Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №3 по дисциплине "Анализ алгоритмов"

Тема Трудоемкость сортировок

Студент Шматко К. М.

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватель: Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

В данной лабораторной работе будут рассмотрены алгоритмы сортировок.

Сортировка - это процесс перегруппировки заданной последовательности (кортежа) объектов в некотором определенном порядке [1]. Такой определенный порядок позволяет, в некоторых случаях, эффективнее по времени работать с заданной последовательностью. В частности, одной из целей сортировки является облегчение задачи поиска элемента в отсортированном множестве.

Алгоритмы сортировок можно разбить на три основные части:

— сравнение, определяющее упорядоченность пары элементов;

— перестановку, меняющую местами пару элементов;

— собственно сортирующий алгоритм, который осуществляет сравнение и перестановку элементов данных до тех пор, пока все эти элементы не будут упорядочены (для рекурсивного алгоритма).

Целью данной лабораторной работы является описание и исследование трудоемкости алгоритмов сортировки.

Для достижения поставленной цели необходимо выполнить следующие задачи.

1) Описать алгоритмы сортировок.

2) Создать программное обеспечение, реализующее следующие алгоритмы сортировок:

   — быстрая;

   — «расчёской»;

   — бинарным деревом.

3) Оценить трудоемкости сортировок.

4) Замерить время реализации.

5) Провести анализ затрат работы программы по времени, выяснить влияющие на них характеристики.

1 Аналитическая часть

В этом разделе будут рассмотрены три алгоритма сортировки: быстрая сортировка, сортировка «расчёской» и сортировка с помощью бинарного дерева.

1.1 Быстрая сортировка

Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы [2].

Общая идея алгоритма состоит в следующем.

— Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.

— Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».

— Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

1.2 Сортировка «расчёской»

Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы «причёсываем» массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247. Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы как и в сортировке пузырьком, но такая итерация одна.

Оптимальное значение фактора уменьшения:

$$1.24733... = \frac{1}{1 - e^{-\phi}}, \tag{1.1}$$

где e — основание натурального логарифма, а $\phi = 1.61803...$ — золотое сечение.

1.3 Сортировка с помощью бинарного дерева

Основная идея данной сортировки заключается в построении двоичного дерева поиска по ключам массива, с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Под деревом понимается упорядоченная структура данных, в которой каждому элементу – предшественнику или корню (под)дерева – поставлены в соответствие по крайней мере два других элемента (преемника) [3]. Причем для каждого предшественника выполнено следующее правило: левый преемник всегда меньше, а правый преем-

ник всегда больше или равен предшественнику. Вместо «предшественник» и «преемник» также употребляют термины «родитель» и «сын». Все элементы дерева также называют «узлами».

При добавлении в дерево нового элемента его последовательно сравнивают с нижестоящими узлами, таким образом вставляя на место. Если элемент больше или равен значению корня, то он идет в правое поддерево, где сравниваем его уже с правым сыном, иначе он идет в левое поддерево, где сравниваем с левым, и так далее, пока есть сыновья, с которыми можно сравнить.

Вывод

В данном разделе были рассмотрены три алгоритма сортировки: быстрая сортировка, сортировка «расчёской» и сортировка с помощью бинарного дерева.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов сортировок – быстрая сортировка, сортировка «расчёской» и сортировка с помощью бинарного дерева.

2.1 Разработка алгоритмов

На вход алгоритмов подается массив A.

На рисунке 2.1 представлена схема алгоритма быстрой сортировки. На рисунке 2.2 – схема алгоритма сортировки «расчёской». На рисунках 2.3 – 2.5 представлены схемы алгоритма сортировки с помощью бинарного дерева.

