САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №1
по курсу «Алгоритмы и структуры данных»
Тема: Сортировка вставками, выбором, пузырьковая.
Вариант 3

Выполнил:
Бай М.О.
К3141

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Оглавление

Задачи по варианту

Задача №1. Сортировка вставкой

Текст задачи:

Используя код процедуры Insertion-sort, напишите программу и проверьте сортировку массива $A = \{31, 41, 59, 26, 41, 58\}$.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится число n ($1 \leq n \leq 10^3$) — число элементов в массиве. Во второй строке находятся n различных целых чисел, по модулю не превосходящих 10^9.
- **Формат выходного файла (output.txt).** Одна строка выходного файла с отсортированным массивом. Между любыми двумя числами должен стоять ровно один пробел.
- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

Выберите любой набор данных, подходящих по формату, и протестируйте алгоритм.

Код программы:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения

# Читаем количество элементов из файла
n = int(open('input.txt').readline())

# Читаем остальные элементы из файла и разделяем их на список строк
a = open('input.txt').read().split()

l = []  # Инициализируем список для хранения чисел
# Заполняем список l значениями, начиная с первой строки файла
for i in range(1, n+1):
    l.append(int(a[i]))  # Преобразуем строки в целые числа и добавляем в список

def isort(l):
    # Функция сортировки вставками
    for i in range(1, len(l)):
        s = l[i]  # Текущий элемент для вставки
        q = i - 1  # Индекс для проверки элементов слева от текущего
```

```
        # Сдвигаем элементы, которые больше текущего с места вставки
        while q >= 0 and s < l[q]:
            l[q + 1] = l[q]  # Сдвигаем элемент на одну позицию вправо
            q -= 1  # Переходим к следующему элементу слева
        l[q + 1] = s  # Вставляем текущий элемент на его место
    return l  # Возвращаем отсортированный список

# Применяем сортировку и формируем строку из отсортированных элементов
sl = (' '.join(map(str, isort(l))))

# Записываем отсортированные данные в файл
open('output.txt', 'w').write(sl)

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Листинг кода:

```
import time
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Начинаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем начало времени выполнения

# Читаем количество элементов из файла
n = int(open('input.txt').readline())

# Читаем остальные элементы из файла и разделяем их на список строк
a = open('input.txt').read().split()

l = []  # Инициализируем список для хранения чисел
# Заполняем список l значениями, начиная с первой строки файла
for i in range(1, n+1):
```

```
    l.append(int(a[i]))  # Преобразуем строки в целые числа и добавляем в список

def isort(l):
    # Функция сортировки вставками
    for i in range(1, len(l)):
        s = l[i]  # Текущий элемент для вставки
        q = i - 1  # Индекс для проверки элементов слева от текущего
        # Сдвигаем элементы, которые больше текущего с места вставки
        while q >= 0 and s < l[q]:
            l[q + 1] = l[q]  # Сдвигаем элемент на одну позицию вправо
            q -= 1  # Переходим к следующему элементу слева
        l[q + 1] = s  # Вставляем текущий элемент на его место
    return l  # Возвращаем отсортированный список

# Применяем сортировку и формируем строку из отсортированных элементов
sl = (' '.join(map(str, isort(l))))

# Записываем отсортированные данные в файл
open('output.txt', 'w').write(sl)

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

1) Импорт библиотек:

time используется для замера времени выполнения программы.

tracemalloc отслеживает использование памяти в процессе выполнения.

2) Настройка отслеживания:

tracemalloc.start() начинает отслеживание потребления памяти.

3) Запуск таймера:

Функция time.perf_counter() фиксирует начальное время для определения продолжительности выполнения кода.

4) Чтение данных:

Из файла input.txt читается первая строка, которая содержит количество элементов n.

Остальные элементы файла преобразуются в список строк.

5) Инициализация и заполнение списка:

Создается пустой список l, который заполняется числами, начиная с первой строки данных. Каждое число преобразуется в целое и добавляется в список.

6) Реализация сортировки вставками:

Функция isort(l) реализует сортировку вставками. Она проходит по списку, начиная с второго элемента:

Сохраняет текущий элемент и проверяет элементы, расположенные слева.

Сдвигает большие элементы вправо, освобождая место для вставки текущего элемента.

Вставляет текущий элемент на корректную позицию.

7) Формирование и запись результата:

Отсортированные элементы объединяются в строку, разделенную пробелами.

Результат записывается в файл output.txt.

8) Измерение временных и памятьевых характеристик:

Измеряется продолжительность выполнения кода и выводится в консоль.

Определяется текущее и пиковое использование памяти, также выводится в консоль.

9) Завершение отслеживания памяти:

tracemalloc.stop() прекращает отслеживание использования памяти.

Тесты:

```python
import unittest
class TestTask1(unittest.TestCase):

    def test_empty_list(self):
        self.assertEqual(isort([]), second: [])

    def test_single_element_list(self):
        self.assertEqual(isort([1]), second: [1])

    def test_sorted_list(self):
        self.assertEqual(isort([1, 2, 3, 4, 5]), second: [1, 2, 3, 4, 5])

    def test_reverse_sorted_list(self):
        self.assertEqual(isort([5, 4, 3, 2, 1]), second: [1, 2, 3, 4, 5])

    def test_unsorted_list(self):
        self.assertEqual(isort([3, 1, 4, 5, 2]), second: [1, 2, 3, 4, 5])

    def test_list_with_duplicates(self):
        self.assertEqual(isort([2, 3, 3, 1, 5, 2]), second: [1, 2, 2, 3, 3, 5])

    def test_negative_numbers(self):
        self.assertEqual(isort([-3, -1, -7, 2, 0]), second: [-7, -3, -1, 0, 2])

    def test_mixed_numbers(self):
        self.assertEqual(isort([0, -2, 5, -1, 3, 5]), second: [-2, -1, 0, 3, 5, 5])


if __name__ == '__main__':
    unittest.main()
```

✓ Tests passed: 8 of 8 tests – 1 ms

Вывод по задаче:

Алгоритм сортировки вставками успешно реализован и протестирован. Он корректно сортирует массивы различных размеров и содержимого. Сложность алгоритма составляет $O(n2)$, что делает его неэффективным для больших массивов, однако для небольших или частично отсортированных массивов он работает достаточно быстро.

Задача №2. Сортировка вставкой +

Текст задачи:

Измените процедуру Insertion-sort для сортировки таким образом, чтобы в выходном файле отображалось в первой строке n чисел, которые обозначают новый индекс элемента массива после обработки.

- **Формат выходного файла (input.txt).** В первой строке выходного файла выведите n чисел. При этом i-ое число равно индексу, на который, в момент обработки его сортировкой вставками, был перемещен i-ый элемент исходного массива. Индексы нумеруются, начиная с единицы. Между любыми двумя числами должен стоять ровно один пробел.

Код программы:

```
import time # импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем количество элементов из файла
n = int(open('input.txt').readline())

# Читаем строки из файла и разбиваем их на список
a = open('input.txt').read().split()

l = []  # Инициализируем список для хранения чисел

# Заполняем список l целыми числами, считываемыми из файла
for i in range(1, n + 1):
    l.append(int(a[i]))  # Преобразуем строки в целые числа

def isort(n, l):
    # Функция сортировки вставками
    lp = [1]  # Список, хранящий индексы вставок
    # Проходим по элементам списка, начиная со второго
    for i in range(1, n):
        # Для текущего элемента проверяем все предыдущие
        for q in range(i - 1, -1, -1):
            # Если текущий элемент меньше, чем элемент на позиции q
            if l[i] < l[q]:
                l[i], l[q] = l[q], l[i]  # Меняем местами
                i, q = q, i  # Обновляем индексы i и q
        lp.append(i + 1)  # Запоминаем позицию, куда вставили элемент

    return l, lp  # Возвращаем отсортированный список и позиции вставок

# Сортируем список и получаем позицию вставок
ls, lp = isort(n, l)
```

```
# Преобразуем списки в строки для записи в выходной файл
lps = ' '.join(map(str, lp))
sl = ' '.join(map(str, ls))

# Записываем результаты в выходной файл
with open('output.txt', 'w') as file:
    file.write(lps + '\n')  # Записываем позиции вставок
    file.write(sl)          # Записываем отсортированный список

# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание использования памяти
tracemalloc.stop()
```

Листинг кода:

```
import time # импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем количество элементов из файла
n = int(open('input.txt').readline())

