ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировки»

Выполнил работу

Филатова Элина

Академическая группа C3100

Принято

Практикант, Вершинин Владислав Константинович

Санкт-Петербург

2024

**Введение**

Цель: исследовать производительность трех алгоритмов сортировки: Bubble sort, Quick sort, Count sort.

Задачи:

Реализовать указанные алгоритмы сортировки на языке C++.

Сравнить время выполнения каждого алгоритма при различных размерах входных данных.

Построить графики зависимости времени выполнения от количества элементов.

Проанализировать результаты и оценить эффективность алгоритмов.

**Теоретическая подготовка**

Для выполнения работы в основном использовался тип данных vector<int> для хранения входного массива данных. А также потребовались следующие алгоритмы:

Bubble sort – это один из наиболее известных алгоритмов, суть которого состоит в последовательном сравнении двух соседних элементов. В том случае, если предыдущий элемент больше последующего, они меняются местами.

Quick sort – это алгоритм, использующийся для работы с большими массивами данных и работает по принципу «разделяй и властвуй».

*Основные шаги алгоритма:*

Из массива выбирается опорный элемент, чаще всего посередине массива.

Другие элементы массива распределяются таким образом, чтобы меньшие размещались до него, а большие — после.

Далее первые шаги рекурсивно применяются к подмассивам, которые разделились опорным элементом на две части — слева и справа от него.

Count sort – это алгоритм сортировки, который работает путем подсчета количества вхождений каждого элемента в массив и использует эту информацию для построения отсортированного массива. Он эффективен для сортировки целых чисел или объектов с ограниченным диапазоном значений.

*Основные шаги алгоритма:*

Нахождение максимального элемента в массиве, чтобы определить размер массива для подсчёта.

Создание массива подсчёта (count), где индекс представляет возможные значения элементов исходного массива, а значение в индексе — это количество вхождений этого числа в массив.

Заполнение массива подсчёта: проходим по исходному массиву и увеличиваем счетчик в массиве подсчёта для каждого встреченного числа.

Воссоздание отсортированного массива: на основе массива подсчёта восстанавливаем отсортированный массив, повторно записывая элементы в порядок, определённый их частотой

**Реализация**

В ходе работы были реализованы три алгоритма сортировки. Каждый алгоритм был помещен в отдельную функцию, а также была реализована функция для генерации случайных массивов размерами 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000 элементов соответственно и замера времени выполнения каждого алгоритма сортировки.

В процессе работы были подключены следующие библиотеки:

<iostream> - используется для ввода и вывода.

<vector> - используется для работы с динамическими массивами (векторами).

<cassert> - используется для тестирования, позволяет проверять условия во время выполнения программы.

<chrono> - предоставляет возможности для работы с временем. Используется для измерения производительности программы.

<cstdlib> - для генерации случайных чисел.

Реализация функций:

***Пузырьковая сортировка:***

Сначала объявляются функции *BubbleSort, IsSorted, GenerateRandomArray, RunTests* для выполнения функции main, которой реализуется инициализация генератора случайных чисел с помощью текущего времени, запуск тестов для проверки корректности сортировки, определение размеров массивов для тестирования сортировки, генерация случайных массивов и измерение времени, необходимого для их сортировки с помощью пузырьковой сортировки.

*Функция BubbleSort* реализует стандартный алгоритм пузырьковой сортировки:

Используются два вложенных цикла: внешний цикл проходит по элементам массива, а внутренний цикл сравнивает соседние элементы.

Если элементы находятся в неправильном порядке (первый больше второго), они меняются местами.

С помощью флага swapped отслеживается, были ли произведены обмены. Если за проход не было обменов, сортировка завершается, так как массив уже отсортирован.

*Генерация случайного массива* с помощью функции GenerateRandomArray:

Данная функция создает массив случайных чисел от 0 до 999 заданного размера, используя функцию rand().

*Замер времени*:

Для замера времени выполнения сортировки используется chrono::high\_resolution\_clock, который позволяет получать высокоточные временные метрики. Время выполнения сортировки для массивов указанного размера выводится в микросекундах.

