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 № 1 по курсу "Анализ алгоритмов"

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Хамзина Р. Р.

Группа ИУ7-53Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2021 г.

Содержание

Введение

Задача поиска и сравнения последовательностей возникает при обработке больших объемов информации, при работе с неструктурированными данными или при поисковых запросах. Алгоритмы для решения этой задачи разделяются на три группы: точный поиск подстрок, неточный поиск и поиск наибольшей общей подпоследовательности. Алгоритмы нечеткого поиска применяются в компьютерной лингвистике (коррекция ошибок, поиск с учетом формы одного и того же слова) и в биоинформатике (сравнение ДНК). В основе методов лежит определение расстояния между строками, которое задается в конкретном алгоритме.

Данная задача впервые была поставлена советским математиком Владимиром Левенштейном для двоичных кодов. Расстояние между строками для прозвольного алфавита назвали расстоянием Левенштейна и определили как минимальное количество операций вставки, удаления и замены одного символа, необходимых для превращения одной строки в другую [1].

Модификацией алгоритма Левенштейна является алгоритм Дамерау-Левенштейна. В данном алгоритме к операциям, необходимым для перевода одной строки в другую, добавляется транспозиция символов (перестановка двух соседних символов).

Данные алгоритмы реализуются методом динамического программирования. Существуют их рекурсивная и нерекурсивная реализации, которые различаются временной эффективностью.

Цель работы - изучить метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Для решения поставленной цели требуется решить следующие задачи:

- изучить алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- разработать алгоритм Левенштейна в рекурсивной, матричной и в рекурсивной с использованием матрицы версиях;
- разработать алгоритм Дамерау-Левенштейна в рекурсивной версии;
- реализовать рассматриваемые алгоритмы;

- провести сравнительный анализ реализованных алгоритмов по времени и памяти;
- подготовить отчет о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут описаны алгоритмы нахождения расстояния Левенштейна в рекурсивной, матричной и в рекурсивной с использованием матрицы версиях и расстояния Дамерау-Левенштейна в рекурсивной версии.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рассмотрим перевод строки $A' = A[:i]$ в строку $B' = B[:j]$, последние символы которых соответственно a_{i-1} и b_{j-1}, при помощи следующих операций, каждая из которых имеет свою стоимость (x, y - символы, λ - строка):

- вставка символа в произвольное место - I, стоимостью $w(x, \lambda) = 1$;
- удаление символа с произвольной позиции - D, $w(\lambda, y) = 1$;
- замена символа на другой - R, $w(x, y) = 1$, $x \neq y$.

Расстояние Левенштейна - минимальное количество операций I, D, R для перевода $A' = A[:i]$ в $B' = B[:j]$ [1]. Его можно подсчитать по следующей реккурентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1, j-1) + m(a_{i-1}, b_{j-1}) & (1.2) \\ \} & \end{cases} \tag{1.1}$$

Функция 1.2 определена как:

$$m(x,y) = \begin{cases} 0 & \text{если x = y,} \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Рекурсия используется для реализации формулы 1.1. Пусть $D(i,j)$ - есть расстояние Левенштейна между строками $A' = A[:i]$ и $B' = B[:j]$. $B' = B[:j]$ может быть получен из $A' = A[:i]$ следующими способами:

- если символ a_{j-1} был добавлен при редактировании, то необходимо $A[:i]$ превратить в $B[:j-1]$, на что необходимо $D(i,j-1)$ операций редактирования. В этом случае $D(i,j) = D(i,j-1) + 1$;
- если символ a_{i-1} был удален при редактировании, тогда необходимо $A[:i-1]$ превратить в $B[:j]$, на что необходимо $D(i-1,j)$ операций редактирования. В этом случае $D(i,j) = D(i-1,j) + 1$;
- если последние символы префиксов совпадают, т. е. $a_{i-1} = b_{j-1}$, то в этом случае можно не менять эти последние символы. Тогда $D(i,j) = D(i-1,j-1)$;
- если $a_{i-1} \neq b_{j-1}$, то тогда можно потратить 1 операцию на замену символа на и также потратить операцию $D(i-1,j-1)$ на превращение $A[:i-1]$ в $B[:j-1]$. Тогда $D(i,j) = D(i-1,j-1) + 1$.

Далее при вычислении необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 3 или 4 рассматривается только один, в зависимости от условия $a_{i-1} = b_{j-1}$).