# Читаем строки из файла и разбиваем их на список
a = open('input.txt').read().split()

l = []  # Инициализируем список для хранения чисел

# Заполняем список l целыми числами, считываемыми из файла
for i in range(1, n + 1):
    l.append(int(a[i]))  # Преобразуем строки в целые числа

def isort(n, l):
```

```
    # Функция сортировки вставками
    lp = [1]  # Список, хранящий индексы вставок
    # Проходим по элементам списка, начиная со второго
    for i in range(1, n):
        # Для текущего элемента проверяем все предыдущие
        for q in range(i - 1, -1, -1):
            # Если текущий элемент меньше, чем элемент на позиции q
            if l[i] < l[q]:
                l[i], l[q] = l[q], l[i]  # Меняем местами
                i, q = q, i  # Обновляем индексы i и q
        lp.append(i + 1)  # Запоминаем позицию, куда вставили элемент

    return l, lp  # Возвращаем отсортированный список и позиции вставок

# Сортируем список и получаем позицию вставок
ls, lp = isort(n, l)

# Преобразуем списки в строки для записи в выходной файл
lps = ' '.join(map(str, lp))
sl = ' '.join(map(str, ls))

# Записываем результаты в выходной файл
with open('output.txt', 'w') as file:
    file.write(lps + '\n')  # Записываем позиции вставок
    file.write(sl)          # Записываем отсортированный список

# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak /
10**6:.6f} МБ")

