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»

Отчёт

по лабораторной работе №1

Название: Алгоритмы сортировки

Дисциплина: Анализ алгоритмов

Студент ИУ7-54Б _____ Л.Е.Тартыков
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2021

Оглавление

Введение

При работе с большими объемами данных, расположенных в хаотичном порядке относительно друг друга, часто возникает проблема нахождения нужной записи по некоторому ключу. Например, нахождение в телефонной книге некоторого абонента, если при этом записи не находятся в алфавитном порядке по фамилии.

Для решения подобных задач существует такое решение как сортировка данных. Алгоритм сортировки - алгоритм, необходимый для упорядочивания элементов в списке. Сортировка может проводиться по какому-то критерию (ключу). Было разработано множество алгоритмов, которые различаются по трудоемкости и эффективности в связи с затрачиваемыми ими ресурсами ЭВМ. Целью лабораторной работы является изучение и реализация нерекурсивных алгоритмов сортировок. Для её достижения необходимо выполнить следующие задачи:

- изучение алгоритмов нерекурсивной сортировки;
- рассчитать трудоемкость каждого из выбранных алгоритмов;
- разработать данные алгоритмы;
- выполнить тестирование методом черного ящика;
- провести сравнительный анализ этих алгоритмов по затратам памяти и процессорному выполнению времени на основе экспериментальных данных.

1 Аналитический раздел

1.1 Понятие сортировки

Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

1.2 Критерии выбора алгоритма сортировки

К основным параметрам выбора необходимого алгоритма сортировки относят:

- временная сложность - описывает только то, как производительность алгоритма изменяется в зависимости от размера набора данных.;
- память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы;
- устойчивость - сортировка является устойчивой в том случае, если для любой пары элементов с одинаковым ключами, она не меняет их порядок в отсортированном списке (является важным критерием для баз данных).

1.3 Пузырьковая сортировка

Является одним из самых известных алгоритмов сортировки за счет своей простоты. Идея такой сортировки заключается в том, что все элементы массива сравниваются друг с другом. Они меняются местами в том случае, если предшествующий элемент больше последующего. Этот процесс повторяется до тех пор, пока перестановок в массиве не будет.

Данный алгоритм почти не применяется на практике ввиду своей низкой временной эффективности: он работает медленнее на больших данных. Исключение составляет сортировка малого числа элементов (примерно, до 10 штук), когда такой подход выигрывать по скорости выполнения алгоритмам с меньшей временной сложностью на больших данных.

1.4 Сортировка простыми вставками

Основная идея алгоритма сортировки вставками заключается в том, что исходный массив делится на две части: отсортированную и неотсортированную. В качестве отсортированной части считается первый элемент массива. Затем берется элемент из неотсортированной и добавляется в отсортированную так, чтобы упорядоченность в первой части не нарушилась. Такие дейсвтия продолжаются до тех пор, пока все элементы не окажутся на своих местах в отсортированной части массива.

1.5 Сортировка методом Шелла

Данный алгоритм является модификацией сортировки простыми вставками. Отличие такого метода заключается в том, что сначала выполняется сравнение элементов, находящихся друг от друга на некотором расстоянии. Изначально оно задается как d или N/2, где N - общее число элементов последовательности. На первом шаге каждая группа включает в себя два элемента, расположенных друг от друга на расстоянии N/2. На последующих шагах также происходит проверка и обмен, но

расстояния d при этом сокращается на d/2. Постепенно расстояни между элементами уменьшается, и на d=1 проход по массиву происходи в последний раз.

Вывод

В данном разделе были рассмотрены основные теоретические сведения о трех алгоритмах сортировки.

2 Конструкторский раздел

2.1 Схемы алгоритмов Левенштейна

Ниже представлены следующие схемы алгоритмов:

- рис. 2.1 - схема алгоритма сортировки пузырьком;
- рис. 2.2 - схема алгоритма сортировки простыми вставками;
- рис. 2.3 - схема алгоритма сортировки методом Шелла.

