ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Реализация алгоритмов сортировки вставками, расческой и подсчетом»

Выполнил работу

Адриановская Александрв

Академическая группа C3100

Принято

Магистр практики Вершинин Владислав

Санкт-Петербург

2024

Введение

Цель лабораторной работы – реализация трех выбранных алгоритмов сортировки числовых массивов размерностью до 1e6 целочисленных элементов. Алгоритмы относятся к разным классам сложности. В процессе работы необходимо сравнить скорости работы алгоритмов между собой в зависимости от количества элементов в сортируемом массиве. Также целью работы является построение графического представления полученной зависимости для получения выводов, которые должны практически подтвердить теоретические ожидания.

1. Теоретическая подготовка

Для выполнения работы необходимо обладать теоретической подготовкой в части понимания принципов сортировки числовых массивов различными способами, работы с циклами и условными операторами, работы с файловым вводом/выводом, юнит-тестирования функций, оценки длительности работы алгоритма. 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, cassert для проведения юнит тестирования, vector для работы со структурой данных типа вектор, fstream для файлового ввода/вывода.

2.1 Реализация алгоритма сортировки вставками

Используются 2 вложенных цикла. Внешний цикл в прямую сторону по всем элементам массива, кроме первого и внутренний цикл в обратную сторону от текущего выбранного во внешнем цикле до нулевого элемента. Дополнительным условием завершения цикла является достижение отсортированной области, идущей с начала массива. В теле цикла производится обмен значениями сравниваемых элементов массива.

vector<int> insertionSort(vector<int> array) {

    int n = array.size(); //int = 4 bytes, O(1)

    for(int i = 1; i < n; i++) { //int = 4 bytes, O(1) + O(1) + O(1) \* O(N)

        for(int j = i; j > 0 && array[j - 1] > array[j]; j--) { //int = 4 bytes, 6 \* O(1) \* среднее кол-во итераций N/2 => O(N)

            // меняем местами элементы

            swap(array[j - 1], array[j]);//O(1) \* 5

        }

    }

    //Сложность алгоритма O(N^2) =  3 \* O(1) \* O(N) \* ((6 + 5) \* O(1) \* O(N/2))

    //доп память только на вспомогательные переменные, перестановка на месте. Пространственная сложность O(1)

    return array;

}

2.2 Реализация алгоритма сортировки расческой

Используются 2 вложенных цикла. Внешний цикл итерируется по параметру step, который логарифмически уменьшается от итерации к итерации в фиксированное число 1,247. Считается, что это наилучшим образом подходящее значение для реализации алгоритма сортировки расческой. До запуска цикла в значение переменной step помещается n - количество элементов массива. Во внутреннем цикле происходит перебор элементов начиная с начала массива до элемента с индексом n – step. В теле цикла происходит сравнение элементов, отстоящих друг от друга на величину step и обмен значениями сравниваемых элементов массива при необходимости перестановки.

vector<int> combSort(vector<int> array) {

    int n = array.size(); //int = 4 bytes, O(1)

    int step = n; //int = 4 bytes, O(1)

    bool flag = false; //int = 1 byte, O(1)

    int num = 0; //int = 4 bytes, O(1)

    while (step > 1 || flag) { //O(1) \* 3

        if (step > 1) {// O(1)

            step = step \* 1000 / 1247; // O(1) \* 3

        }//O(logN) шаг уменьшается в 1,247 раз

        // cout << step << " ";

        flag = false; // O(1)

        int i = 0; //int = 4 bytes, O(1)

        while (i + step < n) { //O(1) \* 2  сложность цикла линейная O(N), среднее значение меньше N/2

            if (array[i] > array[i + step]) { //O(1) \* 3

                flag = true; //O(1)

                swap(array[i], array[i + step]); //O(1) \* 5

            }

            i++; //O(1)

        }

        // num++;

        // cout << step << " ";

    }

    //Сложность алгоритма O(N\*logN) =  6 \* O(1) \* O(logN) \* ((6 + 5) \* O(1) \* O(N/2))

    //доп память только на вспомогательные переменные, перестановка на месте. Пространственная сложность O(1)

    return array;