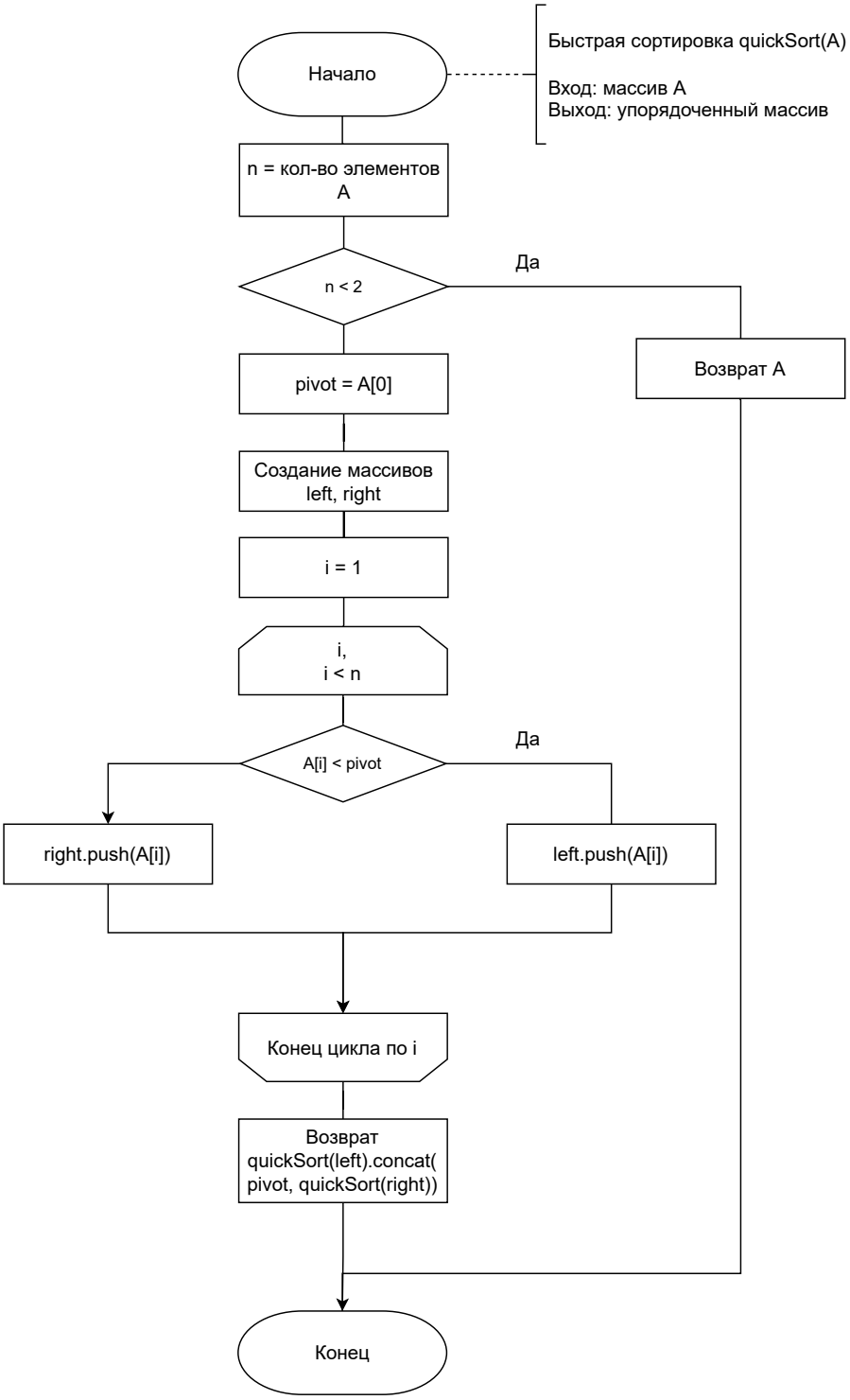

Рисунок 2.1 – Схема алгоритма быстрой сортировки

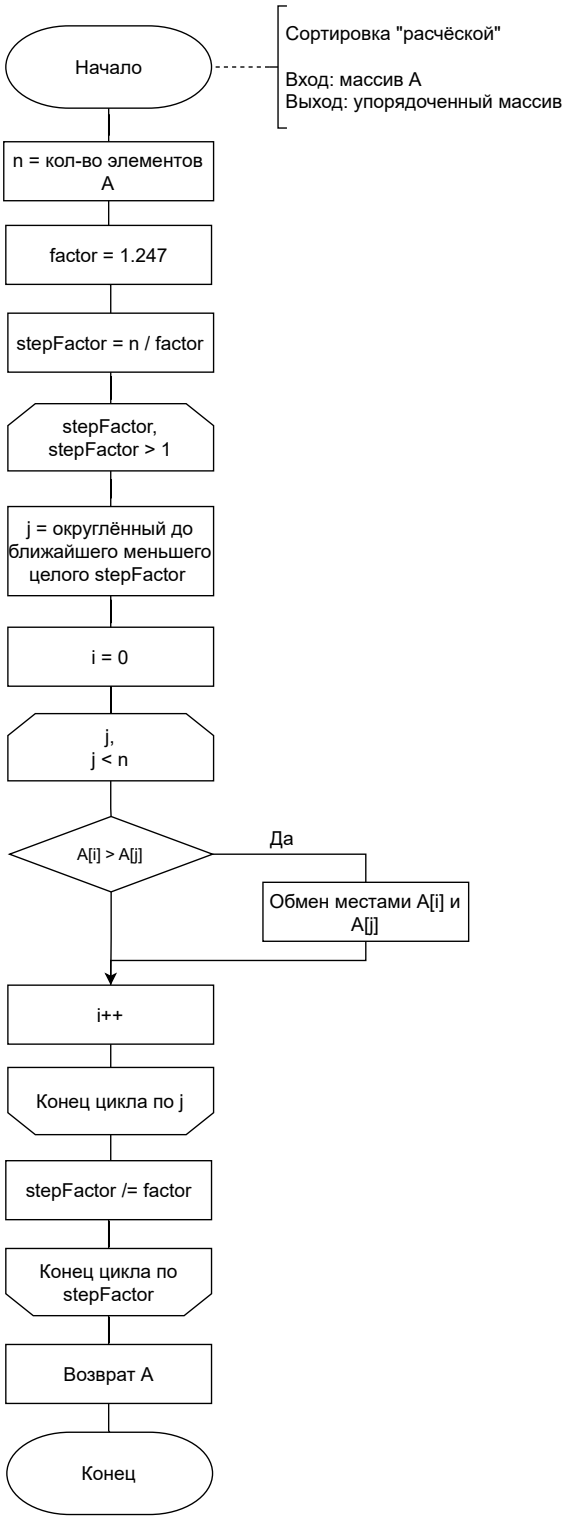

Рисунок 2.2 – Схема алгоритма сортировки «расчёской»

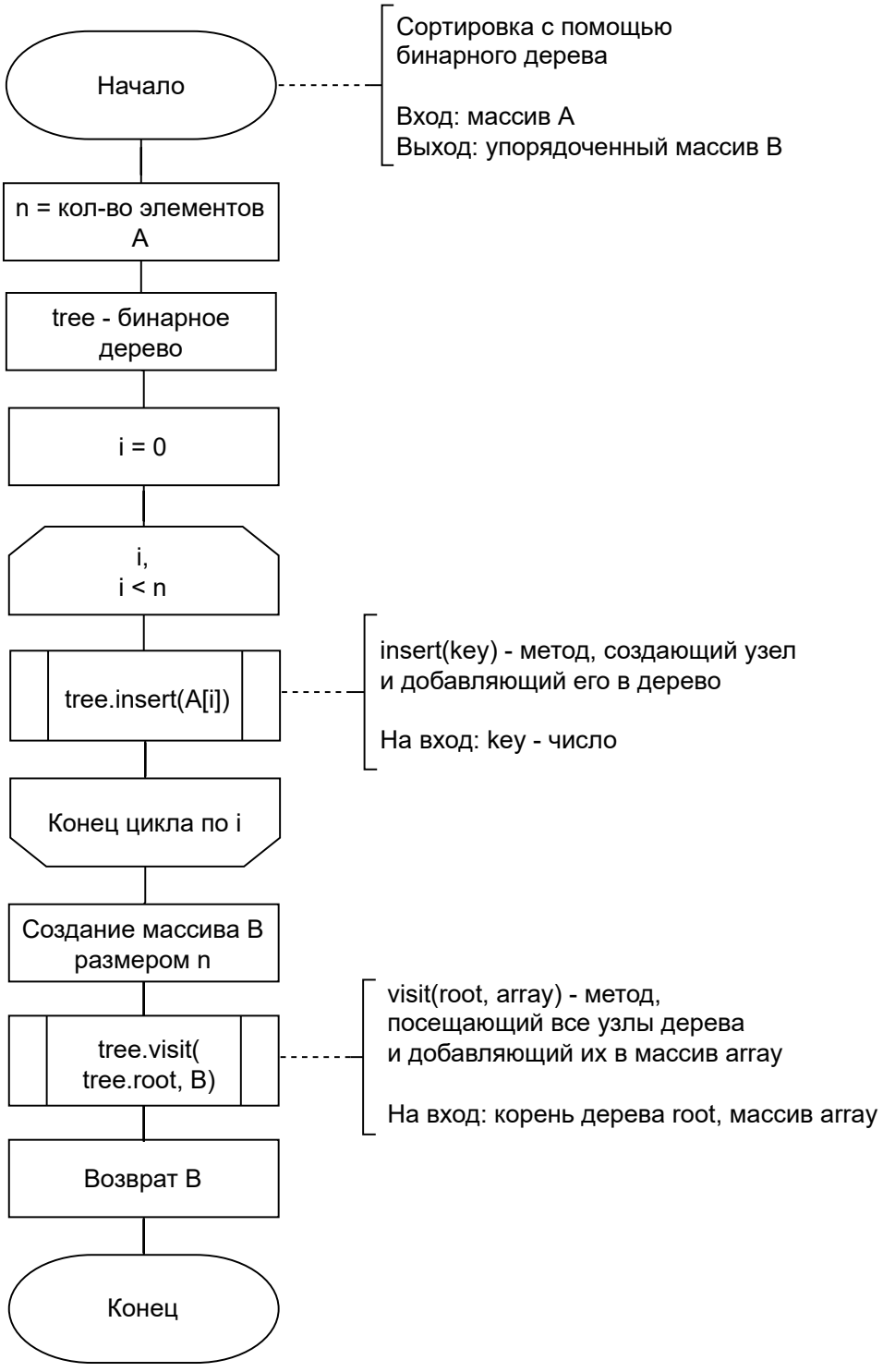

Рисунок 2.3 – Схема алгоритма сортировки с помощью бинарного дерева (Часть 1)