*Тестирование*:

Функция *RunTests* проверяет работу функции сортировки на трех случаях:

Лучший случай: массив уже отсортирован.

Худший случай: массив отсортирован в обратном порядке.

Средний случай: случайный массив, проверяется совместимость с сортировкой.

***Сортировка подсчетом:***

Аналогично предыдущей программе в начале объявляются используемые функции.

*Функция main():*

Устанавливает случайное начальное значение для генерации случайных чисел.

Вызывает функцию *RunTests()* для запуска тестов.

Генерирует случайные массивы заданного размера и сортирует их с помощью *CountingSort*, измеряя при этом время выполнения.

*Функция CountingSort:*

Эта функция принимает вектор arr и сортирует его.

Сначала определяется максимальное и минимальное значение в массиве, чтобы установить диапазон значений.

Создается вспомогательный вектор count, который используется для хранения количества вхождений каждого числа в массиве.

Сначала подсчитывается количество вхождений каждого числа, затем массив сортируется, заполняя его на основе этого количества.

Как и в предыдущей программе используются функция генерации случайного массива (*GenerateRandomArray*) и функция *IsSorted*, которая проверяет правильно ли отсортирован массив.

*Тестирование:*

В этой функции проверяются различные сценарии сортировки:

Лучший случай: уже отсортированный массив.

Худший случай: массив, отсортированный в обратном порядке.

Средний случай: произвольный массив чисел.

Массив с дубликатами: массив с повторяющимися элементами.

Для каждого случая выполняется сортировка и проверка на отсортированность с помощью assert

***Быстрая сортировка:***

Как и в предыдущих двух программах в данной также в начале объявляются функции, используемые впоследствии в функции *main().*

Рассмотрим реализацию функции самого алгоритма сортировки *QuickSort* и функции *partition*.

*partition*:

Устанавливает опорный элемент (последний элемент массива).

Проходит по массиву и перемещает элементы, меньше или равные опорному элементу влево.

В конечном итоге помещает опорный элемент на его правильное место и возвращает его индекс.

*QuickSort:*

Принимает в качестве аргументов ссылку на массив, индекс начала и конца.

Если низкий индекс меньше высокого, выполняется разбиение:

*partition* возвращает индекс опорного элемента.

Рекурсивно вызывается *QuickSort* для частей массива, полученных после разбиения.

**Экспериментальная часть**

Подсчет используемой памяти для каждого алгоритма:

*Алгоритм сортировки пузырьком*:

vector<int>& arr: Массив передаётся по ссылке, то есть он не копируется, и вся память, которая используется для хранения данных, зависит от количества элементов в массиве: следовательно, весь массив займет 4n байт.

int n: 4 байта

int i: 4 байта

int j: 4 байта

bool swapped: 1 байт

Итого: 4n + 4+4+4+1 = 4n + 13 байт. Таким образом, основная память, которую занимает алгоритм, является константной и не зависит от размера массива, что делает его O(1) по пространственной сложности.

*Алгоритм быстрой сортировки*:

В функции partition:

Размер массива: 4k байт

int pivot: 4 байта

int i: 4 байта

int j: 4 байта

В функции QuickSort:

int pivotIndex: 4 байта

Память для рекурсивного стека: Максимальная глубина рекурсии в худшем случае может быть O(n), что приведёт к потреблению O(n) памяти в стеке. В лучшем случае (если массив сбалансирован) глубина рекурсии будет O(logn) и потребление памяти в стеке составит O(logn).

Итого: 4k + 12n байт в худшем случае; 4k+12(logn) байт в лучшем случае.