2.2 Модель оценки трудоемкости алгоритмов

Введем модель оценки трудоемкости.

1. Трудоемкость базовых операций. Пусть трудоемкость операций *, /, //, %, *=, /= равна 2.

   Примем трудоемкость следующих операций равной 1:

   =, +, -, +=, -=, ==, !=, <, >, <=, >=, |, &&„ ||, []

2. Трудоемкость цикла. Пусть трудоемкость цикла определяется по формуле 2.1:

$$f = f_{init} + f_{comp} + N_{iter} * (f_{in} + f_{inc} + f_{comp}) \tag{2.1}$$

   где:

   - f_{init} - трудоемкость инициализации переменной-счетчика;
   - f_{comp} - трудоемкость сравнения;
   - N_{iter} - номер выполняемой итерации;
   - f_{in} - трудоемкость команд из тела цикла;
   - f_{inc} -трудоемкость инкремента;

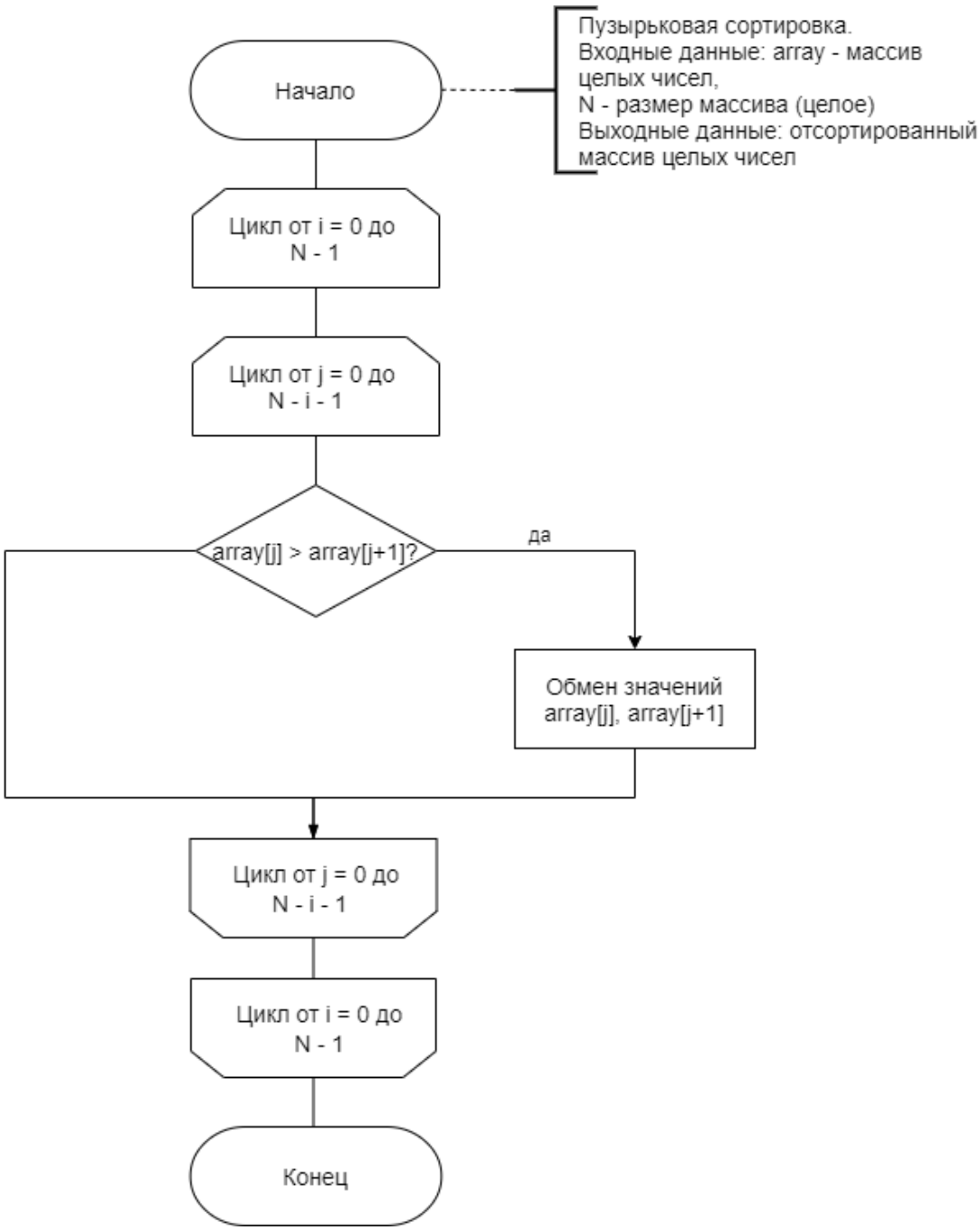

Рисунок 2.1 – Ссхема алгоритма итеративного Левенштейна с использованием двух строк.