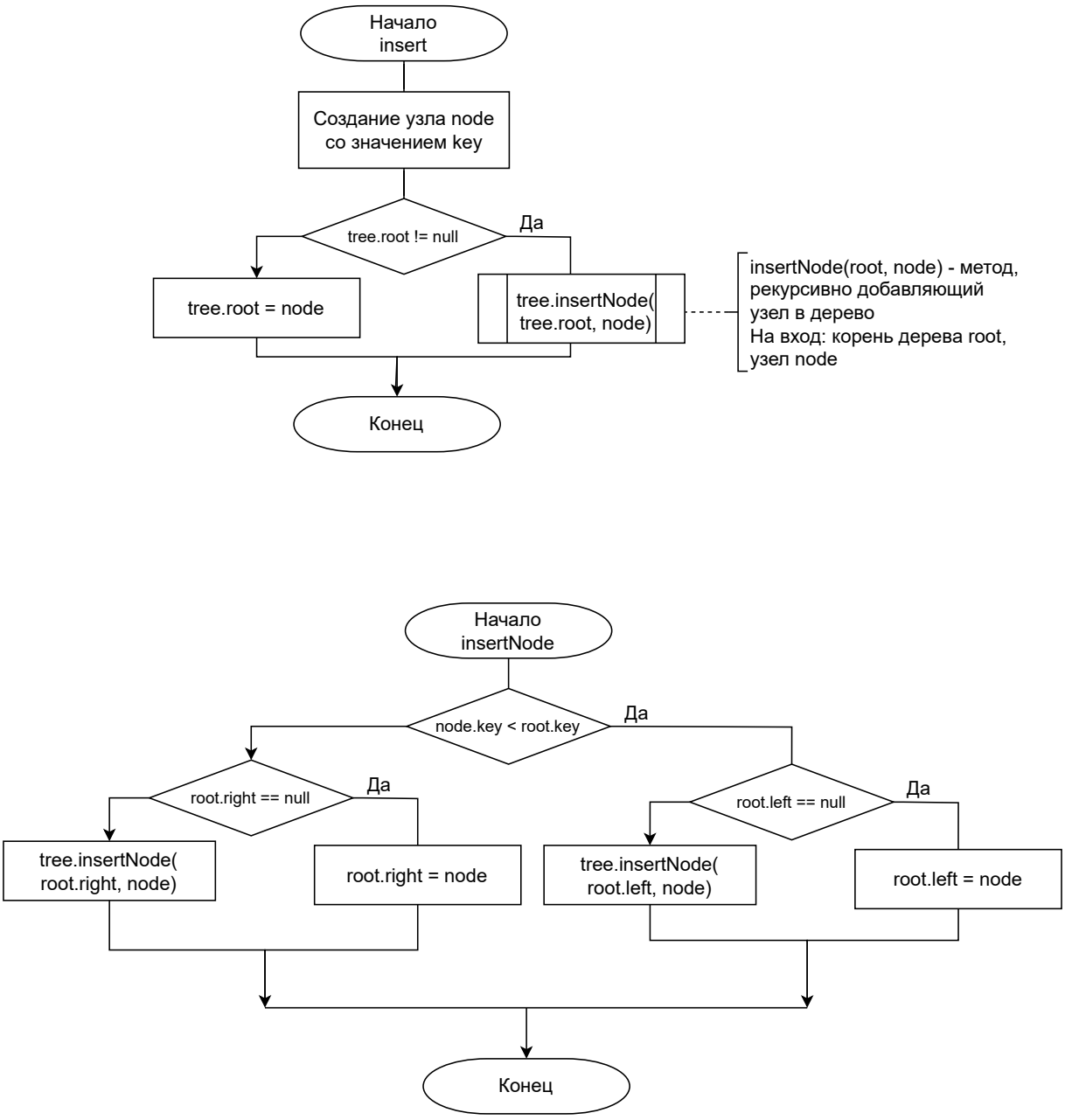

Рисунок 2.4 – Схема алгоритма сортировки с помощью бинарного дерева (Часть 2)