Начальные значения: $D(i,0) = i$, $D(0,j) = j$.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Промежуточные значения $D(i,j)$ вычисляются несколько раз, поэтому при больших значениях i, j временная эффективность алгоритма сни-

жается. Для решения этой проблемы используют матрицу для хранения значений расстояний.

Матрицу M размером $N+1$ и $L+1$, где N и L - длины строк $A' = A[:i]$ и $B' = B[:j]$, заполняют по следующим правилам:

- $M[0][0] = 0$;
- $M[i][j]$ вычисляется по формуле 1.3.

$$M[i][j] = \min \begin{cases} M[i][j-1] + 1 \\ M[i-1][j] + 1 \\ M[i-1][j-1] + m(A[:i-1], B[:j-1]) \quad (1.4) \end{cases} \quad (1.3)$$

Функция 1.4 определена как:

$$m(x,y) = \begin{cases} 0 & \text{если x = y,} \\ 1 & \text{иначе} \end{cases}. \quad (1.4)$$

$M[N][L]$ содержит значение расстояния Левенштейна.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы

Для оптимизации рекурсивного алгоритма можно использовать матрицу как промежуточный буфер, в который при выполнении рекурсии заносятся значения расстояния Левенштейна для еще не рассмотренных символов. Рассмотренные раннее символы в матрицу не добавляются.

1.4 Алгоритм нахождения расстояния Дамерау-Левенштейна

Как было сказано раннее, алгоритм Дамерау-Левенштейна отличается от алгоритма Левенштейна добавлением операции транспозиции символов при переводе одной строки в другую. Расстояние Дамерау-Левенштейна может быть найдено по формуле:

$$d(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d(i,j-1)+1, & \\ \quad d(i-1,j)+1, & \\ \quad d(i-1,j-1)+m(a_{i-1},b_{j-1}), & \text{иначе} \\ \quad \begin{cases} d(i-2,j-2)+1, & \text{если } i,j>1; \\ & a_i = b_{j-1}; \\ & b_j = a_{i-1} \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.5}$$

Пусть $d(i,j)$ - есть расстояние Дамерау-Левенштейна между строками $A' = A[:i]$ и $B' = B[:j]$. $B' = B[:j]$ может быть получен из $A' = A[:i]$ следующими способами:

- если символ a_{i-1} был удален при редактировании, тогда необходимо $A[:i-1]$ превратить в $B[:j]$, на что необходимо $d(i-1,j)$ операций редактирования. В этом случае $d(i,j) = d(i-1,j)+1$;
- если символ a_{j-1} был добавлен при редактировании, то необходимо $A[:i]$ превратить в $B[:j-1]$, на что необходимо $d(i,j-1)$ операций редактирования. В этом случае $d(i,j) = d(i,j-1)+1$;
- если последние символы префиксов совпадают, т. е. $a_{i-1} = b_{j-1}$, то в

этом случае можно не менять эти последние символы. Тогда $d(i, j) = d(i-1, j-1)$;

- если $a_{i-1} \neq b_{j-1}$, то тогда можно потратить 1 операцию на замену символа на и также потратить операцию $D(i-1, j-1)$ на превращение $A[:i-1]$ в $B[:j-1]$. Тогда $d(i, j) = d(i-1, j-1) + 1$;
- если последние два символа a_i и b_j были переставлены, то $d(i, j)$ может быть равно $d(i-2, j-2) + 1$.

Далее при вычислении необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 3 или 4 рассматривается только один, в зависимости от условия $a_{i-1} = b_{j-1}$).

Начальные значения: $D(i, 0) = i$, $D(0, j) = j$.

Вывод

Были рассмотрены алгоритмы нахождения расстояния перевода одной строки в другую при помощи алгоритма Левенштейна и алгоритма Дамерау-Левенштейна.

2 Конструкторская часть

В данном разделе будут указаны требования к программному обеспечению и представлены схемы алгоритмов сортировки выбором, Шеллом и гномьей сортировки.

2.1 Требования к ПО

К программе представлен ряд требований:

- на вход подается массив целых чисел в диапазоне [-1000;1000];
- на выходе - отсортированный массив, поданный на вход. При сортировке изменяется изначальный массив, а не его копия.

2.2 Разработка алгоритмов

На рисунках 2.1, 2.2, 2.3 представлены схемы алгоритмов сортировки выбором, Шелла и гномьей сортировки.

