ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Сортировки»

Выполнил работу

Воробьев Егор

Академическая группа J3113

Принято

Ассистент, Дунаев Максим

Санкт-Петербург

2024

1. Введение

Цель: Подобрать и реализовать алгоритмы сортировок по определенным требованиям.

Задачи:

1) Реализовать алгоритм сортировки с лучшей сложностью - O(N^2) и пространственной сложностью - O(1).
2) Реализовать алгоритм сортировки со средней сложностью до O(N^2) и пространственной сложностью - до O(N).
3) Реализовать алгоритм сортировки со средней сложностью O(N*k) и пространственной сложностью - до O(N*k), где k << N (значительно меньше N).
4) Протестировать и сравнить алгоритмы на различных размерах массива

2. Теоретическая подготовка

В качестве первого алогритма выберем Gnome Sort. Он похож на сортировку вставками, но отличается перемещением элементов.

В качестве второго алгоритма выберем Tree Sort. Он использует бинарное дерево для упорядочивания элементов.

В качестве третьего алгоритма выберем Counting Sort. Это один из алгоритмов сортировки с линейной сложностью, и его время работы зависит от диапазона значений, а не от количества элементов в массиве.

Типы данных:

1) std::vector – для реализации основого массива, в котором будем находить подмассивы;
2) std::string – для считывания входных данных из файла поэлементно.

Заголовочные файлы:

1) <vector> - для использования типа данных std::vector;
2) <string> - для использования типа данных std::string;
3) <fstream> - для считывания входных данных из файла;
4) <iostream> - для вывода отсортированных массивов;
5) <chrono> - для измерения времени выполнения алгоритма.

3. Реализация

1) Считывание входных данных из файла “dataset.txt” с помощью функции getline, которая на каждой итерации цикла while записывает элементы в переменную element типа std::string до знака “;”, и массива array типа std::vector, в который на каждой итерации добавляется ранее полученный элемент, конвертированный из типа std::string в тип int с помощью функции stoi. Цикл while работает до тех пор, пока функция getline может считывать данные.

```
std::ifstream data_set("dataset.txt");

std::string element; // 32 байта
std::vector<int> array; // 24 байта

while (getline(data_set, element, ';')) {
    array.push_back(stoi(element));
} //+4*n байт в vector, где n - количество элементов в нем
```

Изображение №1 – Считывание входных данных

2) Алгоритм Gnome Sort работает путем сравнения элементов, начиная с первого, и если текущий элемент меньше предыдущего, они меняются местами, иначе алгоритм двигается дальше.
Если нужно переместиться на предыдущий элемент, индекс сдвигается назад.
Процесс продолжается, пока не достигнем конца массива или пока элементы не будут упорядочены.

```
void gnomeSort(std::vector<int>& array) {
    int index = 0;
    int n = array.size();

    while (index < n) {
        if (index == 0 || array[index] >= array[index - 1]) {
            index++;
        } else {
            std::swap(array[index], array[index - 1]);
            index--;
        }
    }
}
```

Изображение №2 – Реализация Gnome Sort.

3) В Tree Sort каждый элемент массива вставляется в бинарное дерево поиска. Все элементы в левом поддереве меньше его корня. Все элементы в правом поддереве больше его корня. После того как все элементы вставлены в дерево, производится обход дерева в порядке возрастания (in-order traversal)**.** При этом элементы посещаются в отсортированном порядке и поэтому массив будет отсортирован.

```
void treeSort(std::vector<int>& arr) {
    TreeNode* root = nullptr;

    for (int value : arr) {
        root = insert(root, value);
    }

    arr.clear();
    inOrderTraversal(root, arr);
} //итого вставка каждого элемента (N штук) за O(logN), то есть получаем O(NlogN
```

Изображение №3 – Реализация Tree Sort.