*Алгоритм сортировки подсчетом*:

Для vector<int>count: 24 + 4n байт, где n = maxVal – minVal +1

maxVal: 4 байта

minVal: 4 байта

range: 4 байта

num: 4 байта

Итого: 24+4n+16 = 40 + 4n байт. Таким образом, Пространственная сложность Counting Sort составляет O(n), где n - диапазон значений (разница между максимальным и минимальным элементами)

Подсчет асимптотики для каждого алгоритма:

Теоретически сложность Bubble Sort составляет O(N^2) в худшем и среднем случаях, так как внешний цикл выполняется (n-1) раз (где n - количество элементов в массиве), а внутренний цикл в самом худшем случае (массив отсортирован в обратном порядке) будет выполнятся от (n-1) до 0. То есть общее количество итераций можно рассчитать по формуле:

А значит итоговая временная сложность будет O(n^2)

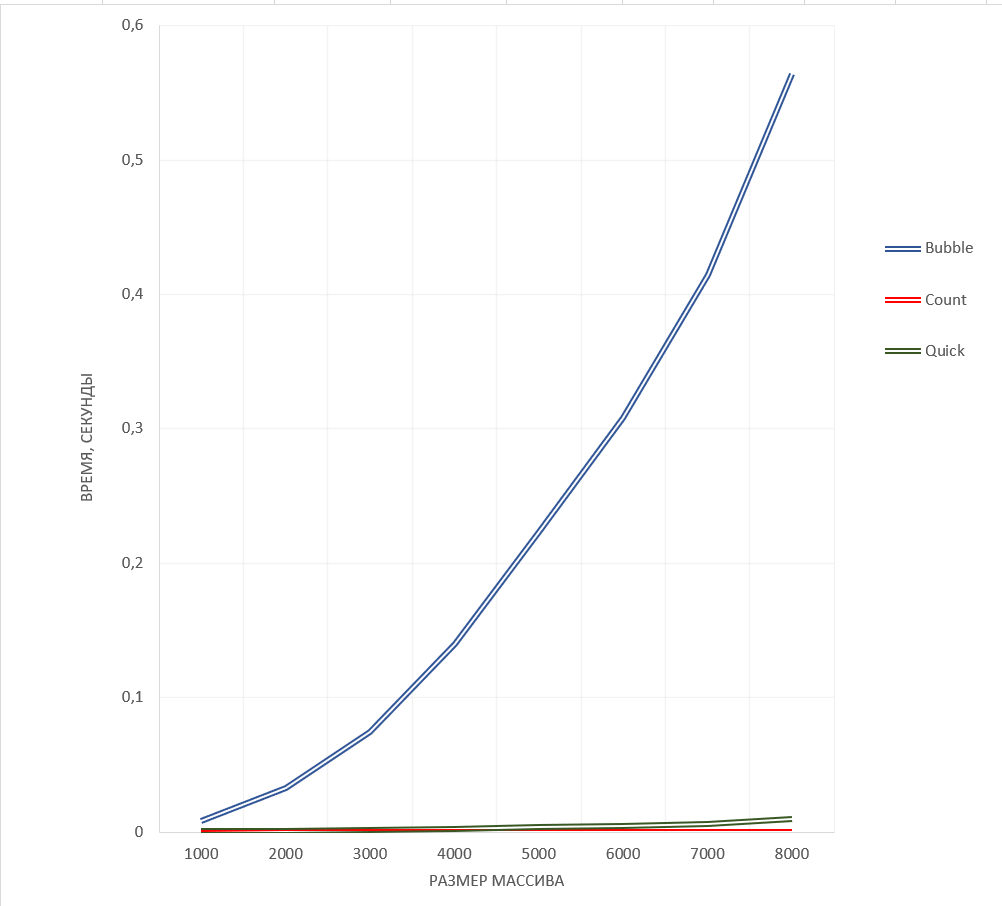
Теоретически сложность Quick sort составит O(N^2). Худший случай: массив делится крайне неравномерно, например, на подмассивы размера 1 и (n−1). Глубина рекурсии составит O(n), и на каждом уровне будет выполняться O(n) работы. Суммарная сложность: O(n^2). В лучшем случае массив будет делиться на равные части и глубина рекурсии будет О(logn), а на каждом уровне выполняется О(n) работы (вызов partition). Таким образом, суммарная сложность в лучшем случае будет O(nlongn)