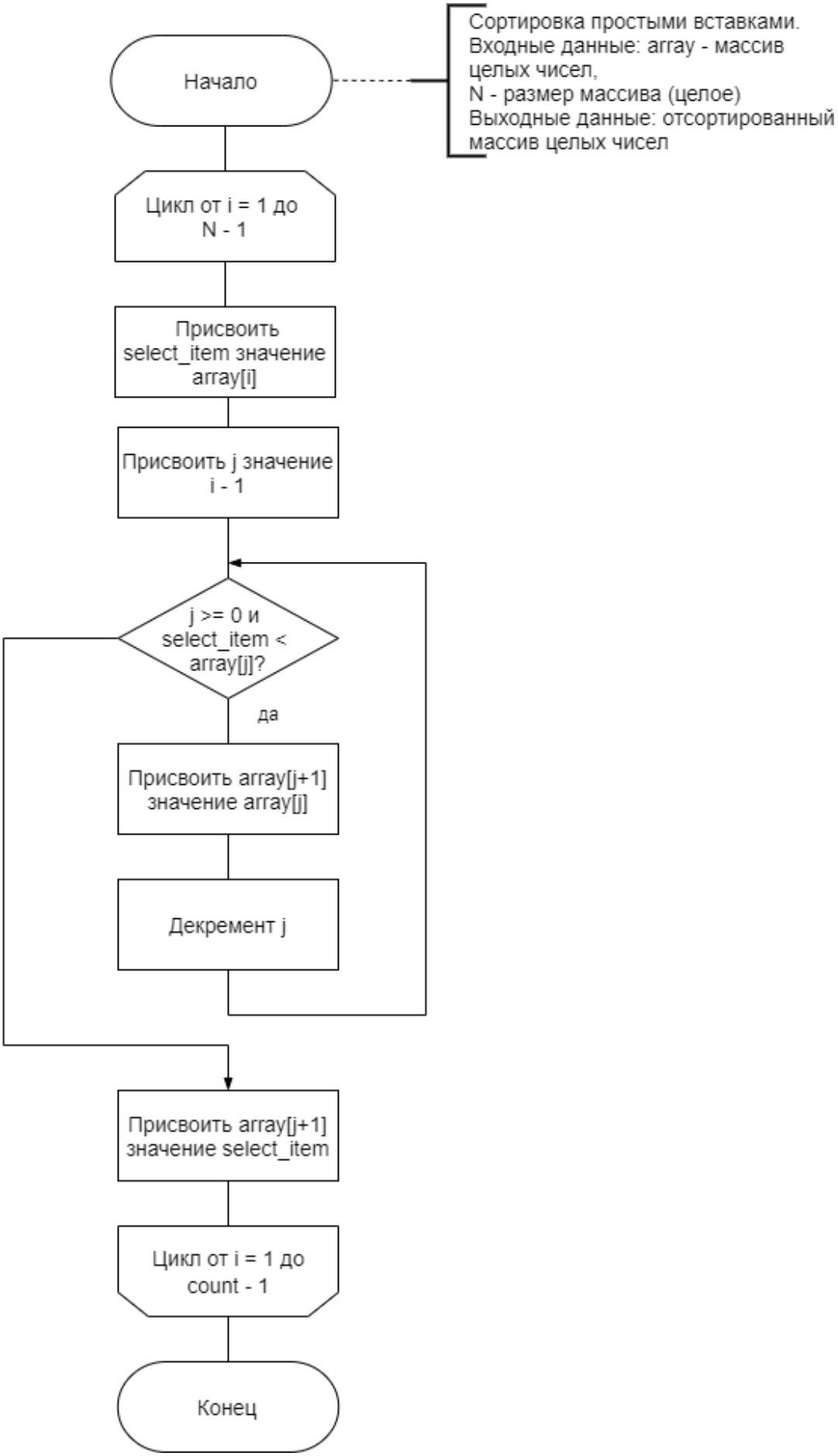

Рисунок 2.2 – Схема алгоритма рекурсивного Левенштейна без кэша.