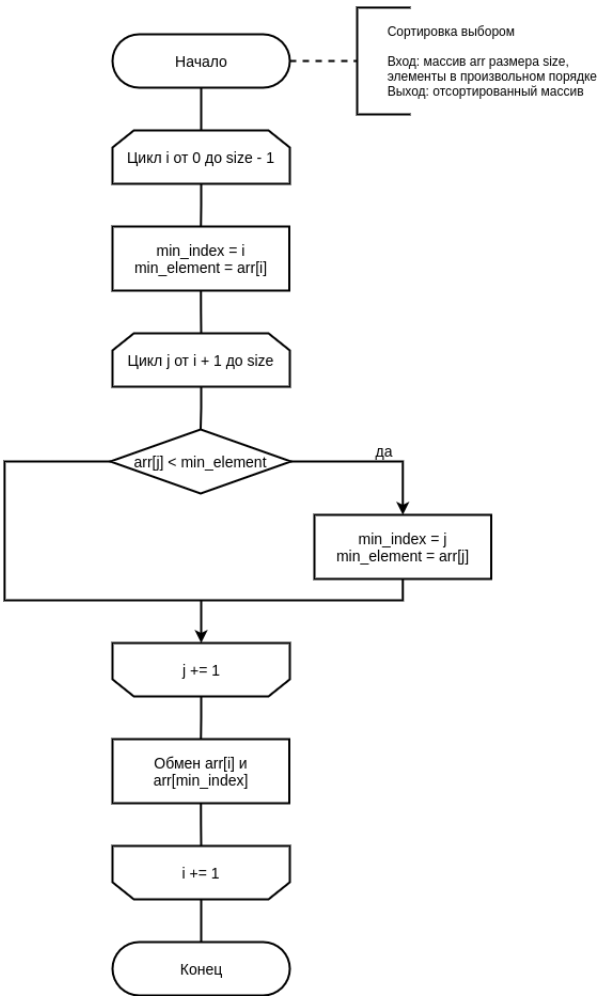

Рисунок 2.1 – Схема алгоритма сортировки выбором

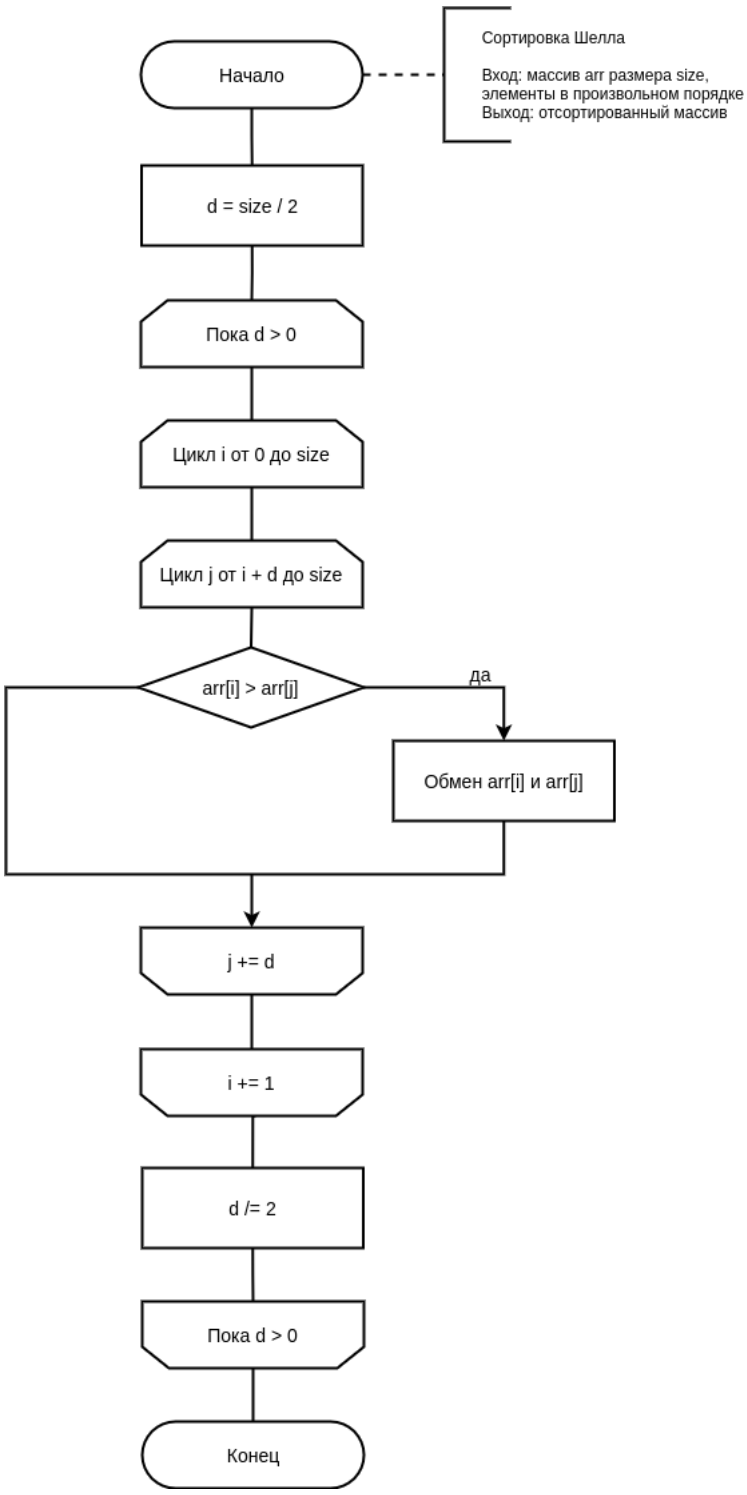

Рисунок 2.2 – Схема алгоритма сортировки Шелла

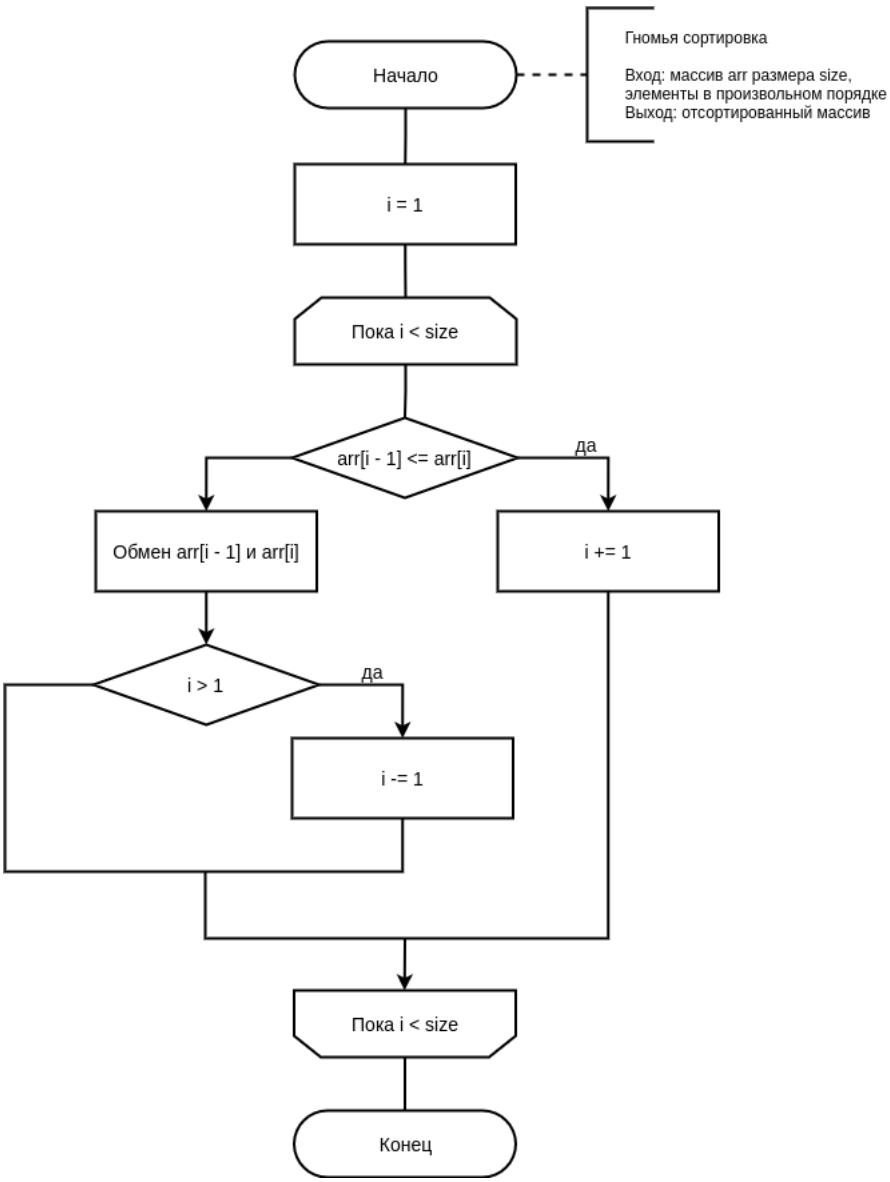

Рисунок 2.3 – Схема алгоритма гномьей сортировки

Вывод

Были представлены схемы алгоритмов сортировки выбором, Шелла и гномьей сортировки и указаны требования к ПО.

3 Технологическая часть

В данном разделе будут указаны средства реализации, будут представлены листинги кода, а также функциональные тесты.

3.1 Средства реализации

Реализация данной лабораторной работы выполнялась при помощи языка программирования Python [2]. Выбор ЯП обусловлен простотой синтаксиса, большим числом библиотек и эффективностью визуализации данных.

Замеры времени проводились при помощи функции process_time из библиотеки time [3].

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- main.py - главный файл программы, предоставляющий пользователю меню для выполнения основных функций;
- sort.py - файл, содержащий функции сортировок массива;
- arr.py - файл, содержащий функции создания массива различного типа и работы с массивом;
- time_test.py - файл, содержащий функции замеров времени работы сортировок;
- graph_result.py - файл, содержащий функции визуализации временных характеристик алгоритмов сортировок.

3.3 Листинги кода

Реализации алгоритмов сортировок выбором, Шелла и гномьей представлены на листингах 3.1, 3.2, 3.3.

Листинг 3.1 – Алгоритм сортировки выбором

```
def selection_sort(arr, size):
    for i in range(size - 1):
        min_element = arr[i]
        min_index = i

        for j in range(i + 1, size):
            if arr[j] < min_element:
                min_element = arr[j]
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr
```

Листинг 3.2 – Алгоритм сортировки Шелла

```
def shell_sort(arr, size):
    d = size // 2

    while d > 0:
        for i in range(0, size):
            for j in range(i + d, size, d):
                if arr[i] > arr[j]:
                    arr[i], arr[j] = arr[j], arr[i]
        d //= 2

    return arr
```