Рассмотрим подробнее два важных шага:

1. **Вставка в дерево**:
   Для каждого элемента массива создается новый узел дерева. Элемент сравнивается с текущим узлом дерева. Если элемент меньше текущего узла, переходим в левое поддерево. Если элемент больше текущего узла, переходим в правое поддерево.
   Это продолжается до тех пор, пока не найдём пустое место для вставки.

```
TreeNode* insert(TreeNode* root, int value) { //вставка за O(logN)
    if (root == nullptr) {
        return new TreeNode(value);
    }
    if (value < root->value) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}
```

Изображение №4 – Реализация вставки элемента в дерево

2. **Обход дерева в порядке возрастания (in-order traversal)**:
   После того как все элементы вставлены в дерево, мы начинаем обход дерева. Сначала рекурсивно обходим левое поддерево. Затем посещаем текущий узел (элемент). И наконец рекурсивно обходим правое поддерево.
   Во время обхода элементы добавляются в новый массив, который будет отсортирован.

```
void inOrderTraversal(TreeNode* root, std::vector<int>& sortedArray) { //обход з
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left, sortedArray);
    sortedArray.push_back(root->value);
    inOrderTraversal(root->right, sortedArray);
}
```

Изображение №5 – Реализация обхода дерева

4) Counting Sort работает путем подсчёта числа вхождений каждого элемента в массив, а затем использования этой информации для формирования отсортированного массива. Принцип действия алгоритмы заключается в следующем:

1. **Определение диапазона значений**: Нужно определить минимальное и максимальное значение в массиве. Это необходимо, чтобы знать диапазон значений, которые нужно учитывать при подсчёте.

```
int maxVal = *std::max_element(arr.begin(), arr.end()); //4 байт
int minVal = *std::min_element(arr.begin(), arr.end()); //4 байт
```

Изображение №6 – Определение минимального и максимального элементов

2. **Создание массива подсчёта**: Создаётся вспомогательный массив для подсчёта количества вхождений каждого элемента. Индексы этого массива будут соответствовать значениям в исходном массиве, а значения в массиве счётчиков — количеству вхождений каждого числа.

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

std::vector<int> count(range, 0); //8k байт
```

Изображение №7 – Вычисление диапазона и создание массива подсчета

3. **Подсчёт вхождений**: Для каждого элемента исходного массива увеличиваем соответствующую ячейку в массиве подсчёта.

```
for (int num : arr) { //перебираем все элементы массива - O(N)
    count[num - minVal]++;
}
```

Изображение №8 – Подсчет вхождений каждого элемента

4. **Формирование отсортированного массива**: Пробегаем массив подсчёта и для каждого элемента, который встречается k раз, записываем его k раз в отсортированный массив.