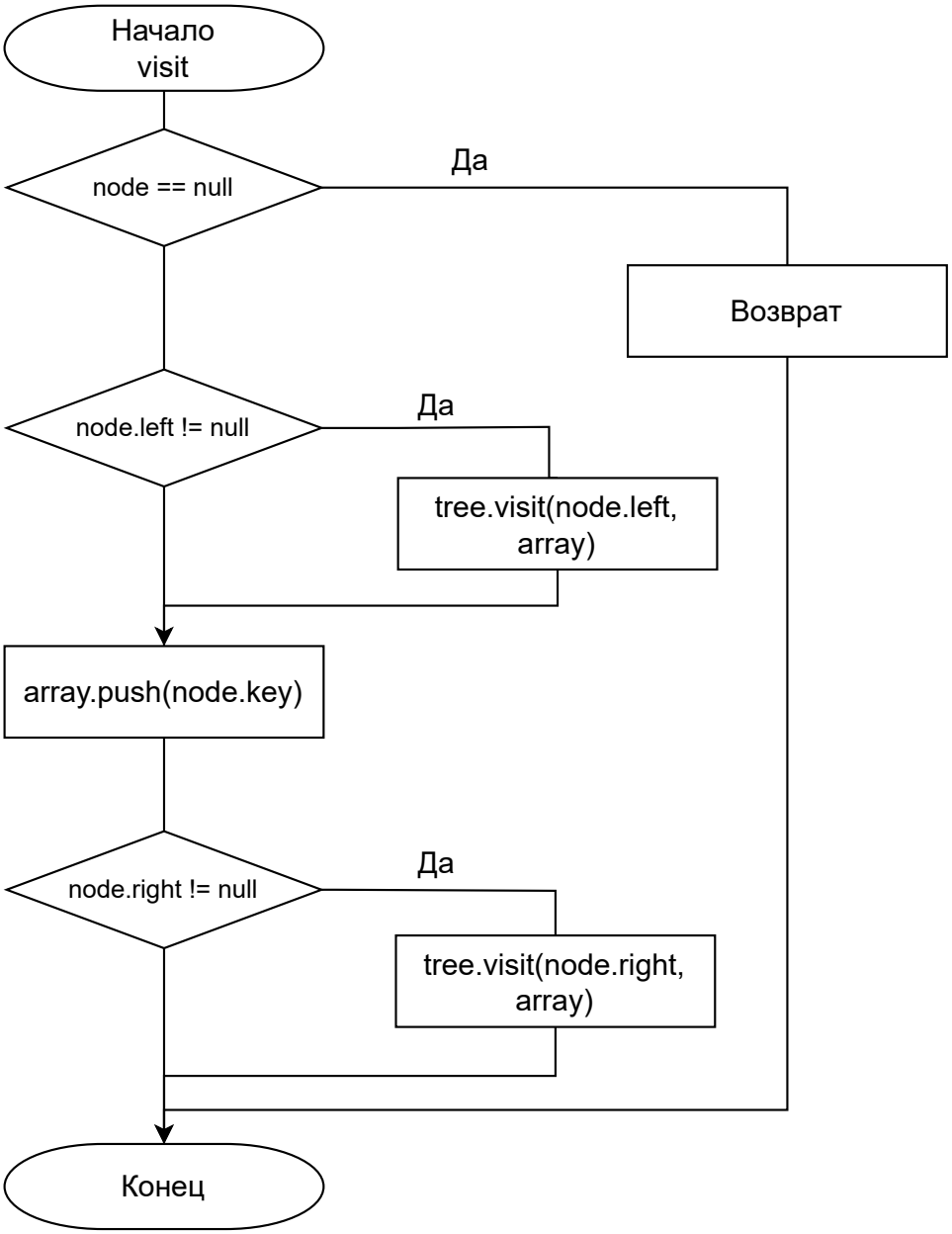

Рисунок 2.5 – Схема алгоритма сортировки с помощью бинарного дерева (Часть 3)

2.2 Модель вычислений для проведения оценки трудоемкости

Введем модель вычислений, которая потребуется для определения трудоемкости каждого отдельного взятого алгоритма сортировки.

1) Трудоемкость базовых операций имеет:

   — равную 1:

$$+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], ++, --, \\ \&\&, >>, <<, ||, \&, | \tag{2.1}$$

   — равную 2:

$$*, /, \%, * =, / =, \% = \tag{2.2}$$

2) Трудоемкость условного оператора:

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай} \end{cases} \tag{2.3}$$

3) Трудоемкость цикла:

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.4}$$

4) Трудоемкость передачи параметра в функции и возврат из функции равны 0.

2.3 Трудоемкость алгоритмов

Была рассчитана трудоемкость алгоритмов сортировки.

2.3.1 Алгоритм итеративной быстрой сортировки

Трудоёмкость алгоритма итеративной быстрой сортировки (2.5)

$$T(N) = T(J) + T(N - J) + M(N) \tag{2.5}$$

где

1) $T(N)$ - трудоемкость быстрой сортировки массива размера N.
2) $T(J)$ - трудоемкость быстрой сортировки массива размера J.
3) $T(N - J)$ - трудоемкость быстрой сортировки массива размера N-J.
4) $M(N)$ - трудоёмкость разделения массива на две части.

Вычислим для лучшего случая:

— $T(N) = 2T(\frac{N}{2}) + C * N$ - трудоемкость быстрой сортировки массива размера N.

  – $2T(\frac{N}{2})$ поскольку мы разделяем массив на 2 равные части
  – $C * N$ поскольку мы будем проходить все элементы массива на каждом уровне "дерева"

— Следующий шаг - разделить дальше на 4 части ((2.6) и (2.7))

$$T(N) = 2(2 * T(\frac{N}{4}) + C * N/2) + C * N \tag{2.6}$$

$$T(N) = 4T(\frac{N}{4}) + 2C * N \tag{2.7}$$

— В общем случае (2.8):

$$T(N) = 2^k * T(N/(2^k)) + k * C * N \tag{2.8}$$

— Для лучшего случая - $N = 2^k$ - идеально распределенное дерево (отсюда следует, что $k = log_2(N)$) (2.9)

$$T(N) = 2^k * T(1) + k * C * (2^k) \tag{2.9}$$

– Вычислим T(1) - трудоемкость при массиве длинной 1 - и C - трудоемкость разделения (2.10)

$$T(1) = 6 + 1 * 9 = 15; C = 12 + N/(2^k) * 8 \tag{2.10}$$

– Полная сложность (при $k = log_2(N)$) (2.11):

$$T(N) = 15N + log_2(N) * (12 + 8) = 15N + 20Nlog_2(N) \tag{2.11}$$

Теперь вычислим для худшего случая:

– $T(N) = T(N-1)+C*N$ - трудоемкость быстрой сортировки массива размера N.

  – $T(N-1)$ поскольку мы разделяем массив на 2 неравные части: пустое множество и полное множество за исключением "середины"

– Следующие шаги очевидны (2.12), (2.13)

$$T(N) = T(N-2)+C(N-1)+C*N = T(N-2)+2C*N-C \tag{2.12}$$

$$T(N) = T(N-3) + 3C * N - 2C * N - C \tag{2.13}$$

– В общем случае (2.14):

$$T(N) = T(N-k) + k * C * N - C(\frac{k(k-1)}{2}) \tag{2.14}$$

– Для худшего случая - $N = k$ - нераспределенное дерево (2.15)

$$T(N) = T(0) + N * C * N - C(\frac{N(N-1)}{2}) \tag{2.15}$$

– Вычислим параметры (2.16)

$$T(0) = 1; C = 12 + (N-k) * 8 \tag{2.16}$$

— Полная сложность (при $N = k$) (2.17)

$$T(N) = 1 + N * N * 12 - 12 * (\frac{N(N-1)}{2}) = 1 + 6N^2 + 6N \quad (2.17)$$

Трудоёмкость в **лучшем** случае (2.18)

$$f_{best} = 15N + 20Nlog_2(N) \approx 20Nlog_2(N) = O(Nlog_2(N)) \quad (2.18)$$

Трудоёмкость в **худшем** случае (2.19)

$$f_{worst} = 1 + 6N^2 + 6N \approx 6N^2 = O(N^2) \quad (2.19)$$

2.3.2 Алгоритм сортировки «расчёской»

Рассчитаем трудоемкость сортировки «расчёской».

Вычислим для лучшего случая:

$$f_{comb} = 3 + 2 + 1 + M \cdot (1 + (3 + N \cdot (3 + 2 + 1)) + 2 + 1) = \\ = 6 + 7N + 6NM \approx 6NM \quad (2.20)$$

В лучшем случае при удачном получении шага обхода M может равняться $log_2(N)$.

Вычислим для худшего случая:

$$f_{comb} = 3 + 2 + 1 + M \cdot (1 + (3 + N \cdot (3 + 7 + 2 + 1)) + 2 + 1) = \\ = 6 + 7N + 13NM \approx 13NM \quad (2.21)$$

В худшем случае M может равняться N.