Листинг 3.3 – Алгоритм гномьей сортировки

```
def gnome_sort(arr, size):
    i = 1

    while i < size:
        if arr[i - 1] <= arr[i]:
            i += 1
        else:
            arr[i - 1], arr[i] = arr[i], arr[i - 1]

            if i > 1:
                i -= 1

    return arr
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для функций, реализующих алгоритмы сортировок. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4, 5, 6]$	$[1, 2, 3, 4, 5, 6]$	$[1, 2, 3, 4, 5, 6]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[0, -5, 3, 7, -8]$	$[-8, -5, 0, 3, 7]$	$[-8, -5, 0, 3, 7]$
$[17]$	$[17]$	$[17]$
$[]$	$[]$	$[]$

Вывод

Были реализованы функции алгоритмов сортировки выбором, Шелла и гномьей. Было проведено функциональное тестирование указанных функций.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, и будут проведены сравнительный анализ реализованных алгоритмов сортировки по затраченному процессорному времени и сравнительный анализ трудоемкости алгоритмов.

4.1 Модель вычислений для оценки трудоёмкости алгоритмов

Для определения трудоемкости алгоритмов необходимо ввести модель вычислений [?]:

1. операции из списка (4.1) имеют трудоемкость равную 1;

$$+, -, /, *, \%, =, +=, -=, *=, /=, \%=, ==, !=, <, >, <=, >=, [], ++, -- \tag{4.1}$$

2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (4.2);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{4.2}$$

3. трудоемкость цикла рассчитывается, как (4.3);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{4.3}$$

4. трудоемкость вызова функции равна 0.

4.2 Трудоёмкость алгоритмов

Проведем сравнительный анализ реализованных алгоритмов по трудоемкости.

4.2.1 Алгоритм сортировки выбором

Трудоемкость в лучшем случае (4.2.1):

$$f_{best} = 1 + (N-1)(3+3+1+(N/2)2+7) =$$
$$= N^2 + 14N - N - 14 + 1 = N^2 + 13N - 13 = O(N^2) \quad (4.4)$$

Трудоёмкость в худшем случае (4.2.1):

$$f_{worst} = 1 + (N-1)(3+3+1+(N/2)(2+3)+7) =$$
$$2.5N^2 + 14N - 2.5N - 14 + 1 = 2.5N^2 + 11.5N - 13 = O(N^2) \quad (4.5)$$

4.2.2 Алгоритм гномьей сортировки

Трудоемкость в лучшем случае (4.6):

$$f_{best} = 1 + N(4+1) = 5N + 1 = O(N) \quad (4.6)$$

Трудоёмкость в худшем случае (4.7):

$$f_{worst} = 1 + N(4 + (N-1) * (7+2)) = 9N^2 - 5N + 1 = O(N^2) \quad (4.7)$$

