перемещаем указатель i вправо, пока не найдем элемент, который больше или равен mid

        while (arr[i] < mid) {

            i++;

        }

        // перемещаем указатель j влево, пока не найдем элемент, который меньше или равен mid

        while (arr[j] > mid) {

            j--;

        }

        // если указатели i и j не пересеклись, меняем элементы местами

        if (i <= j) {

            swap(arr[i], arr[j]); // Используем std::swap для удобства

            i++;

            j--;

        }

    // продолжаем, пока указатели не пересекутся

    } while (i <= j);

    // Рекурсивные вызовы для сортировки левой и правой частей массива

    if (j >= 0) {

        // сортируем левую часть до j (включительно)

        qsortRecursive(arr, j + 1);

    }

    if (i < size) {

        // сортируем правую часть начиная с i

        qsortRecursive(&arr[i], size - i);

    }

}

// Функция для проверки результата теста и вывода соответствующего сообщения

void assertTest(bool ans) {

    if (ans) {

        cout << "The test is passed" << '\n';

    } else {

        cout << "The test is not passed" << '\n';

    }

}

void generateRandomNumbers(vector<int>& numbers, int count) {

    // Инициализация генератора случайных чисел

    srand(static\_cast<unsigned int>(time(nullptr)));

    for (int i = 0; i < count; ++i) {

        numbers.push\_back(rand() % 100); // Генерация случайного числа от 0 до 99

    }

}

// тесты

void runTest() {

// лучший случай

// O(n\*log(n))

int best\_case[] = {1, 2, 4, 6, 8, 12};

int expected\_best\_case[] = {1, 2, 4, 6, 8, 12};

int size\_best\_case = sizeof(best\_case) / sizeof(best\_case[0]);

qsortRecursive(best\_case, size\_best\_case);

bool result\_best\_case = equal(begin(best\_case), end(best\_case), begin(expected\_best\_case));

assertTest(result\_best\_case);

// средний случай

// O(n\*log n)

int averege\_case[] = {1, 4, 2, 8, 6, 12};

int expected\_averege\_case[] = {1, 2, 4, 6, 8, 12};

int size\_averege\_case = sizeof(averege\_case) / sizeof(averege\_case[0]);

qsortRecursive(averege\_case, size\_averege\_case);

bool result\_averege\_case = equal(begin(averege\_case), end(averege\_case), begin(averege\_case));

assertTest(result\_averege\_case);

// худший случай

// O(n^2)

int worst\_case[] = {12, 8, 6, 4, 2, 1};

int expected\_worst\_case[] = {1, 2, 4, 6, 8, 12};

int size\_worst\_case = sizeof(worst\_case) / sizeof(worst\_case[0]);

qsortRecursive(worst\_case, size\_worst\_case);

bool result\_worst\_case = equal(begin(worst\_case), end(worst\_case), begin(expected\_worst\_case));

assertTest(result\_worst\_case);

}

int main() {

    int arr[] = {0, 23, 65, 4, 1, 90, 7, 7};

    int size = sizeof(arr) / sizeof(arr[0]);

    qsortRecursive(arr, size);

    for (int i = 0; i < size; ++i) {

        cout << arr[i] << " ";

    }

    runTest();

    const int count = 10000; // Количество случайных чисел

    vector<int> randomNumbers;

    // Генерация случайных чисел

    generateRandomNumbers(randomNumbers, count);

    // Преобразование std::vector в массив для сортировки

    int\* randomArr = randomNumbers.data();

    // Записываем текущее время работы

    auto start = chrono::high\_resolution\_clock::now();

    qsortRecursive(randomArr, count);

    // Текущее время окончания выполнения работы

    auto end = chrono::high\_resolution\_clock::now();

    // Продолжительность выполнения работы

    auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

    cout << "Sorting completed." << endl;

    cout << "Running time: " << duration.count() << " microseconds" << endl;

    return 0;

}

Вёдерная сортировка:

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <cstdlib>

using namespace std;

void bucketSort(int arr[], int size) {

    // находим максимальное и минимальное значение в массиве

    int maxVal = arr[0]; // 4 байта

    int minVal = arr[0]; // 4 байта

    for (int i = 1; i < size; i++) {

        if (arr[i] > maxVal) { // O(n)

            maxVal = arr[i];