Сложность Count Sort составит O(N\*k), так как поиск максимального и минимального значения займет O(n), где n – количество элементов в массиве, а инициализация массива count займет O(k), где k – диапазон значений исходного массива.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размер | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 |
| Bubble sort | 0,008669 | 0,032837 | 0,074167 | 0,140105 | 0,223065 | 0,308158 | 0,414815 | 0,563912 |
| Count sort | 0,000022 | 0,000035 | 0,000069 | 0,000076 | 0,000086 | 0,000395 | 0,000116 | 0,000145 |
| Quick sort | 0,000511 | 0,000784 | 0,001542 | 0,002565 | 0,004028 | 0,004869 | 0,006203 | 0,00946 |

Таблица №1 - Подсчёт сложности реализованного алгоритма

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

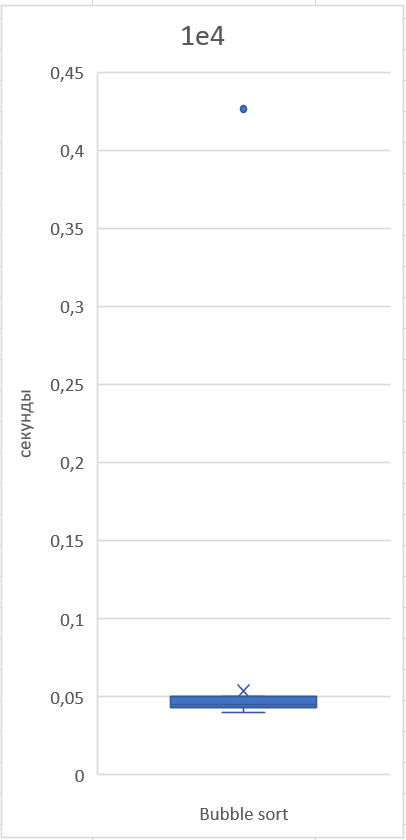


Изображение №1 - График работы алгоритмов

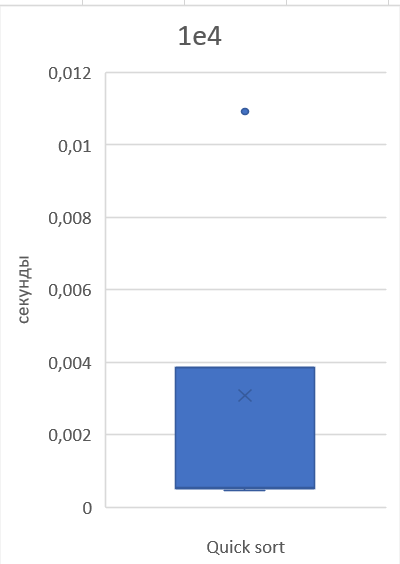
Согласно теоретическим оценкам: Bubble Sort, Quick Sort имеют сложность O(N^2). Результаты эксперимента подтверждают это: время выполнения значительно увеличивается с ростом размера входного массива. Count Sort, имея сложность O(N\*k), демонстрирует значительно более быстрое выполнение по сравнению с другими двумя алгоритмами, что соответствует теоретической оценке.

Также для оценки эффективности алгоритмов были проведены замеры времени выполнения соответствующих алгоритмов для двух наборов данных: с 10000 (1е4) и 100000 (1е5) элементов. Для каждого набора данных было выполнено 50 запусков каждого алгоритма, что обеспечило статистическую значимость результатов.

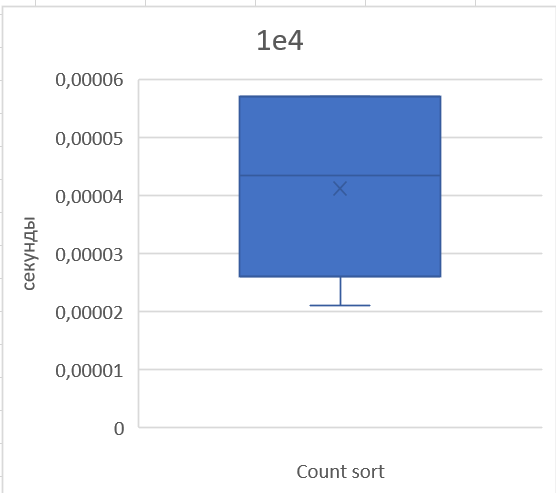
Box plot графики представлены на изображениях №2, №3, №4, №5, №6 соответственно.