Тогда трудоёмкость в **лучшем** случае (2.22)

$$f_{best} = 6 + 7N + 6N \cdot log_2(N) \approx 6N \cdot log_2(N) = O(Nlog_2(N)) \quad (2.22)$$

Трудоёмкость в **худшем** случае (2.23)

$$f_{worst} = 6 + 7N + 13N^2 \approx 13N^2 = O(N^2) \quad (2.23)$$

2.3.3 Алгоритм сортировки с помощью бинарного дерева

Трудоёмкость данного алгоритма посчитаем следующим образом: сортировка — преобразование массива в бинарное дерево поиска посредством операции вставки нового элемента в бинарное дерево. Операция вставки в бинарное дерево имеет среднюю алгоритмическую сложность $\log_2(N)$, где N — количество элементов в дереве. Для преобразования массива или списка размером N потребуется использовать операцию вставки в бинарное дерево N раз, таким образом, итоговая трудоёмкость данной сортировки в лучшем случае будет равна:

$$f_{radix} = N \cdot \log_2(N) = O(Nlog_2(N)) \tag{2.24}$$

Однако, сложность добавления объекта в разбалансированное дерево может достигать N. Тогда в худшем случае трудоёмкость сортировки будет равна:

$$f_{radix} = N \cdot N = O(N^2) \tag{2.25}$$

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов и проведена теоретическая оценка трудоемкости алгоритмов. Трудоемкость всех алгоритмов одинакова и в лучшем случае составляет $O(Nlog_2(N))$, в худшем – $O(N^2)$.

3 Технологическая часть

В данном разделе будут описаны требования к программному обеспечению, средства реализации, выбранные типы данных, листинг кода и функциональные тесты.

3.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

— принимает на вход массив;

— выдает массив, являющийся результатом сортировки;

— имеет интерфейс для выбора действий;

— имеет функциональность замера процессорного времени работы реализаций алгоритмов.

3.2 Средства реализации

Для реализации данной лабораторной работы был выбран язык JavaScript [4]. Данный выбор обусловлен наличием у языка встроенной функции $process.cpuUsage()$ для измерения процессорного времени. Это позволяет удовлетворить требованиям для выполнения лабораторной работы.

Время выполнения программы было замерено с использованием функции *process.cpuUsage()* из программной платформы Node.js [5].

3.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— количество элементов — целое число типа $number$;

— массив — массив значений типа $number$;

— бинарное дерево — структура класса $BinaryTree$.

3.4 Сведения о модулях программы

Данная программа разбита на следующие модули.

— `main.js` — файл, содержащий точку входа в программу, из которой происходит запуск работы алгоритмов и вывод графиков.

— `algorithms.js` — файл содержит функции алгоритмов сортировок.

— `times.cjs` — файл содержит функции, замеряющие процессорное время алгоритмов сортировок.

— `index.html` — файл веб-страницы, который используется для отображения результата работы алгоритмов.

— `style.css` — файл, который используется для определения стилей интерфейса веб-страницы.

— `measures.json` — файл, в который `times.cjs` записывает результаты замеров работы алгоритмов, и из которого впоследствии считывает данные файл `main.js` для построения графиков.

3.5 Реализация алгоритмов

В листингах 3.1 – 3.4 приведены реализации алгоритмов сортировок – быстрая сортировка, сортировка «расчёской» и сортировка с помощью бинарного дерева.

Листинг 3.1 – Функция алгоритма быстрой сортировки

```
export function quickSort(array) {
    let n = array.length;
    if (n < 2) {
        return array;
    }
    let pivot = array[0];
    const left = [];
    const right= [];
    for (let i = 1; i < n; i++) {
        if (array[i] < pivot) {
            left.push(array[i]);
        }
        else {
            right.push(array[i]);
        }
    }

    return quickSort(left).concat(pivot, quickSort(right));
}
```

Листинг 3.2 – Функция алгоритма сортировки «расчёской»

```
export function combSort(array) {
    let n = array.length;
    const factor = 1.247;
    let stepFactor = n / factor;
    while (stepFactor > 1) {
        const step = Math.floor(stepFactor);
        for(let i = 0, j = step; j < n; i++, j++) {
            if(array[i] > array[j]) {
                [array[i], array[j]] = [array[j], array[i]];
            }
        }
        stepFactor /= factor;
    }

    return array;
}
```

Листинг 3.3 – Функция алгоритма сортировки бинарным деревом

```
class Node {
    constructor(key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }
}

class BinaryTree {
    constructor() {
        this.root = null;
    }

    insert(key) {
        const newNode = new Node(key);

        if (!this.root) {
            this.root = newNode;
        }
        else {
            this.insertNode(this.root, newNode);
        }
    }

    insertNode(node, newNode) {
        if (newNode.key < node.key) {
            if (node.left === null) {
                node.left = newNode;
            } else {
                this.insertNode(node.left, newNode);
            }
            } else {
            if (node.right === null) {
                node.right = newNode;
            } else {
                this.insertNode(node.right, newNode);
            }
        }
    }
```

Листинг 3.4 – Продолжение листинга 3.3

```
    visit(node, array) {
        if(node === null) return;
        if(node.left !== null) {
            this.visit(node.left, array);
        }
        array.push(node.key);
        if(node.right !== null) {
            this.visit(node.right, array);
        }
    }
}

export function binaryTreeSort(array) {
    let n = array.length;
    let tree = new BinaryTree();
    for (let i = 0; i < n; i++) {
        tree.insert(array[i]);
    }
    let res = [];
    tree.visit(tree.root, res);
    return res;
}
```

3.6 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для вышеизложенных алгоритмов сортировки. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Массив	Быстрая	Расческой	Бинарным деревом
1, 2, 3, 4, 5	1, 2, 3, 4, 5	1, 2, 3, 4, 5	1, 2, 3, 4, 5
3, -9, 42, 1.5	-9, 1.5, 3, 42	-9, 1.5, 3, 42	-9, 1.5, 3, 42
8, 6, 4, 2	2, 4, 6, 8	2, 4, 6, 8	2, 4, 6, 8
1	1	1	1

Вывод

Были реализованы алгоритмы сортировок – быстрая сортировка, сортировка «расчёской» и сортировка с помощью бинарного дерева. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, проведение исследования и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени.

— Процессор: 11th Gen Intel(R) Core(TM) i5-1135G7 2.42 ГГц.

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 11 Домашняя 64-разрядная система версии 22H2 [6].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

Программа получает на вход массив и выдает 3 массива соответствующих алгоритмов сортировок – быстрой сортировки, сортировки «расчёской» и сортировки с помощью бинарного дерева.

На рисунках 4.1 – 4.3 представлена демонстрация работы программы.