}

2.3 Реализация алгоритма сортировки подсчетом

Для реализации алгоритма сортировки подсчетом необходимо пройтись в цикле по исходному массиву для накопления количества встречающихся значений в накопительном массиве. Далее осуществляется проход по накопительному массиву и наполнение исходного массива значением, равным текущему индексу накопительного массива в количестве, равном текущему значению в накопительном массиве.

vector<int> countingSort(vector<int> array) {

    const int K = 1e5;// 4 byte O(1)

    vector<int> c(K+1);// K \* 4 byte + 24 byte Vector

    int n = array.size(); // 4 byte O(1)

    for (int i = 0; i < n; i++) // 4 byte once, (O(1) + O(1)) \* O(N)

        c[array[i]]++; // O(1) + O(1) + O(1)

    int b = 0;// 4 byte O(1)

    for (int i = 0; i <= K; i++) {// 4 byte once, (O(1) + O(1)) \* O(K)

        while (c[i] != 0) { //O(1) + O(1)

            array[b] = i; // Суммарно O(N)

            b++; //O(1)

            c[i]--; //O(1) + O(1)

        }

    }

    //Сложность алгоритма O(K\*N) =  5 \* O(1) \* O(N) + (4 \* O(1) \* O(K) \* 3 \* O(1) \* O(N))

    //доп память на вектор для подсчета частоты + вспомогательные переменные. Пространственная сложность O(K)

    return array;

}

1. Экспериментальная часть

Подсчёт по памяти.

Алгоритмы сортировки вставками и расческой осуществляют сортировку на месте, таким образом, память расходуется только на исходный массив N \* 4 байта, где N < 1000000 и на вспомогательные переменные. Сортировка подсчетом требует вспомогательного массива размерностью K, значение которого было задано равным 100000. Таким образом потребление памяти для такого массива составляет 4 \* 100000.

Подсчёт асимптотики.

Алгоритм сортировки вставками использует два вложенных цикла. Внешний цикл проводит N итераций, внутренний цикл N/2. Таким образом, сложность этого алгоритма O(N^2).

Сортировка расческой несмотря на то, что имеет в своем составе также 2 вложенных цикла, имеет сложность O(NlogN). Достигается это за счет изменения текущего шага сравнения в фиксированное количество раз.

Сортировка подсчетом имеет линейную асимптотику размера N \* K поскольку последовательно проходится по двум массивам – исходному размерности N и вспомогательному размерности K.

График зависимости времени от числа элементов сортируемого массива.

Согласно требованиям задания необходимо построить график зависимости времени работы алгоритмов сортировки от количества элементов массива от 1000 до 1e6 с шагом 1000.

С помощью генератора датасетов последовательно создавались массивы, включающие элементы от 0 до 1e5 и оценивались времена выполнения трех разработанных алгоритмов сортировки. Результат фиксировался в текстовом файле, который в последствие был использован для построения графика. Поскольку алгоритмы имеют различную сложность, с ростом размерности входного массива, время выполнения значительно отличалось. При достижении размера массива 1e5, алгоритм вставками достиг времени выполнения 20 секунд. Продолжать дальнейшую оценку через 1000 значений было бы крайне накладно по времени, поэтому были рассчитаны опорные точки через каждые 50000 элементов массива. Достигнув значение 600000, время выполнения составило 750 секунд и дальнейшее увеличение было нецелесообразным. Для проверки ранее вынесенного заключения о сложности алгоритма O(N^2) по точке N=100000 были рассчитаны параметры параболы T = a \* N ^ 2. Значение a составило 0,0000020748. Используя этот коэффициент с помощью MS Excel был построен график теоретический зависимости времени выполнения сортировки от значения N. Ранее полученные экспериментальные значения для размера массива > 100000 имеют максимальное отклонение от расчетного значения равное 1 %, что подтверждает корректность проведенного предположения и расчета. Экспериментальные и теоретические данные были сведены в таблицу по которой построен график зависимости. Для алгоритма сортировки вставками была выбрана вспомогательная ось значений.

График представляющий зависимость экспериментальных и теоретических значений времени работы алгоритма представлен на изображении №1.