        }

        if (arr[i] < minVal) { // O(n)

            minVal = arr[i];

        }

    }

    // количество ведер

    int bucketCount = max(1, size / 5); //  1 ведро на каждые 5 элементов

    vector<vector<int>> buckets(bucketCount);

    // размер ведра

    double bucketRange = static\_cast<double>(maxVal - minVal + 1) / bucketCount;

    // распределяем элементы по ведрам

    // O(n)

    for (int i = 0; i < size; i++) {

        int bucketIndex = static\_cast<int>((arr[i] - minVal) / bucketRange);

        if (bucketIndex >= bucketCount) { // в случае, если индекс выходит за пределы

            bucketIndex = bucketCount - 1;

        }

        buckets[bucketIndex].push\_back(arr[i]);

    }

    // сортируем каждое ведро и собираем результаты

    int index = 0;

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

        // используем сортировку вставками для небольших ведер

        if (!buckets[i].empty()) {

            sort(buckets[i].begin(), buckets[i].end());

            for (int j = 0; j < buckets[i].size(); j++) {

                arr[index++] = buckets[i][j]; // собираем отсортированные элементы обратно в массив

            }

        }

    }

}

// функция для проверки результата теста и вывода соответствующего сообщения

void assertTest(bool ans) {

    if (ans) {

        cout << "The test is passed" << '\n';

    } else {

        cout << "The test is not passed" << '\n';

    }

}

void generateRandomNumbers(vector<int>& numbers, int count) {

    // Инициализация генератора случайных чисел

    srand(static\_cast<unsigned int>(time(nullptr)));

    for (int i = 0; i < count; ++i) {

        numbers.push\_back(rand() % 100); // Генерация случайного числа от 0 до 99

    }

}

// тесты

void runTest() {

// лучший случай

// O(n + k), где k — это максимальное количество элементов в любом ведре (при равномерном распределении)

int best\_case[] = {1, 2, 4, 4, 6, 8, 12};

int expected\_best\_case[] = {1, 2, 4, 4, 6, 8, 12};

int size\_best\_case = sizeof(best\_case) / sizeof(best\_case[0]);

bucketSort(best\_case, size\_best\_case);

bool result\_best\_case = equal(begin(best\_case), end(best\_case), begin(expected\_best\_case));

assertTest(result\_best\_case);

// средний случай

// O(n + k \* log(k)), где k — количество ведер

int averege\_case[] = {1, 4, 2, 8, 6, 12, 4};

int expected\_averege\_case[] = {1, 2, 4, 4, 6, 8, 12};

int size\_averege\_case = sizeof(averege\_case) / sizeof(averege\_case[0]);

bucketSort(averege\_case, size\_averege\_case);

bool result\_averege\_case = equal(begin(averege\_case), end(averege\_case), begin(averege\_case));

assertTest(result\_averege\_case);

// худший случай

// O(n^2), все элементы попадают в одно ведро

int worst\_case[] = {12, 12, 12, 12, 12, 12};

int expected\_worst\_case[] = {12, 12, 12, 12, 12, 12};

int size\_worst\_case = sizeof(worst\_case) / sizeof(worst\_case[0]);

bucketSort(worst\_case, size\_worst\_case);

bool result\_worst\_case = equal(begin(worst\_case), end(worst\_case), begin(expected\_worst\_case));

assertTest(result\_worst\_case);

}

int main() {

    int arr[] = {0, 23, 65, 4, 1, 90, 7, 7};

    int size = sizeof(arr) / sizeof(arr[0]);

    bucketSort(arr, size);

    for (int i = 0; i < size; ++i) {

        cout << arr[i] << " ";

    }

    runTest();

    const int count = 10000; // Количество случайных чисел

    vector<int> randomNumbers;

    // Генерация случайных чисел

    generateRandomNumbers(randomNumbers, count);

    // Преобразование std::vector в массив для сортировки

    int\* randomArr = randomNumbers.data();

    // Записываем текущее время работы

    auto start = chrono::high\_resolution\_clock::now();

    bucketSort(randomArr, count);

    // Текущее время окончания выполнения работы

    auto end = chrono::high\_resolution\_clock::now();

    // Продолжительность выполнения работы

    auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

    cout << "Sorting completed." << endl;

    cout << "Running time: " << duration.count() << " microseconds" << endl;

    return 0;

}