```
int index = 0; //4 байт
for (int i = 0; i < range; ++i) { //перебираем k значений, где k - диапазон значений массив
    while (count[i] > 0) {
        arr[index++] = i + minVal;
        count[i]--;
    }
}
```

Изображение №9 – Сортировка массива

5) Далее измеряем время выполнения функции, рассмотренной в прошлом пункте. Перед вызовом функции фиксируем стартовое время с помощью функции std::chrono::high_resolution_clock::now, которая возвращает текущее время. Затем вызываем функцию сортировки и после ее выполнения еще раз фиксируем время. Теперь вычисляем само время выполнения в секундах и выводим его.

4. Экспериментальная часть

1) Теоретически, сложность Gnome Sort ≈ O(N^2).

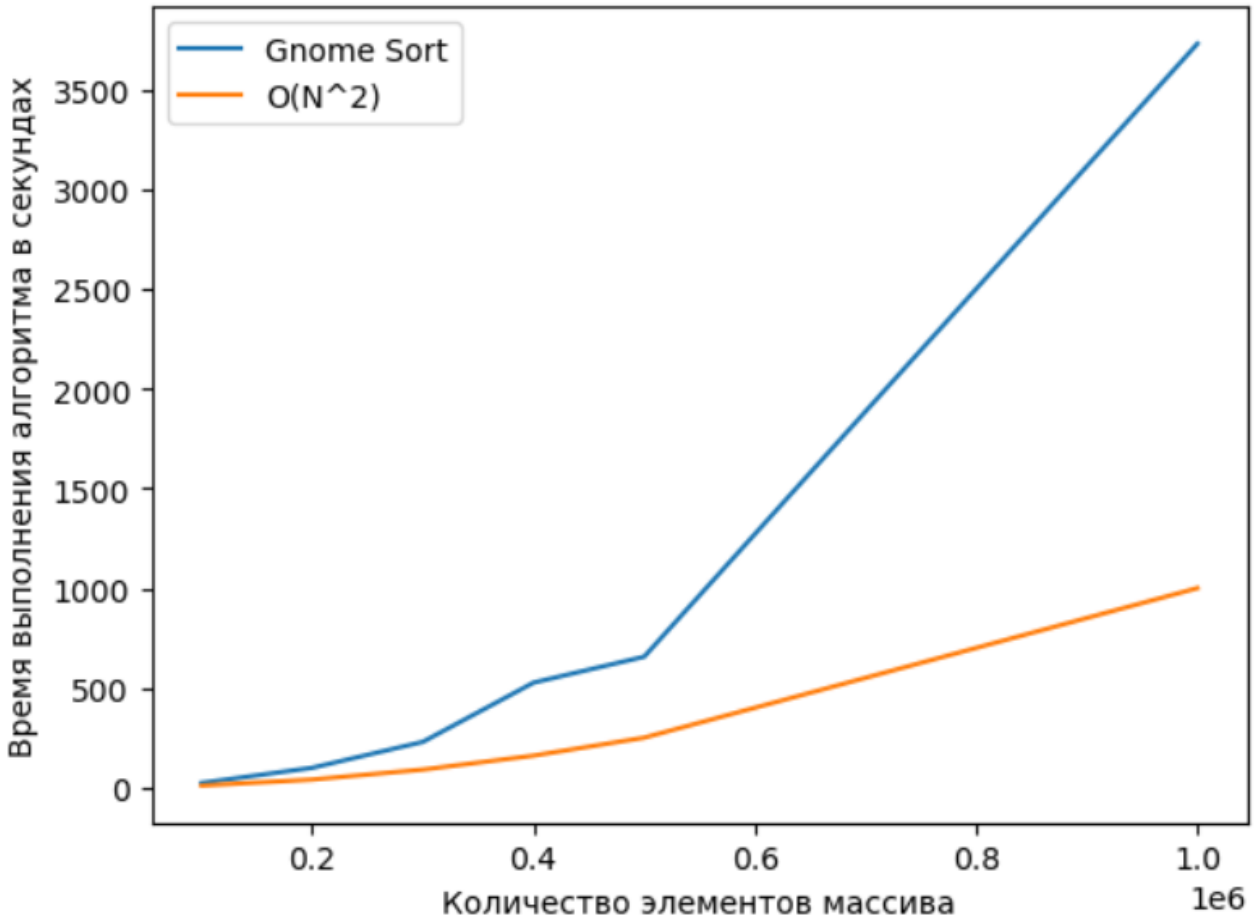
Для определения эффективности алгоритма, было измерено его время выполнения на разных размерах массива. Собранная статистика отображена в таблице №1.

Таблица №1 – Время выполнения алгоритма при различных размерах массива

N (количество элементов массива)	Gnome Sort	O(N^2)
100000	23	10
200000	99	40

300000	229	90
400000	526	160
500000	656	250
1000000	3730	1000

Ниже предоставлен график, визуализирующий данные из таблицы №1.



Изображение №10 – График зависимости времени выполенения алгоритма от размера массива и нотации, близкой к нашей