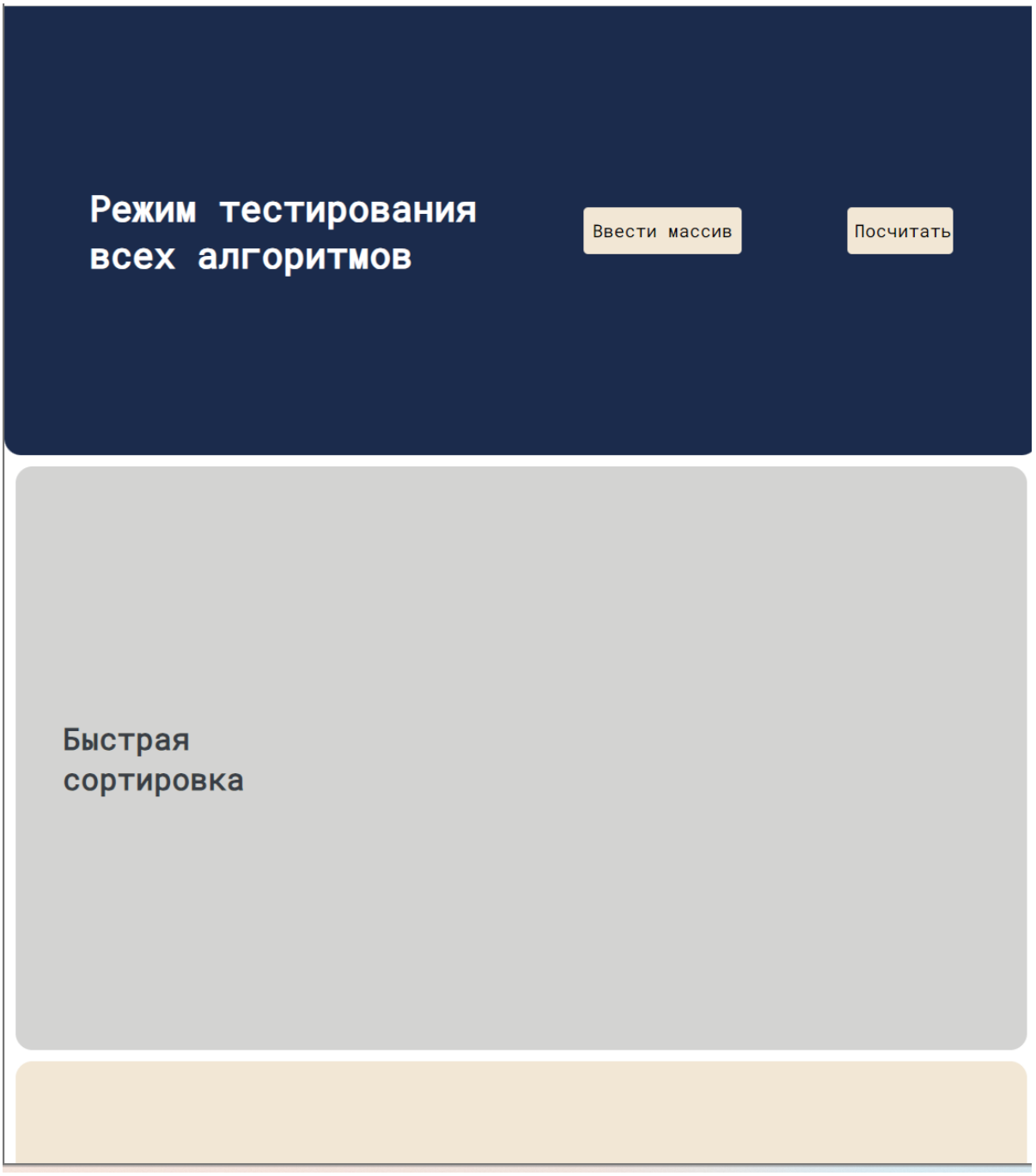

Рисунок 4.1 – Демонстрация интерфейса программы

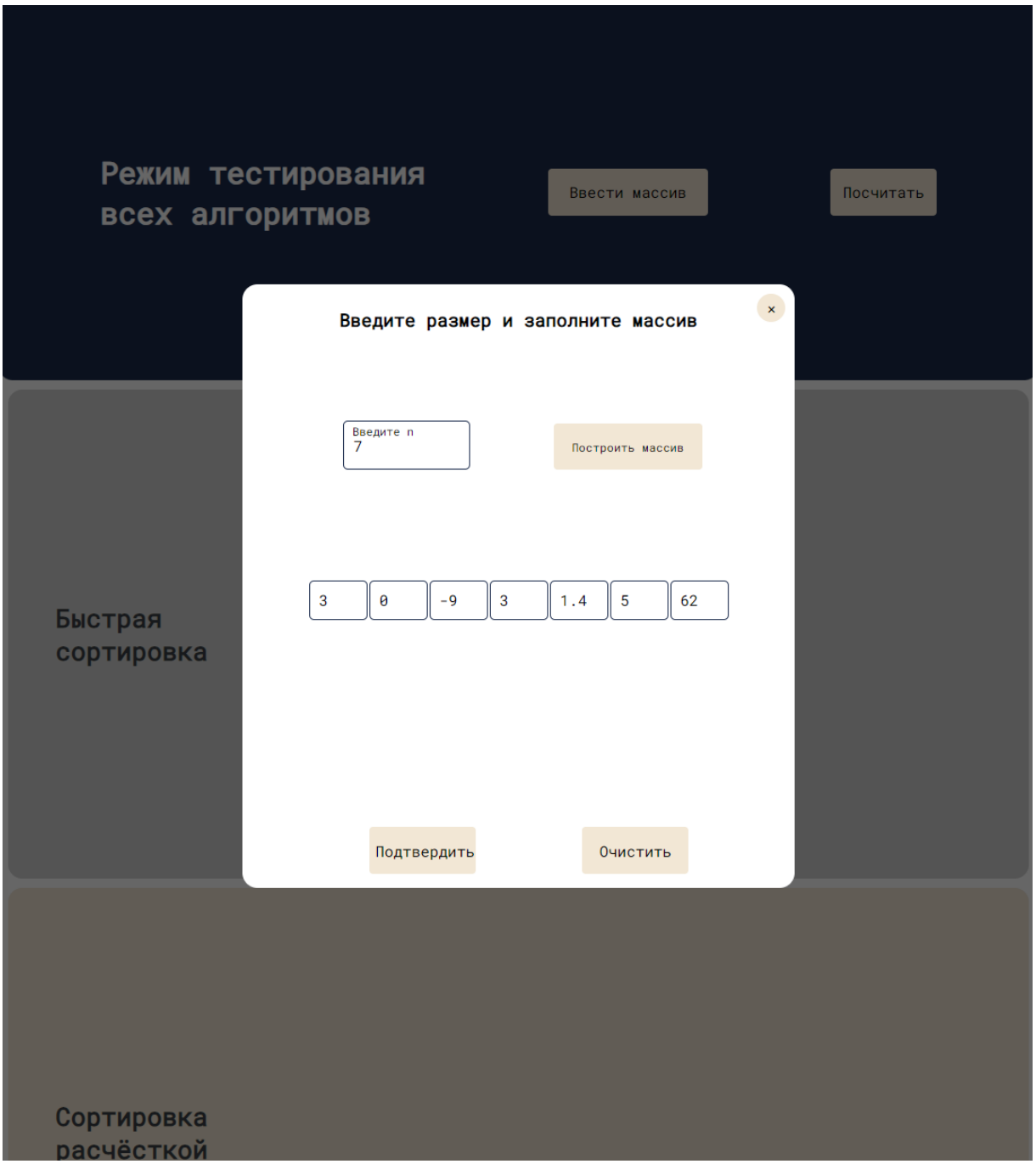

Рисунок 4.2 – Демонстрация ввода данных в программу

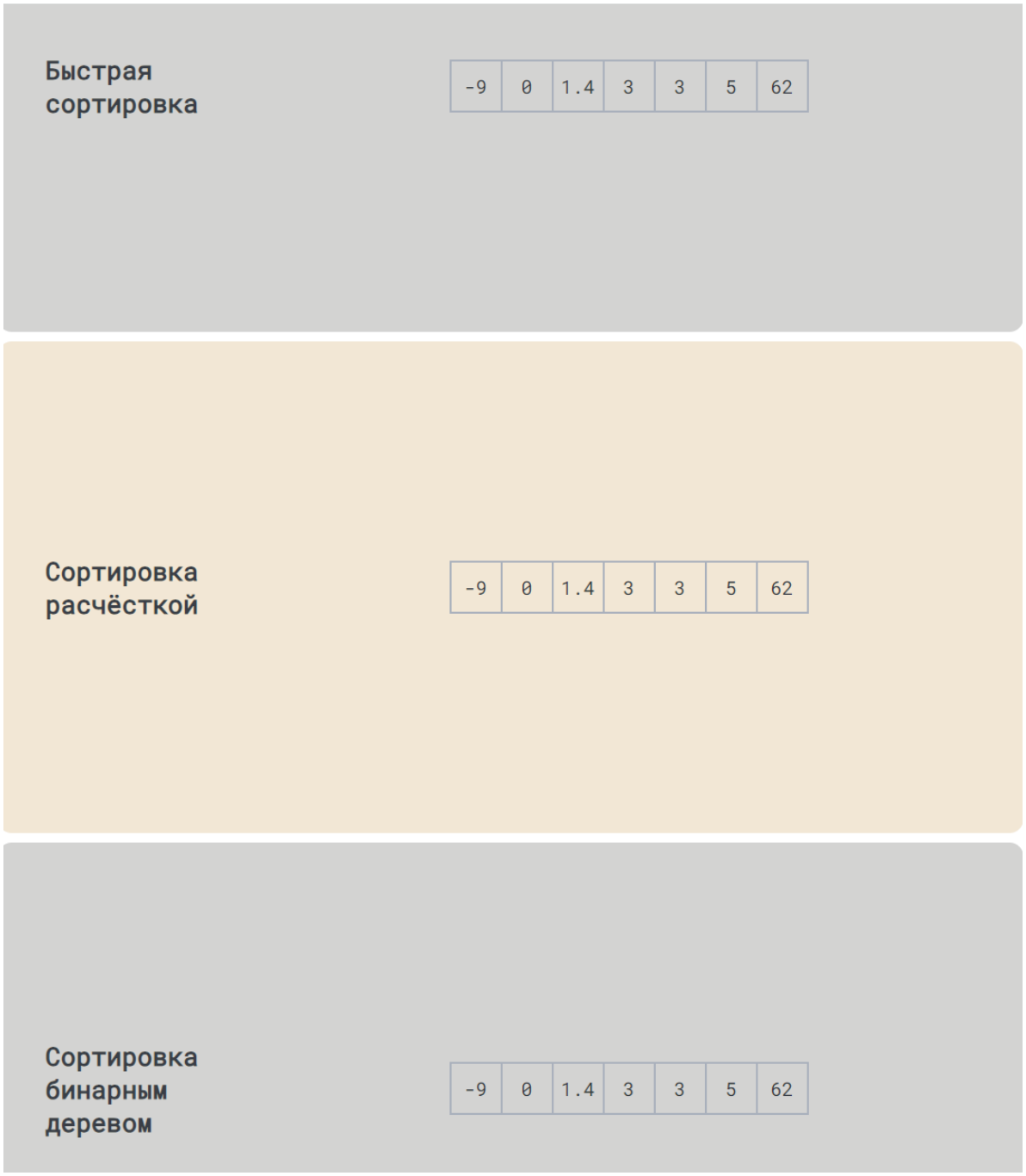

Рисунок 4.3 – Демонстрация результата работы программы

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в таблицах 4.1, 4.3.

В таблице 4.1 приведены результаты замеров по времени алгоритмов сортировок при упорядоченном по возрастанию массиве.

Таблица 4.1 – Замер по времени для упорядоченных по возрастанию массивов, размер которых от 100 до 900

Размер	**Быстрая, нс**	**Расческой, нс**	**Бинарным деревом, нс**
100	0.000275	0.000067	0.000341
200	0.001005	0.000160	0.001549
300	0.002371	0.000263	0.003779
400	0.004120	0.000428	0.007028
500	0.006574	0.000586	0.011227
600	0.009509	0.000730	0.016528
700	0.012926	0.000920	0.022879
800	0.017154	0.001134	0.030228
900	0.021821	0.001290	0.038361

В таблице 4.2 приведены результаты замеров по времени алгоритмов сортировок при случайных элементах массива.

Таблица 4.2 – Замер по времени для массивов со случайными элементами, где размер массива от 100 до 900

Размер	**Быстрая, нс**	**Расческой, нс**	**Бинарным деревом, нс**