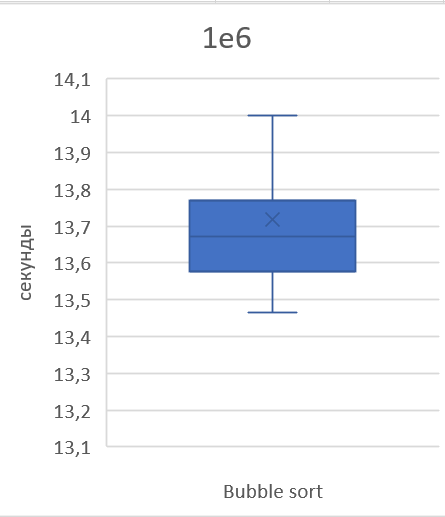
Изображение №2 – График работы Bubble sort для 10000 элементов



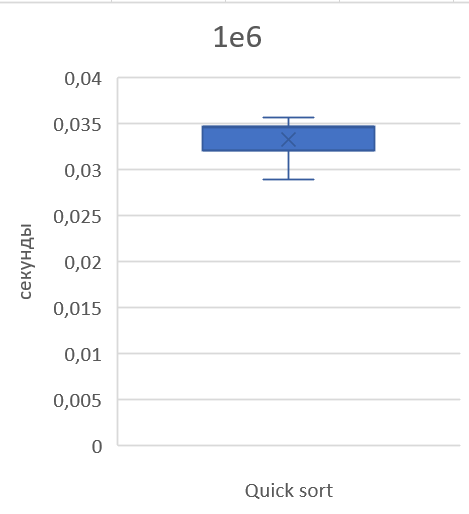
Изображение №3 – График работы Quick sort для 10000 элементов



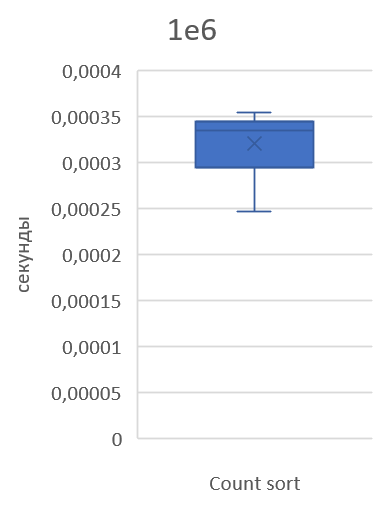
Изображение №4 – График работы Count sort для 10000 элементов



Изображение №5 – График работы Bubble sort для 100000 элементов



Изображение №6 – График работы Quick sort для 100000 элементов



Изображение №6 – График работы Quick sort для 100000 элементов

Как можно было заметить, на графиках box plot возникают выбросы даже при одном размере массива, что имеет несколько причин. Например,

состояние системы в момент запуска теста также может влиять на результаты. Если в момент тестирования система перегружена, это может привести к увеличению времени выполнения. Фоновые процессы, такие как обновления, антивирусные сканирования и другие задачи, могут также влиять на производительность. Даже если размер массива одинаковый, его содержимое может значительно влиять на время выполнения алгоритма. Например, алгоритмы сортировки могут работать быстрее на уже отсортированных данных и медленнее на случайных или обратных данных.

**Заключение**

В этой работе были рассмотрены три алгоритма сортировки: сортировка пузырьком, сортировка подсчетом и быстрая сортировка. Реализация алгоритмов подтвердила теоретические ожидания по использованию памяти и времени в зависимости от размера массива. Стоит отметить, что выбор алгоритма сортировки должен основываться на конкретных требованиях задачи: размерах данных, диапазоне значений, наличии дубликатов и итд. Сортировка пузырьком может быть полезна в учебных целях и для очень малых массивов, сортировка подсчетом оптимальна для известного и ограниченного диапазона значений, а быстрая сортировка предоставляет гибкое и эффективное решение для большинства практических случаев. Каждой из данных техник следует применять с учетом их особенностей для достижения оптимальных результатов.