Как видно из графика, сложность алгоритма точно больше O(N^2), но в то же время точно меньше O(N^3), поэтому сложность находится между.

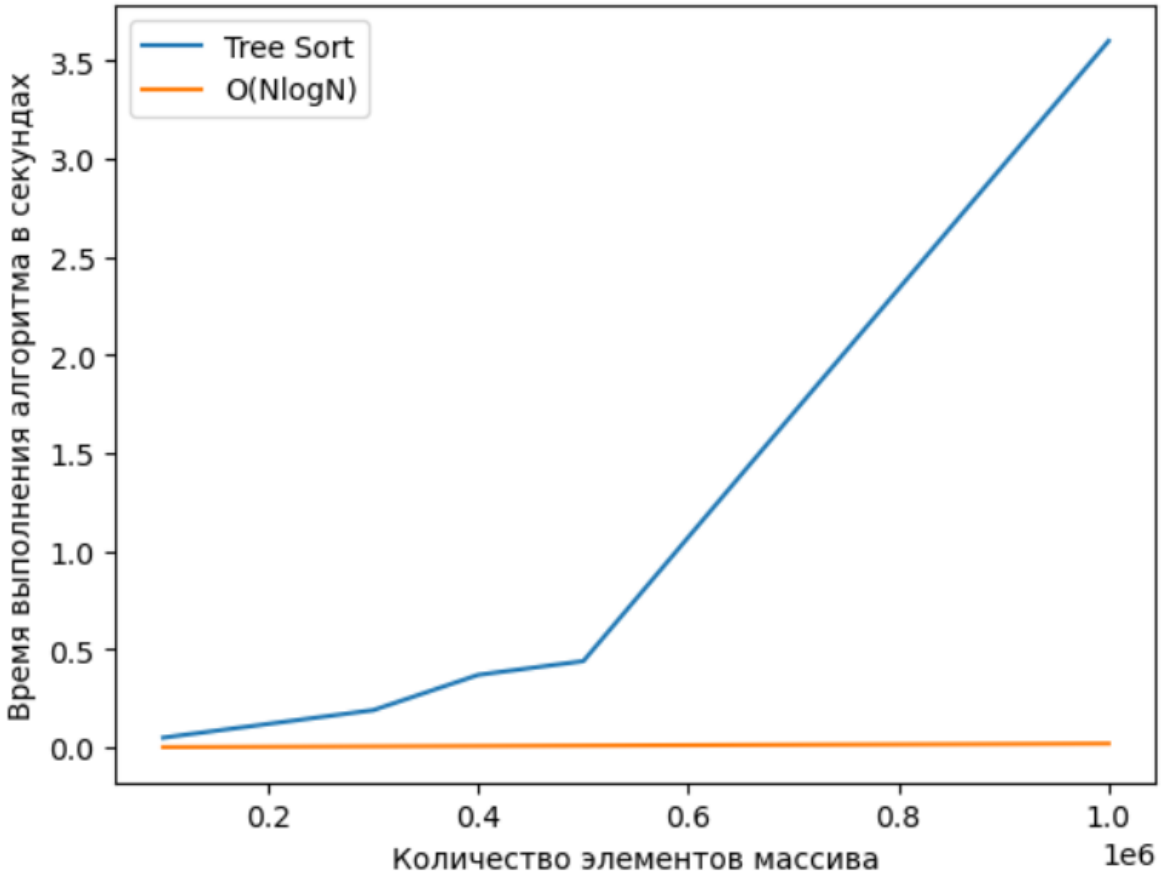
2) Теоретически, сложность Tree Sort = O(N*logN).

   Для определения эффективности алгоритма, было измерено его время выполнения на разных размерах массива. Собранная статистика отображена в таблице №2.

Таблица №2 – Время выполнения алгоритма при различных размерах массива

N (количество элементов массива)	Tree Sort	O(N*logN)
100000	0.05	0.001
200000	0.12	0.003
300000	0.19	0.005
400000	0.37	0.007
500000	0.44	0.009
1000000	3.6	0.02

Ниже предоставлен график, визуализирующий данные из таблицы №2.



Изображение №11 – График зависимости времени выполенения алгоритма от размера массива и нотации, близкой к нашей

Из графика видно, что сложность сортировки точно больше O(N*logN), но находится достаточно близко. Время выполнения в разы меньше, чем у Gnome Sort. Время выполнения Tree Sort может колебаться от O(N*logN) до O(N^2). Оно зависит от сбалансированности дерева, то есть от количества элементов в левых и правых поддеревьях. Чем сильнее разница, тем больше будет время выполнения.