Изображение №1 - время работы алгоритмов сортировки

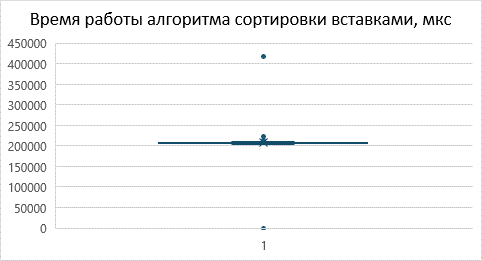
Зависимость времени работы реализованных алгоритмов от числа элементов в массиве соответствует теоретической оценке сложности O(N^2), O(N\*logN) и O(K\*N).

Также в задании требовалось провести оценку времени работы алгоритмов для 50 различных массивов для каждой из размерностей 1e4 и 1e5 и отобразить на графиках формата boxplot.

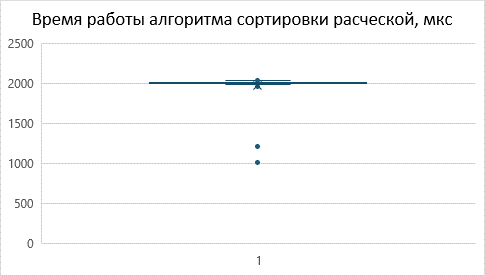
Основная часть массивов была сформирована с помощью генератора датасетов. Также для корректности оценки были добавлены массивы, представляющие из себя крайние случаи –отсортированный массив и обратно сортированный массив. Результаты приведены на изображениях 2 – 7.

Для алгоритма сортировки вставками скорость работы зависит от того, насколько исходный массив предварительно отсортирован. Для сортированного массива время работы алгоритма минимально и составляет 0.5 мс даже для размера массива 1e6. Для несортированного массива – наоборот, максимально.

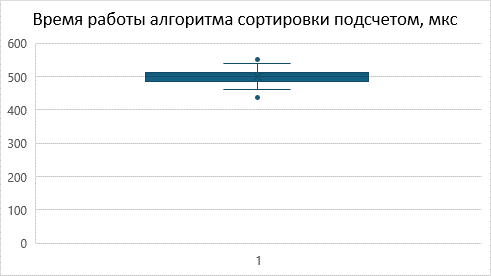
Алгоритм сортировки расческой и подсчетом наоборот показывают лучшие скорости на отсортированном в любом направлении массиве.



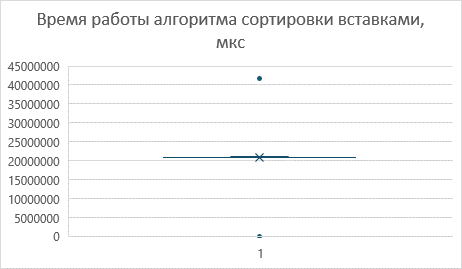
Изображение №2 - время работы алгоритма сортировки вставками при N = 10000



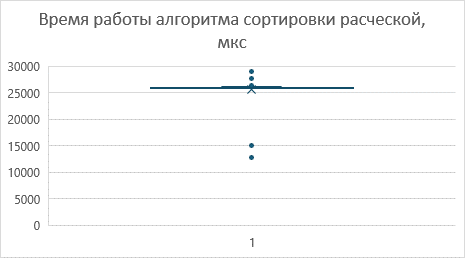
Изображение №3 - время работы алгоритма сортировки расческой при N = 10000



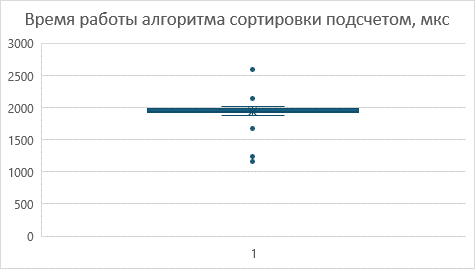
Изображение №4 - время работы алгоритма сортировки подсчетом при N = 10000



Изображение №5 - время работы алгоритма сортировки вставками при N = 100000



Изображение №6 - время работы алгоритма сортировки расческой при N = 100000



Изображение №7 - время работы алгоритма сортировки подсчетом при N = 100000

Заключение

В ходе выполнения работы мною были реализованы три алгоритмы сортировки – вставками, расческой и подсчетом. Цель работы была достигнута путём тестирования на массивах с различным числом и комбинациями входных значений. В результате было показано совпадение экспериментальных результатов работы алгоритмов сортировки с расчетными. Также была проведена оценка зависимости времени работы алгоритмов сортировки для крайних значений (отсортированный и обратно сортированный ряд) и для выборки из 50 повторений.