**ПРИЛОЖЕНИЕ А**

Листинг кода файла bubble\_sort.cpp

#include <iostream>

#include <vector>

#include <chrono> // Для замера времени

#include <cassert>

#include <cstdlib> // Для генерации случайных чисел

using namespace std;

void BubbleSort(vector<int>& arr);

void PrintArr(const vector<int>& arr);

bool IsSorted(const vector<int>& arr);

vector<int> GenerateRandomArray(int size);

void RunTests();

int main() {

srand(time(0)); //получает начальное значение (семя) - текущее время (в сек с начала эпохи Unix (1 января 1970 года))

//для генерации псевдослучайных чисел

// Запуск тестов

RunTests();

// Замеры времени на различных размерах массивов

vector<int> sizes = {10000,100000};

for (int size : sizes) {

// Генерация случайного массива

vector<int> randomArray = GenerateRandomArray(size);

// Сообщение о начале теста

cout << "Bubble sorting array of size: " << size << endl;

// Замер времени

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

BubbleSort(randomArray);

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

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

cout << "Time taken: " << duration << " microseconds" << endl;

}

return 0;

}

// Функция пузырьковой сортировки

void BubbleSort(vector<int>& arr) {

int n = arr.size(); // 4 байта

for (int i = 0; i < n - 1; i++) { //4 байта

bool swapped = false; // 1 байт

for (int j = 0; j < n - i - 1; j++) {//4 байта

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

swapped = true; // Устанавливаем флаг, если произошел обмен/ //1 байт

}

}

if (!swapped) break; // Если не было обменов, массив уже отсортирован

}

}

// Функция для вывода массива

void PrintArr(const vector<int>& arr) {

for (const int& i : arr) {

cout << i << " ";

}

cout << endl;

}

// Проверка, отсортирован ли массив

bool IsSorted(const vector<int>& arr) {

int n = arr.size();

for (int i = 1; i < n; i++) {

if (arr[i - 1] > arr[i]) return false;

}

return true;

}

// Генерация случайного массива заданного размера

vector<int> GenerateRandomArray(int size) {

vector<int> arr(size);//создаем вектор нужного размера

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

arr[i] = rand() % 1000; //rand() используется для генерации случайных целых чисел (здесь генерируем числа от 0 до 99)

}

return arr;

}

// Функция для выполнения тестов

void RunTests() {

// Лучший случай: отсортированный массив

vector<int> bestCase = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

BubbleSort(bestCase);

assert(IsSorted(bestCase));

cout << "Best case sorted successfully." << endl;

// Худший случай: отсортированный в обратном порядке

vector<int> worstCase = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

BubbleSort(worstCase);

assert(IsSorted(worstCase));

cout << "Worst case sorted successfully." << endl;

// Средний случай: случайный массив

vector<int> averageCase = { 5, 3, 8, 6, 2, 7, 4, 10, 1, 9 };

BubbleSort(averageCase);

assert(IsSorted(averageCase));

cout << "Average case sorted successfully." << endl;

}

**ПРИЛОЖЕНИЕ В**

Листинг кода файла quick\_sort.cpp

#include <iostream>

#include <vector>

#include <cassert>

#include <cstdlib>

#include <chrono>

using namespace std;

void QuickSort(vector<int>& arr, int low, int high);

bool IsSorted(const vector<int>& arr);

vector<int> GenerateRandomArray(int size);

void RunTests();

int main() {

srand(time(0));

//

RunTests();

//

vector<int>sizes = { 10000,100000 };

for (int size : sizes) {

vector<int> randomArray = GenerateRandomArray(size);

cout << "Quick sorting array of size: " << size << endl;

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

QuickSort(randomArray, 0, size - 1);

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

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

cout << "Time taken: " << duration << " microseconds" << endl;

}

return 0;