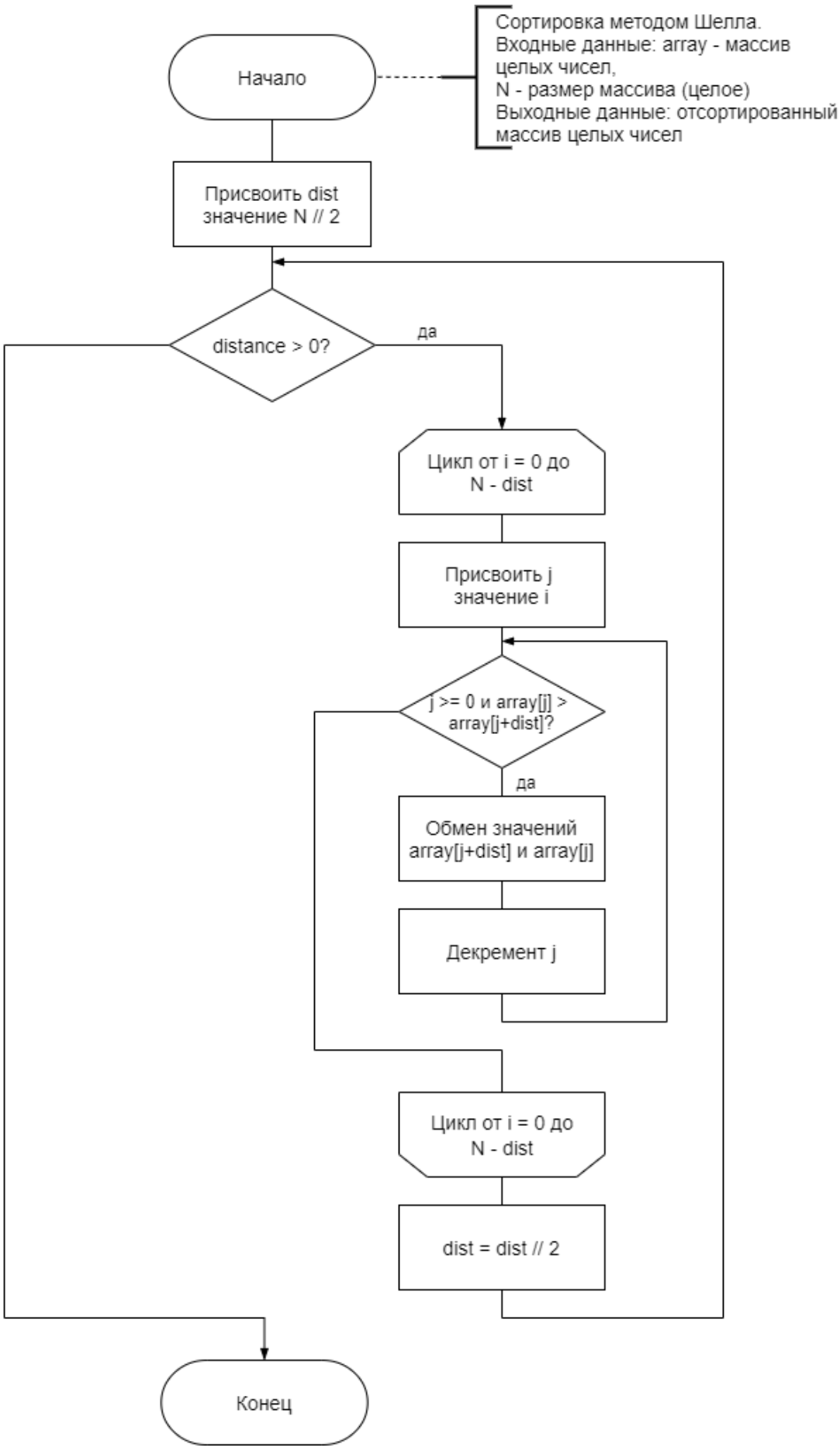

Рисунок 2.3 – Схема алгоритма рекурсивного Левенштейна с использованием матрицы.

- f_{comp} - трудоемкость сравнения.

3. Трудоемкость условного оператора.
   Пусть трудоемкость самого условного перехода равна 0, но она определяется по формуле 2.2.

$$f_{if} = f_{comp\_if} + \begin{cases} f_a \\ f_b \end{cases} \tag{2.2}$$

2.3 Вычисление трудоемкости алгоритмов

Пусть размер массивов во всех дальнейших вычислениях обозначается как N.

2.3.1 Трудоемкость пузырька

Трудоемкость этого алгоритма сортировки состоит из:

- трудоемкость внешнего цикла вычисляется по формуле 2.3;

$$f_i = \underset{=}{1} + \underset{<}{1} + N = 2 + N \tag{2.3}$$

- трудоемкость внутреннего цикла вычисляется по формуле 2.4;

$$f_j = \underset{i++}{2} + \underset{=}{1} + \underset{<}{1} + \underset{-}{1} + \frac{N-1}{2} * (\underset{j++}{2} + f_{if}) = 6 + \frac{N-1}{2} * (2 + f_{if}) \tag{2.4}$$

  где вычисление худшего/лучшего случая определяется по формуле 2.5:

$$f_{if} = \underset{>}{1} + \underset{[\,]}{2} + \underset{+}{1} + \begin{cases} 0 \\ \underset{=}{3} + \underset{[\,]}{4} + \underset{+}{2} \end{cases} = \begin{cases} 0, \text{ лучший случай} \\ 9, \text{ худший случай} \end{cases} \tag{2.5}$$

Итоговая трудоемкость рассчитывается по формуле 2.6:

$$f_{sum} = f_i * f_j \approx O(N^2) \tag{2.6}$$

Трудоемкость пузырька для лучшего случая 2.7:

$$f_{sum} = 3N^2 + 3N + 2 \approx O(N^2) \tag{2.7}$$

Трудоемкость пузырька для худшего случая 2.8:

$$f_{sum} = 7.5N^2 - 1.5N + 2 \approx O(N^2) \tag{2.8}$$

2.3.2 Трудоемкость вставок

2.3.3 Трудоемкость Шелла

Вывод

На основе теоретических данных, полученных в аналатическом разделе, были построены схемы нужных алгоритмов.

3 Технологический раздел

3.1 Требования к программному обеспечению

Программа должна отвечать следующим требованиям:

- на вход программе подается массив целых чисел и размерность этого массива;
- на выход программа выдает отсортированный массив целых чисел.