4.2.3 Алгоритм сортировки Шелла

Трудоемкость данного алгоритма может быть рассчитана с использованием той же модели подсчета трудоемкости.

Трудоемкость алгоритма сортировки Шелла:

- в лучшем случае - $O(Nlog^2N)$;
- в худшем случае -$O(N^2)$.

4.3 Технические характеристики

Проведем сравнительный анализ алгоритмов сортировки по затраченному процессорному времени.

Тестирование проводилось на устройстве со следующими техническими характеристиками:

- операционная система: Ubuntu 20.04.1 Linux x86_64 [4];
- память : 8 GiB;
- процессор: AMD® Ryzen™ 3 3200u @ 2.6 GHz [5].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.4 Демонстрация работы программы

На рисунке 4.1 приведен пример работы программы.

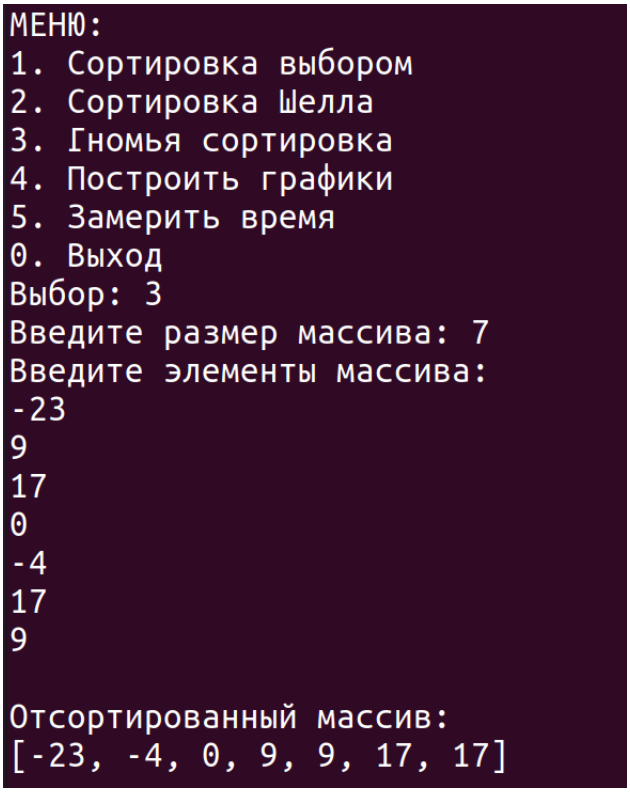

Рисунок 4.1 – Пример работы программы

4.5 Время выполнения алгоритмов

Функция process_time из библиотеки time ЯП Python возвращает процессорное время в секундах - значение типа float.

Для замера времени:

- получить значение времени до начала сортировки, затем после её окончания. Чтобы получить результат, необходимо вычесть из второго значения первое;
- первый шаг необходимо повторить iters раз (в программе iters равно 1500), суммируя полученные значения, а затем усреднить результат.

Результаты замеров времени работы алгоритмов в миллисекундах приведены в таблицах 4.1, 4.2, 4.3.

Таблица 4.1 – На входе отсортированный массив

Размер	Выбором	Шелла	Гномья
100	0.1872	0.4463	0.0107
200	0.6648	1.5769	0.0208
300	1.7652	4.2090	0.0315
400	2.9205	6.5499	0.0415
500	4.7658	9.9785	0.0527
600	6.9592	17.5923	0.0610
700	9.5135	23.0703	0.0695
800	12.3131	26.6032	0.0836
900	15.3724	32.2987	0.0905
1000	18.7373	39.5696	0.0996

Таблица 4.2 – На входе отсортированный в обратном порядке массив

Размер	Выбором	Шелла	Гномья
100	0.2276	0.4635	1.4289
200	0.8215	1.6202	5.4438
300	2.1278	4.2667	12.9537
400	3.5504	6.6461	23.6392
500	5.6489	10.0919	37.7889
600	8.3033	17.7157	55.5283
700	11.2540	23.2613	77.0185
800	14.8231	26.7550	102.4330
900	18.7517	33.3306	129.7009
1000	23.4290	41.0557	161.5674

Таблица 4.3 – На входе случайный массив

Размер	Выбором	Шелла	Гномья
100	0.2079	0.4813	0.7389
200	0.7309	1.7043	3.0184