ПРИЛОЖЕНИЕ А

Листинг кода файла lab5.cpp

#include <iostream>

#include <vector>

#include <cassert>

#include <fstream>

#include <chrono>

using namespace std;

vector<int> insertionSort(vector<int> array) {

    int n = array.size(); //int = 4 bytes, O(1)

    for(int i = 1; i < n; i++) { //int = 4 bytes, O(1) + O(1) + O(1) \* O(N)

        for(int j = i; j > 0 && array[j - 1] > array[j]; j--) { //int = 4 bytes, 6 \* O(1) \* среднее кол-во итераций N/2 => O(N)

            // меняем местами элементы

            swap(array[j - 1], array[j]);//O(1) \* 5

        }

    }

    //Сложность алгоритма O(N^2) =  3 \* O(1) \* O(N) \* ((6 + 5) \* O(1) \* O(N/2))

    //доп память только на вспомогательные переменные, перестановка на месте. Пространственная сложность O(1)

    return array;

}

/\*

//Bubble sort для сравнения с CombSort

vector<int> bubbleSort(vector<int> array) {

    int n = array.size(); //int = 4 bytes, O(1)

    for (int i = 0; i < n; i++) {//int = 4 bytes, O(1) + O(1) + O(1) \* O(N)

        for (int j = 0; j < n - 1; j++) { //int = 4 bytes, O(1) + O(1) + O(1) \* O(N)

            if (array[j] > array[j + 1]) {//O(1) \* 3

                // меняем местами элементы

                swap(array[j + 1], array[j]);//O(1) \* 5

            }

        }

        //O(N^2)

    }

    return array;

}

\*/

vector<int> combSort(vector<int> array) {

    int n = array.size(); //int = 4 bytes, O(1)

    int step = n; //int = 4 bytes, O(1)

    bool flag = false; //int = 1 byte, O(1)

    int num = 0; //int = 4 bytes, O(1)

    while (step > 1 || flag) { //O(1) \* 3

        if (step > 1) {// O(1)

            step = step \* 1000 / 1247; // O(1) \* 3

        }//O(logN) шаг уменьшается в 1,247 раз

        // cout << step << " ";

        flag = false; // O(1)

        int i = 0; //int = 4 bytes, O(1)

        while (i + step < n) { //O(1) \* 2  сложность цикла линейная O(N), среднее значение меньше N/2

            if (array[i] > array[i + step]) { //O(1) \* 3

                flag = true; //O(1)

                swap(array[i], array[i + step]); //O(1) \* 5

            }

            i++; //O(1)

        }

        // num++;

        // cout << step << " ";

    }

    //Сложность алгоритма O(N\*logN) =  6 \* O(1) \* O(logN) \* ((6 + 5) \* O(1) \* O(N/2))

    //доп память только на вспомогательные переменные, перестановка на месте. Пространственная сложность O(1)

    return array;

}

vector<int> countingSort(vector<int> array) {

    const int K = 1e5;// 4 byte O(1)

    vector<int> c(K+1);// K \* 4 byte + 24 byte Vector

    int n = array.size(); // 4 byte O(1)

    for (int i = 0; i < n; i++) // 4 byte once, (O(1) + O(1)) \* O(N)

        c[array[i]]++; // O(1) + O(1) + O(1)

    int b = 0;// 4 byte O(1)

    for (int i = 0; i <= K; i++) {// 4 byte once, (O(1) + O(1)) \* O(K)

        while (c[i] != 0) { //O(1) + O(1)

            array[b] = i; // Суммарно O(N)

            b++; //O(1)

            c[i]--; //O(1) + O(1)