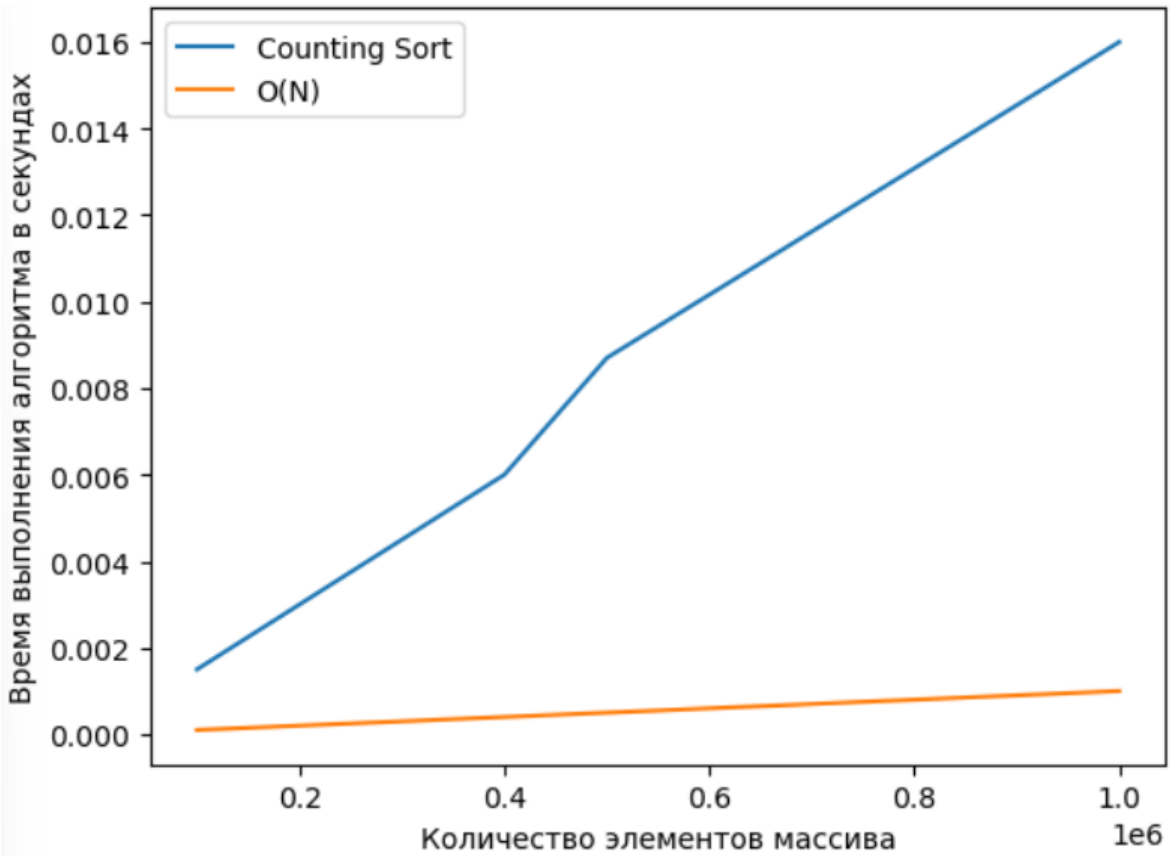
3) Теоретически, сложность Counting Sort = O(N+k), где k – диапазон значений.

   Для определения эффективности алгоритма, было измерено его время выполнения на разных размерах массива. Собранная статистика отображена в таблице №3.

   Таблица №3 – Время выполнения алгоритма при различных размерах массива

N (количество элементов массива)	Counting Sort	O(N)
100000	0.05	0.0001
200000	0.12	0.0002
300000	0.19	0.0003
400000	0.37	0.0004
500000	0.44	0.0005
1000000	3.6	0.0010

Ниже предоставлен график, визуализирующий данные из таблицы №3.



Изображение №12 – График зависимости времени выполенения алгоритма от размера массива и нотации, близкой к нашей

Из графика видно, что время выполнения точно больше O(N) и находится не слишком далеко от него. Прибавление k как раз и повлияло. Этот алгоритм сильно зависит от разницы количества элементов в массиве и диапазона.

5. Заключение

В ходе выполнения работы мною был реализованы 3 алгоритма сортировки по определенным параметрам. Цель работы была достигнута путём тестирования алгоритма на разных входных данных. Полученные результаты примерно равны теоретическим оценкам сложности алгоритма.

6. Приложения

ПРИЛОЖЕНИЕ 1

Основной код алгоритма Gnome Sort