}

int partition(vector<int>& arr, int low, int high) {// 4+4

int pivot = arr[high];//4

int i = low - 1;//4

for (int j = low; j < high; j++) {//4

if (arr[j] <= arr[high]) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void QuickSort(vector<int>& arr, int low, int high) {//4+4

if (low < high) {

int pivotIndex = partition(arr, low, high);//4

QuickSort(arr, low, pivotIndex - 1);

QuickSort(arr, pivotIndex + 1, high);

}

}

bool IsSorted(const vector<int>& arr) {

int n = arr.size();

for (int i = 1; i < n; i++) {

if (arr[i - 1] > arr[i])

return false;

}

return true;

}

vector<int> GenerateRandomArray(int size) {

vector<int>arr(size);

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

arr[i] = rand() % 100; //генерируем числа от 0 до 99

}

return arr;

}

void RunTests() {

// Лучший случай : отсортированный массив

vector<int> bestCase = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

QuickSort(bestCase, 0, bestCase.size() - 1);

assert(IsSorted(bestCase));

cout << "Best case sorted successfully." << endl;

// Худший случай: отсортированный в обратном порядке

vector<int> worstCase = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

QuickSort(worstCase, 0, worstCase.size() - 1);

assert(IsSorted(worstCase));

cout << "Worst case sorted successfully." << endl;

// Средний случай: случайный массив

vector<int> averageCase = { 5, 3, 8, 6, 2, 7, 4, 10, 1, 9 };

QuickSort(averageCase, 0, averageCase.size() - 1);

assert(IsSorted(averageCase));

cout << "Average case sorted successfully." << endl;

// Дополнительный случай: массив с дубликатами

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

QuickSort(duplicates, 0, duplicates.size() - 1);

assert(IsSorted(duplicates));

cout << "Duplicates sorted successfully." << endl;

}

**ПРИЛОЖЕНИЕ С**

Листинг кода файла count\_sort.cpp

#include <iostream>

#include <vector>

#include <cassert>

#include <cstdlib>

#include <chrono>

using namespace std;

bool IsSorted(const vector<int>& arr);

vector<int> GenerateRandomArray(int size);

void CountingSort(vector<int>& arr);

void RunTests();

int main() {

srand(time(0));

RunTests();

vector<int> sizes = { 10000,100000 };

for (int size : sizes) {

vector<int> randomArray = GenerateRandomArray(size);

cout << "Count sorting array of size: " << size << endl;

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

CountingSort(randomArray);

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

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

cout << "Time taken: " << duration << " microseconds" << endl;

}

return 0;

}

void CountingSort(vector<int>& arr) {

if (arr.empty()) return;

int maxVal = arr[0];//4

int minVal = arr[0];//4

for (int num : arr) {

if (num > maxVal) maxVal = num;

if (num < minVal) minVal = num;

}

int range = maxVal - minVal + 1;//4

std::vector<int> count(range, 0);

for (int num : arr) {

count[num - minVal]++;

}

int index = 0;

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

while (count[i] > 0) {

arr[index++] = i + minVal;

count[i]--;

}

}

}

vector<int> GenerateRandomArray(int size) {

vector<int> arr(size);

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

arr[i] = rand() % 100;

}

return arr;

}

bool IsSorted(const vector<int>& arr) {

for (int i = 1; i < arr.size(); i++) {

if (arr[i - 1] > arr[i])

return false;

}

return true;

}

void RunTests() {

vector<int> bestCase = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

CountingSort(bestCase);

assert(IsSorted(bestCase));

cout << "Best case sorted successfully." << endl;

vector<int> worstCase = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

CountingSort(worstCase);

assert(IsSorted(worstCase));

cout << "Worst case sorted successfully." << endl;

vector<int> averageCase = { 5, 3, 8, 6, 2, 7, 4, 10, 1, 9 };

CountingSort(averageCase);

assert(IsSorted(averageCase));

cout << "Average case sorted successfully." << endl;

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

CountingSort(duplicates);

assert(IsSorted(duplicates));

cout << "Duplicates sorted successfully." << endl;

}