300	1.8340	4.3971	6.5836
400	3.1248	6.9417	12.2080
500	4.9933	10.3987	19.7805
600	7.3488	18.3833	29.6323
700	10.0685	23.9719	40.9918
800	13.3175	27.9766	54.7456
900	17.1284	34.7869	70.0897
1000	20.7750	41.5146	87.0879

На рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров времени работы сортировок от длины входного массива в трех случаях: на входе отсортированный массив, отсортированный в обратном порядке и массив, заполненный случайным образом.

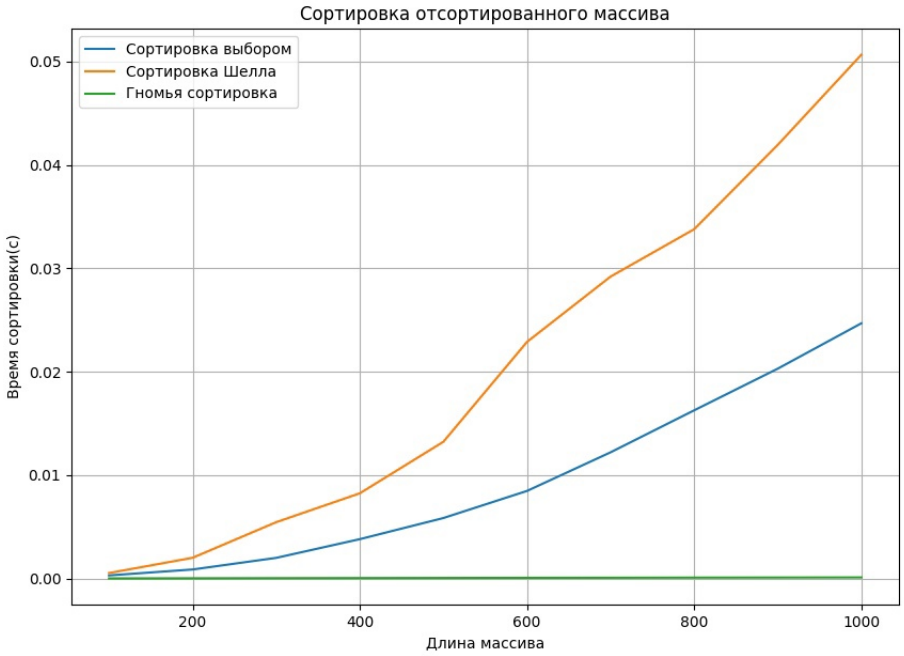

Рисунок 4.2 – На входе отсортированный массив

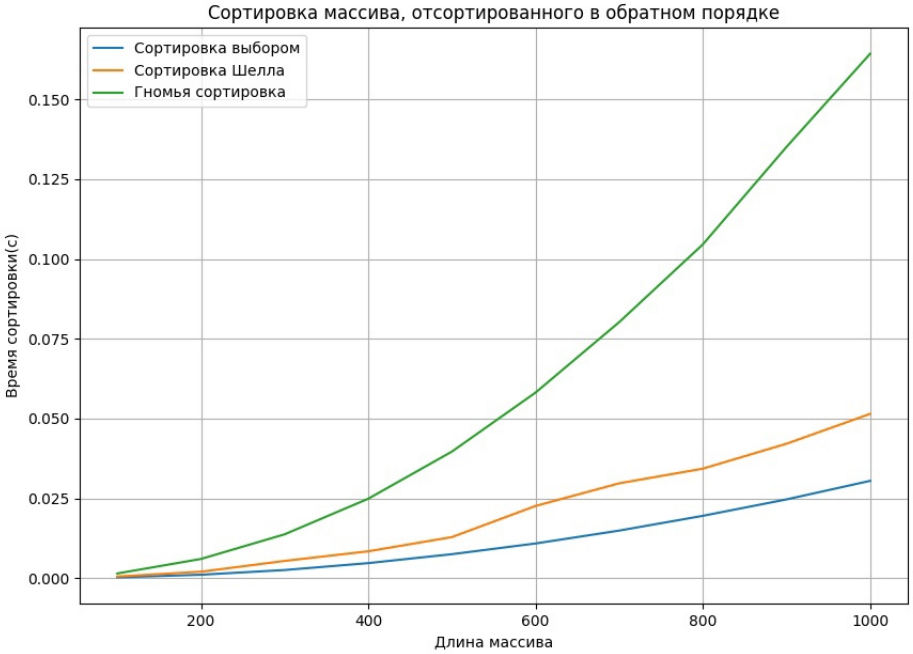

Рисунок 4.3 – На входе отсортированный в обратном порядке массив

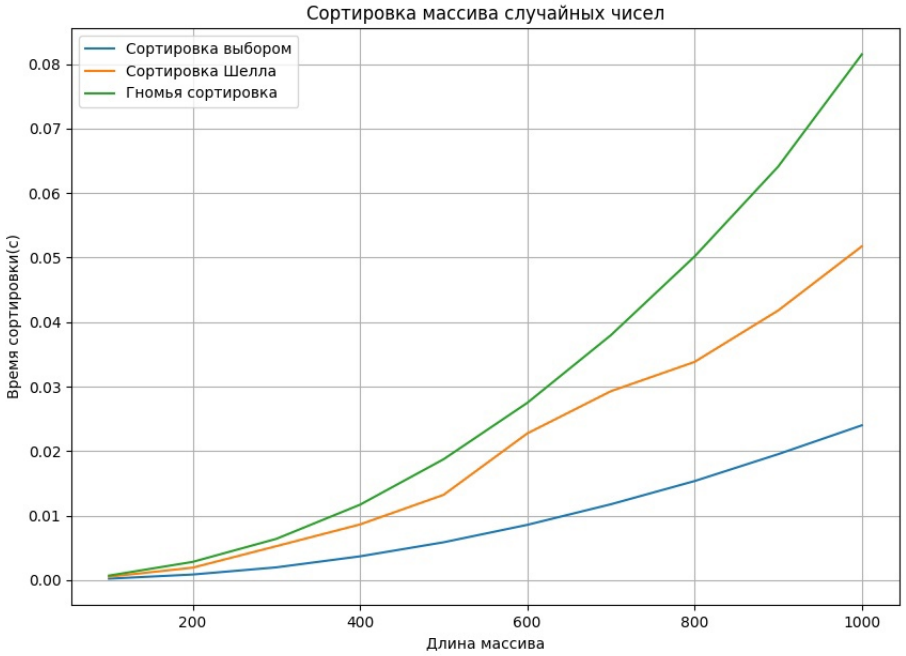

Рисунок 4.4 – На входе заполненный случайно массив

Вывод

Сортировка выбором работает быстрее при сортировке случайно заполненного массива и обратно отсортированного массива. Гномья сортировка в этих случаях работает дольше всех. При этом в случае отсортированного массива гномья сортировка оказывается самой быстрой, самой медленной оказывается сортировка Шелла.

Результаты временных замеров совпадают с вычисленными трудоемкостями.

Заключение

В ходе лабораторной работы были решены следующие задачи:

- были изучены три алгоритма сортировки: выбором, Шелла и гномья;
- были разработаны и реализованы указанные алгоритмы;
- была протестирована реализация рассматриваемых алгоритмов;
- был проведен сравнительный анализ реализованных алгоритмов по затраченному процессорному времени, по трудоемкости и по памяти.

Таким образом, поставленная цель достигнута.

Список литературы

[1] Вычисление расстояния Левенштейна [Электронный ресурс]. Режим доступа: https://foxford.ru/wiki/informatika/vychislenie-rasstoyaniya-levenshteyna (дата обращения: 18.10.2021).

[2] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 18.10.2021).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 18.10.2021).

[4] Ubuntu 20.04 LTS (Focal Fossa) Beta [Электронный ресурс]. Режим доступа: http://old-releases.ubuntu.com/releases/20.04.1/ (дата обращения: 18.10.2021).

[5] Мобильный процессор AMD Ryzen™ 3 3200U с графикой Radeon™ Vega 3 [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-3-3200u (дата обращения: 18.10.2021).