```cpp
void gnomeSort(std::vector<int>& array) {
    int index = 0;
    int n = array.size();

    while (index < n) {
        if (index == 0 || array[index] >= array[index - 1]) {
            index++;
        } else {
            std::swap(array[index], array[index - 1]);
            index--;
        }
    }
}

int main() {
    std::ifstream data_set("dataset.txt");

    std::string element; // 32 байта
    std::vector<int> array; // 24 байта

    while (getline(data_set, element, ';')) {
        array.push_back(stoi(element));
    } //+4*n байт в vector, где n - количество элементов в нем

    std::cout << "Original array: ";
    for (int num : array) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    auto start = std::chrono::high_resolution_clock::now();

    gnomeSort(array);

    auto stop = std::chrono::high_resolution_clock::now();

    std::cout << "Sorted array: ";
    for (int num : array) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    auto duration = std::chrono::duration_cast <std::chrono::seconds>(stop - start);
    data set.close();
```

ПРИЛОЖЕНИЕ 2

Основной код алгоритма Tree Sort

```cpp
struct TreeNode {
    int value; //4 байт
    TreeNode* left; //8 байт
    TreeNode* right; //8 байт

    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
}; //всего в дереве n элементов, то есть оно занимает N*(4+8+8) = 20N байт

TreeNode* insert(TreeNode* root, int value) { //вставка за O(logN)
    if (root == nullptr) {
        return new TreeNode(value);
    }
    if (value < root->value) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}

void inOrderTraversal(TreeNode* root, std::vector<int>& sortedArray) { //обход за O(n)
    if (root == nullptr) {
        return;
    }
    inOrderTraversal(root->left, sortedArray);
    sortedArray.push_back(root->value);
    inOrderTraversal(root->right, sortedArray);
}

void treeSort(std::vector<int>& arr) {
    TreeNode* root = nullptr;

    for (int value : arr) {
        root = insert(root, value);
    }

    arr.clear();
    inOrderTraversal(root, arr);
} //итого вставка каждого элемента (N штук) за O(logN), то есть получаем O(NlogN)
```

ПРИЛОЖЕНИЕ 3

Основной код алгоритма Counting Sort

```
void countingSort(std::vector<int>& arr) {
    if (arr.empty()) return;

    int maxVal = *std::max_element(arr.begin(), arr.end()); //4 байт
    int minVal = *std::min_element(arr.begin(), arr.end()); //4 байт

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

    std::vector<int> count(range, 0); //8k байт

    for (int num : arr) { //перебираем все элементы массива - O(N)
        count[num - minVal]++;
    }

    int index = 0; //4 байт
    for (int i = 0; i < range; ++i) { //перебираем k значений, где k - диапазон значений массива; O(k)
        while (count[i] > 0) {
            arr[index++] = i + minVal;
            count[i]--;
        }
    }
} //итог - O(N + k)

int main() {
    std::ifstream data_set("dataset.txt");

    std::string element; // 32 байта
    std::vector<int> array; // 24 байта

    while (getline(data_set, element, ';')) {
        array.push_back(stoi(element));
    } //+4*n байт в vector, где n - количество элементов в нем

    //std::cout << "Original array: ";
    //for (int num : array) {
    //    std::cout << num << " ";
    //}
    //std::cout << std::endl;

    auto start = std::chrono::high_resolution_clock::now();

    countingSort(array);
```