        }

    }

    //Сложность алгоритма O(K\*N) =  5 \* O(1) \* O(N) + (4 \* O(1) \* O(K) \* 3 \* O(1) \* O(N))

    //доп память на вектор для подсчета частоты + вспомогательные переменные. Пространственная сложность O(K)

    return array;

}

vector<int> readDataset(string filename) {

    vector<int> input\_array;

    // Открываем файл

    ifstream in(filename.c\_str());  // O(1) - открытие файла

    // Считываем следующую строку из файла до конца.

    int number; // O(1) - создание переменной

    while(in >> number) { // O(N) - цикл while

        input\_array.push\_back(number); // O(1) - добавление строки в вектор

    }

    in.close(); //O(1) - закрытие файла

    return(input\_array);

}

//Тесты cassert

void test\_sorts()

{

    vector<int> input\_1 = {9, 0, 11, 10, 3, 2};

    vector<int> expected\_output\_1 = {0, 2, 3, 9, 10, 11};

    vector<int> result\_1\_1 = insertionSort(input\_1);

    vector<int> result\_1\_2 = combSort(input\_1);

    vector<int> result\_1\_3 = countingSort(input\_1);

    assert(

        result\_1\_1 == expected\_output\_1 &&

        result\_1\_2 == expected\_output\_1 &&

        result\_1\_3 == expected\_output\_1

        );

    cout << "test 1 is successful" << endl;

    vector<int> input\_2 = {9, 8, 3, 2, 1, 0};

    vector<int> expected\_output\_2 = {0, 1, 2, 3, 8, 9};

    vector<int> result\_2\_1 = insertionSort(input\_2);

    vector<int> result\_2\_2 = combSort(input\_2);

    vector<int> result\_2\_3 = countingSort(input\_2);

    assert(

        result\_2\_1 == expected\_output\_2 &&

        result\_2\_2 == expected\_output\_2 &&

        result\_2\_3 == expected\_output\_2

        );

    cout << "test 2 is successful" << endl;

    vector<int> input\_3 = {1, 1, 1, 1, 1, 1, 1, 1};

    vector<int> expected\_output\_3 = {1, 1, 1, 1, 1, 1, 1, 1};

    vector<int> result\_3\_1 = insertionSort(input\_3);

    vector<int> result\_3\_2 = combSort(input\_3);

    vector<int> result\_3\_3 = countingSort(input\_3);

    assert(

        result\_3\_1 == expected\_output\_3 &&

        result\_3\_2 == expected\_output\_3 &&

        result\_3\_3 == expected\_output\_3

        );

    cout << "test 3 is successful" << endl;

    vector<int> input\_4 = {};

    vector<int> expected\_output\_4 = {};

    vector<int> result\_4\_1 = insertionSort(input\_4);

    vector<int> result\_4\_2 = combSort(input\_4);

    vector<int> result\_4\_3 = countingSort(input\_4);

    assert(

        result\_4\_1 == expected\_output\_4 &&

        result\_4\_2 == expected\_output\_4 &&

        result\_4\_3 == expected\_output\_4

        );

    cout << "test 4 is successful" << endl;

}

int main() {

    vector<int> input\_array = readDataset("dataset.txt"); //O(1) - создание вектора

    int arr\_size = input\_array.size();

    auto start1 = chrono::high\_resolution\_clock::now();

    vector<int> result1 = insertionSort(input\_array);

    auto start2 = chrono::high\_resolution\_clock::now();

    vector<int> result2 = combSort(input\_array);

    auto start3 = chrono::high\_resolution\_clock::now();

    vector<int> result3 = countingSort(input\_array);

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

    auto duration1 = chrono::duration\_cast<chrono::milliseconds>(start2 - start1);

    auto duration2 = chrono::duration\_cast<chrono::milliseconds>(start3 - start2);

    auto duration3 = chrono::duration\_cast<chrono::milliseconds>(endall - start3);

    ofstream out;

    out.open("result.txt", ios\_base::app);// O(1) - открытие файла

    out << arr\_size << ";" << duration1.count() << ";" << duration2.count() << ";" << duration3.count() << endl;

    out.close(); // O(1) - закрытие файла

    // test\_sorts();

    return 0;

}