100	0.000062	0.000077	0.000074
200	0.000133	0.000200	0.000177
300	0.000238	0.000295	0.000283
400	0.000345	0.000464	0.000410
500	0.000417	0.000636	0.000544
600	0.000534	0.000796	0.000647
700	0.000666	0.000974	0.000767
800	0.000791	0.001182	0.000927
900	0.000866	0.001354	0.001045

В таблице 4.3 приведены результаты замеров по времени алгоритмов сортировок при упорядоченном по убыванию массиве.

По таблице 4.1 был построен график 4.4. Исходя из этих данных можно понять, что лучше всего работает сортировка «расчёской».

Таблица 4.3 – Замер по времени для упорядоченных по убыванию массивов, размер которых от 100 до 900

Размер	**Быстрая, нс**	**Расческой, нс**	**Бинарным деревом, нс**
100	0.000283	0.000072	0.000333
200	0.001063	0.000168	0.001594
300	0.002251	0.000279	0.003761
400	0.004288	0.000446	0.006985
500	0.006696	0.000609	0.011162
600	0.009638	0.000755	0.016448
700	0.013048	0.000955	0.022789
800	0.017463	0.001154	0.030042
900	0.022346	0.001340	0.038335

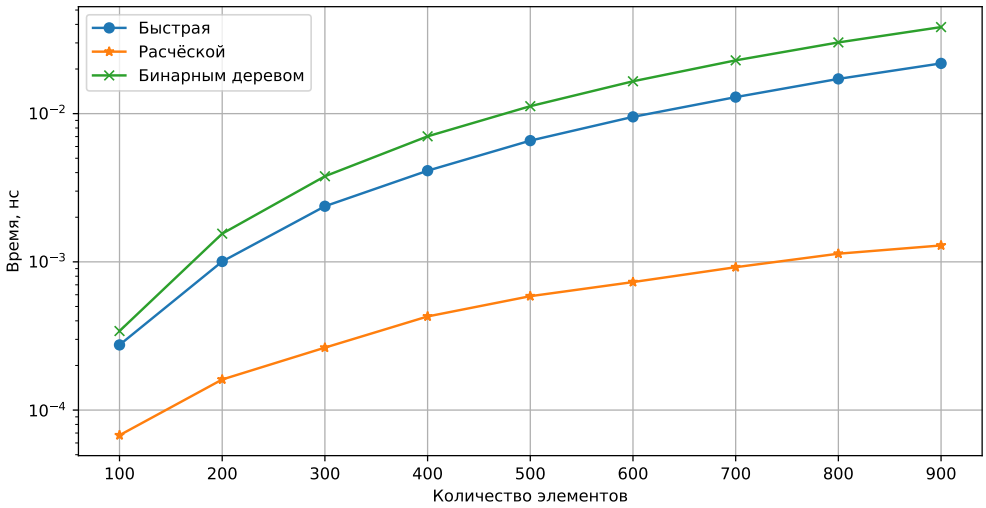

Рисунок 4.4 – Сравнение по времени алгоритмов сортировок при упорядоченном по возрастанию массиве

По таблице 4.2 был построен график 4.5. Исходя из этих данных можно понять, что лучше всего работает быстрая сортировка.