3.2 Выбор средств реализации

Для реализации алгоритмов в данной лабораторной работе был выбран язык программирования Python 3.9.7[1]. Он является кроссплатформенным. Имеется опыт разработки на этом языке. В качестве среды разработки был использован Visual Studio Code[2], так как в нем можно работать как на операционной системе Windows, так и на дистрибутивах Linux. При замере процессорного времени был использован модуль time[3]. Замеры используемой памяти и число вызовов рекурсии проводились при помощи модуля cProfiler[4].

3.3 Листинги программ

Ниже представлены листинги разработанных алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Программный код сортировки пузырьком

```
1 def bubble_sort(array: list[int], count: int) -> list[int]:
2     for i in range(count):
3         for j in range(count - i - 1):
4             if array[j] > array[j + 1]:
5                 array[j], array[j + 1] = array[j + 1], array[j]
```

Листинг 3.2 – Программный код сортировки вставками

```
def insert_sort(array: list[int], count: int) -> list[int]:
    for i in range(1, count):
        select_item = array[i]
        j = i - 1
        while j >= 0 and select_item < array[j]:
            array[j+1] = array[j]
            j -= 1
        array[j + 1] = select_item
```

Листинг 3.3 – Программный код сортировки методом Шелла

```
def shell_sort(array: list[int], count:int) -> list[int]:
    distance = count // 2
    while distance > 0:
        for i in range(count - distance):
        j = i
        while j >= 0 and array[j] > array[j + distance]:
            array[j + distance], array[j] = array[j], array[j + distance]
            j -= 1
        distance //= 2
```

3.4 Тестирование

Для тестирования используется метод черного ящика. В данном разделе приведена таблица 3.1, в которой указаны классы эквивалентностей тестов:

Таблица 3.1 – Таблица тестов

№	Описание теста	Слово 1	Слово 2	Алгоритм	
				Левенштейн	Дамерау-Левенштейн
1	Пустые строки	”	”	0	0
2	Нет повторяющихся символов	deepcopy	раздел	8	8
3	Инверсия строк	insert	tresni	6	6
4	Два соседних символа	heart	heatr	2	1
5	Одинаковые строки	таблица	таблица	0	0
6	Одна строка меньше другой	город	горо	1	1

Вывод

В данном разделе был выбран язык программирования, среда разработки. Реализованы функции, описанные в аналитическом разделе, и проведено их тестирование методом черного ящика по таблице 3.1.

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристии устройства, на котором выполнялось тестирование:

- операционная система: Windows 10 Pro;
- память: 8 GiB;
- процессор: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz.

Тестирование проводилось на ноутбуке, который был подключен к сети питания. Во время проведения тестирования ноутбук был нагружен только встроенными приложениями окружения, самим окружением и системой тестирования.

4.2 Временные харастеристики выполнения

Ниже был проведен анализ времени работы алгоритмов. Исходными данными будут случайно сгенерированные строки длиной {3, 4, 5, 6, 7, 8}. Единичные замеры выдадут крайне маленький результат, поэтому проведем работу каждого алгоритма n = 1000 раз и поделим на число n. Получим среднее значение работы каждого из алгоритмов. Результат приведен на рис 4.1:

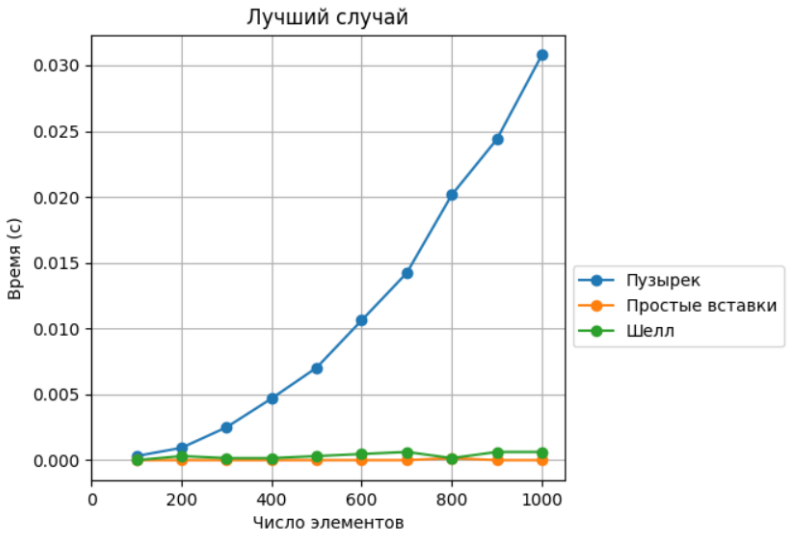

Рисунок 4.1 – Сравнение времени работы алгоритмов Левенштейна.

Как видно из результатов, рекурсивный алгоритм Левенштейна без кэша и алгоритм Дамерау-Левенштейна имеют большой рост, начиная уже со строки длиной 7. Последний имеет наибольший рост. Это объясняется тем, что этот алгоритм задействует дополнительную операцию - транспозицию, которая тоже приводит к вызову рекурсии.

Выполнив анализ двух остальных алгоритмов на значения входных строк длиной {25, 50, 75, 100, 125, 150}, получим следующий результат, представленный на рис 4.2:

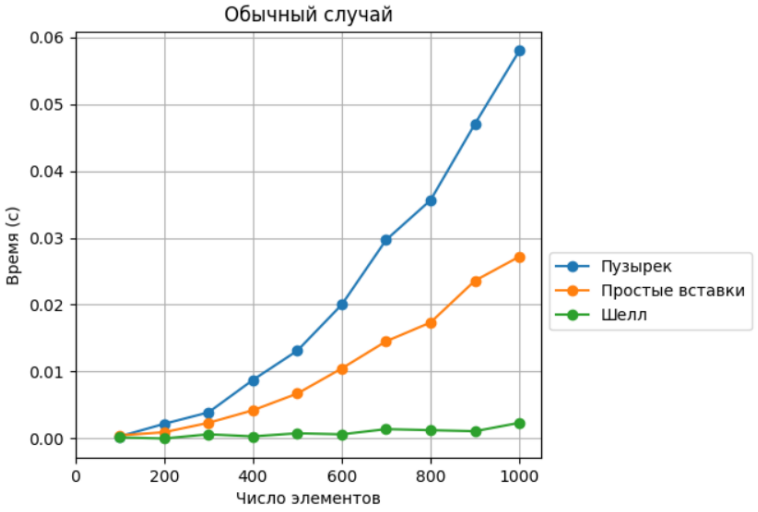

Рисунок 4.2 – Сравнение времени работы рекурсивного и некурсивного алгоритмов Левенштейна.

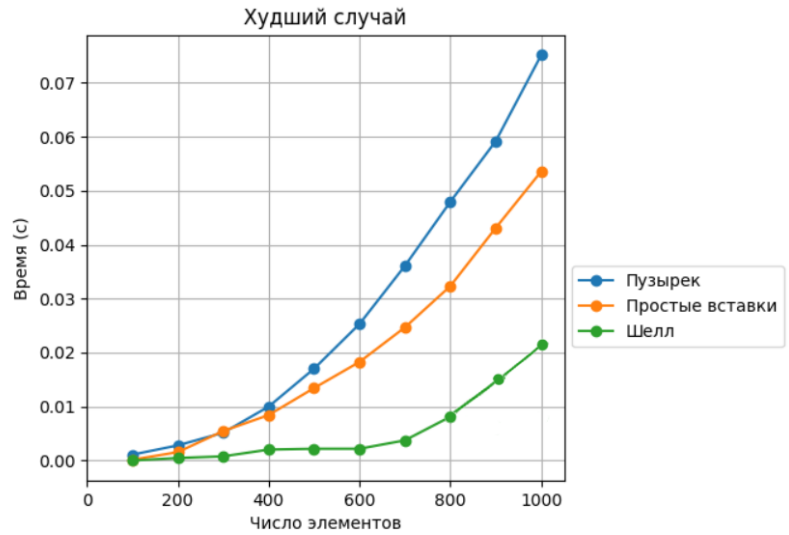

Рисунок 4.3 – Сравнение времени работы рекурсивного и некурсивного алгоритмов Левенштейна.