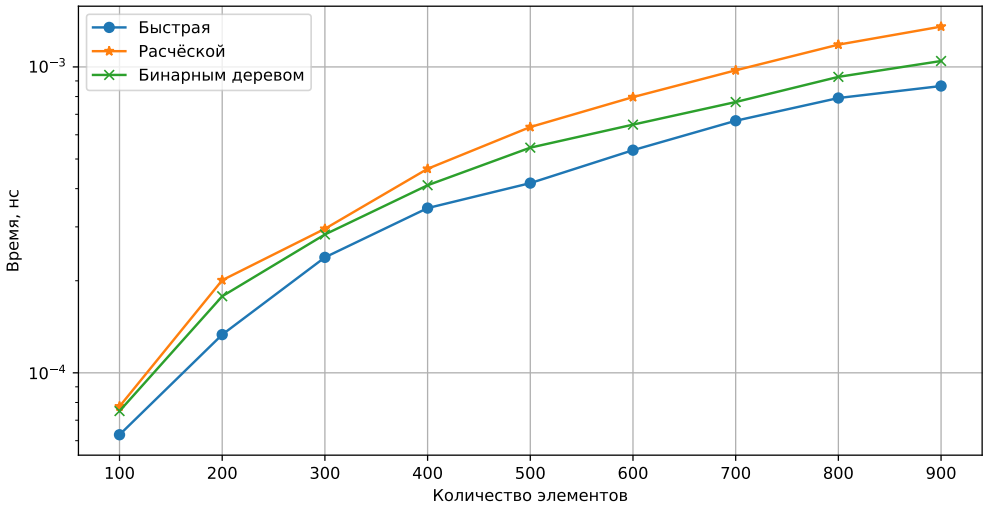

Рисунок 4.5 – Сравнение по времени алгоритмов сортировок при случайном входном массиве

По таблице 4.3 был построен график 4.6. Исходя из этих данных можно понять, что лучше всего работает сортировка «расчёской».

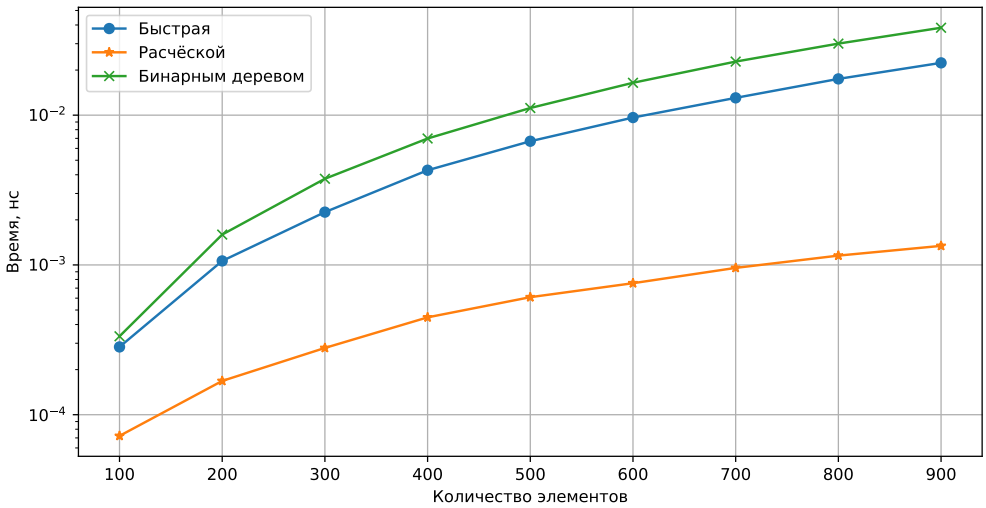

Рисунок 4.6 – Сравнение по времени алгоритмов сортировок при упорядоченном по убыванию массиве

4.4 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов.

Приведенные временные характеристики показывают, что лучшей сортировкой при работе с отсортированным массивом по возрастанию и убыванию является сортировка «расчёской», а при работе со случайным массивом – быстрая сортировка. Сортировка деревом же показала худшие результаты по сравнению с сортировкой «расчёской» и быстрой в отсортированных массивах, но при работе со случайным массивом она обогнала сортировку «расчёской».

Заключение

В результате исследования было определено, что лучшей сортировкой при работе с отсортированным массивом по возрастанию и убыванию является сортировка «расчёской», а при работе со случайным массивом – быстрая сортировка. Сортировка деревом показала худшие результаты по сравнению с сортировкой «расчёской» и быстрой в отсортированных массивах, но при работе со случайным массивом она обогнала сортировку «расчёской».

Трудоемкость всех алгоритмов одинакова и в лучшем случае составляет $O(Nlog_2(N))$, в худшем – $O(N^2)$.

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи. Для достижения поставленной цели необходимо выполнить следующие задачи.

1) Описаны алгоритмы сортировок.
2) Создано программное обеспечение, реализующее следующие алгоритмы сортировок:
   - быстрая;
   - «расчёской»;
   - бинарным деревом.
3) Оценены трудоемкости сортировок.
4) Проведен замер времени реализации.
5) Проведен анализ затрат работы программы по времени, выяснены влияющие на них характеристики.

Список использованных источников

1 Липачѐв. Е.К. Технология программирования. Методы сортировки данных: учебное пособие. — Казань: Казанский университет, 2017. с. 59.

2 Левитин А. В. Алгоритмы. Введение в разработку и анализ. М.: Вильямс, 2006. с. 576.

3 Сортировка бинарным деревом [Электронный ресурс]. Режим доступа: `http://algolist.ru/sort/faq/q7.php` (дата обращения: 13.10.2023).

4 Документация по JavaScript [Электронный ресурс]. — Режим доступа: `https://262.ecma-international.org/13.0/` (дата обращения: 10.10.2023).

5 Node.js function process.cpuUsage([previousValue]) [Электронный ресурс]. — Режим доступа: `https://nodejs.org/api/process.html#processcpuusagepreviousvalue` (дата обращения: 10.10.2023).

6 Windows 11 [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows11` (дата обращения: 10.10.2023).