Рекурсивный алгоритм Левенштейна с использованием матрицы выполняется быстрее, чем итеративный с использованием двух строк. Это объясняется тем, что в итеративном случае выполняется дополнительная операция по обмену значений двух строк. На это требуется дополнительное время.

4.3 Объем потребляемой памяти

При исходных строках, длиною 3, требуется 52,8 Mb памяти. Результаты вызовов и объем потребляемой памяти приведены в таблице 4.1:

Таблица 4.1 – Число вызовов каждого алгоритма

Левенштейн			Дамерау-Левенштейн
Итеративный с двумя строками	Рекурсивный без кэша	Рекурсивный с матрицей	Рекурсивный
1	94	28	94

Общее значение потребляемой памяти складывается по формуле 4.3:

$$S = n\_calls * V \tag{4.3.1}$$

где:

- n_calls - число вызовов функций;
- V - объем памяти, занимаемый одним вызовом функции.

По результатам исследования памяти алгоритмы Левенштейна и Дамерау-Левенштейна потребляют наибольшую память при выполнении по сравнению с другими. Итеративный алгоритм Левенштейна с двумя строками занимает меньше памятив сравнении с остальными.

Вывод

Рекурсивный вызов Левенштейна без кэша и Дамерау-Лвенштейна проигрывают как по скорости, так и по памяти итеративному. Причем рекурсивный алгоритм Левенштейна с матрицей работает быстрее, чем итеративный с двумя строками, но также проигрывает ему по памяти.

Сравнивая между собой рекурсивные вызовы алгоритмов Левенштейна и Дамерау-Левенштейна, можно сделать вывод о том, что рекурретный алгоритм поиска расстояния Левенштейна с матрицей выигрывает как по времени, так и по памяти у других реализаций этих алгоритмов, а рекуррентный Дамерау-Левенштейн проигрывает им по обоим параметрам. Однако, стоит отметить, что в системах автоматического исправления текста, где чаще всего встречаются ошибки, связанные с транспозицией двух символов, выполнение исправления ошибок выбор алгоритма Дамерау-Левенштейна будет наиболее оптимальным решением.

Заключение

В ходе выполнения лабораторной работы были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Были выполнено описание каждого из этих алгоритмов, приведены соотвествующие математические расчёты. Были получены навыки динамического программирования, а также реализованны данные алгоритмы. При тестировании каждого их них и анализе временных характеристик и объема потребляемой памяти можно сделать следующие выводы: выбор алгоритма Дамерау-Левенштейна является наиболее оптимальным решением ввиду того, что чаще всего необходимо исправлять ошибки, связанные с обменом двух соседних символов. В ином случае этот алгоритм является проигрышным как по времени, так и по памяти в сравнении с различными реализациями алгоритма Левенштейна. Самым быстрым среди данной группы алгоритмов по времени является рекурсивный алгоритм Левенштейн с кэшем в виде матрицы; но он достаточно много использует памяти за счет большого числа вызовов. Поэтому в иных ситуациях, не связанных с транспозицей, следует использовать итеративный алгоритм.

Литература

[1] Python. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/ (дата обращения: 21.09.2021).

[2] Documentation for Visual Studio Code. [Электронный ресурс]. Режим доступа: https://code.visualstudio.com/docs (дата обращения: 21.09.2021).

[3] time — Time access and conversions — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 22.09.2021).

[4] The Python Profilers — Python 3.9.7 documentation. [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/profile.html (дата обращения: 22.09.2021).