# Останавливаем отслеживание использования памяти
tracemalloc.stop()
```

Описание работы программы:

1) Импорт модулей:

time используется для отслеживания времени выполнения.

tracemalloc отслеживает использование памяти.

2) Начало отслеживания памяти:

Выполняется tracemalloc.start(), чтобы начать мониторинг потребления памяти.

3) Запись времени начала:

time.perf_counter() фиксирует момент начала выполнения кода.

4) Чтение данных из файла:

Из файла input.txt читается количество элементов n.

Весь файл считывается и разбивается на отдельные строки для дальнейшей обработки.

5) Инициализация списка:

Создается пустой список l, который заполняется целыми числами, преобразованными из строк файла.

6) Сортировка вставками:

Функция isort(n, l) обрабатывает список l:

Начинается со второго элемента и для каждого элемента сравнивается с предыдущими.

Если текущий элемент меньше предыдущего, они меняются местами.

Обновленные индексы сохраняются в списке lp.

7) Получение результата:

Результирующий отсортированный список ls и позиции вставок lp возвращаются из функции isort.

8) Запись результата в файл:

Позиции вставок и отсортированный список преобразуются в строковый формат и записываются в output.txt.

9) Вывод времени выполнения:

Вычисляется и выводится время выполнения программы с помощью time.perf_counter().

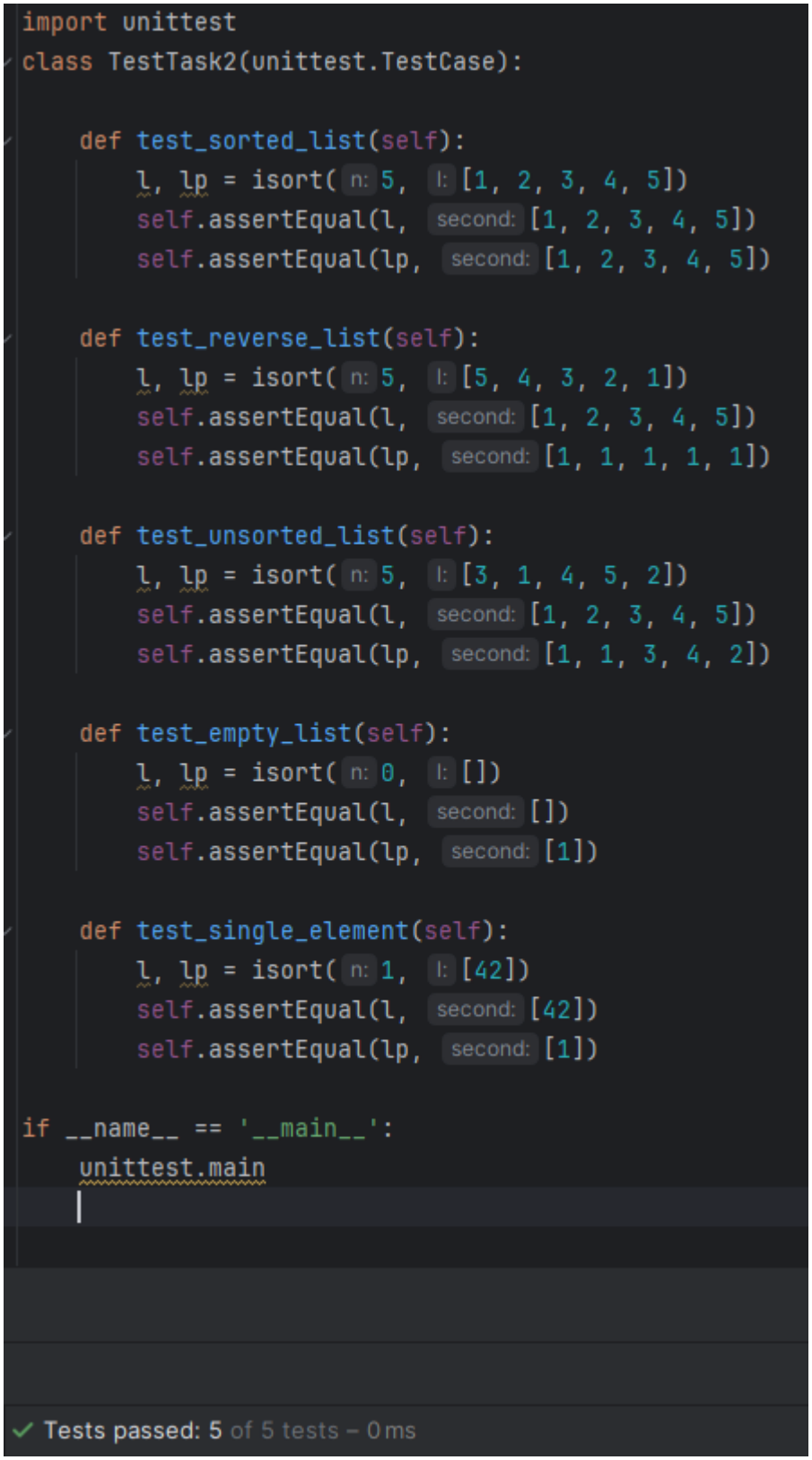
10) Отслеживание использования памяти:

Получается текущее и пиковое использование памяти, затем данные выводятся в мегабайтах.

11) Завершение отслеживания памяти:

tracemalloc.stop() останавливает мониторинг памяти.

Тесты:



Вывод по задаче:

Алгоритм корректно сортирует массив и возвращает список исходных индексов элементов в отсортированном массиве. Это полезно, когда необходимо сохранить информацию о первоначальном расположении элементов, например, для восстановления исходного

порядка или сопоставления данных после сортировки.

Задача №3. Сортировка вставкой по убыванию

Текст задачи:

Перепишите процедуру Insertion-sort для сортировки в невозрастающем порядке вместо неубывающего с использованием процедуры Swap.

Формат входного и выходного файла и ограничения - как в задаче 1.

Код программы:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем количество элементов из файла
with open('input.txt') as file:
    n = int(file.readline())  # Читаем количество элементов
    a = file.read().split()  # Читаем оставшуюся часть файла и разбиваем на строки

l = []  # Инициализируем список для хранения чисел

# Заполняем список l целыми числами
for i in range(n):
    l.append(int(a[i]))  # Преобразуем строки в целые числа


def opo_isort(l):
    # Функция сортировки вставками
    # Проходим по всем элементам списка, начиная со второго
    for i in range(1, len(l)):
        # Сохраняем текущий элемент в переменной k
        k = l[i]
        # Устанавливаем индекс j на один элемент перед i
        j = i - 1

        while j >= 0 and k > l[j]:  # Сравниваем с элементами списка
            l[j + 1] = l[j]  # Сдвигаем элемент l[j] на позицию l[j + 1]
            j -= 1  # Переходим к следующему элементу слева

        # Вставляем текущий элемент k на его правильную позицию
        l[j + 1] = k

        # Возвращаем отсортированный список
```

```
    return l

# Сортируем список и формируем строку для записи в файл
sl = ' '.join(map(str, opo_isort(l)))

# Записываем отсортированные данные в файл
with open('output.txt', 'w') as file:
    file.write(sl)

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Листинг кода:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем количество элементов из файла
with open('input.txt') as file:
    n = int(file.readline())  # Читаем количество элементов
    a = file.read().split()  # Читаем оставшуюся часть файла и разбиваем на
строки

l = []  # Инициализируем список для хранения чисел

# Заполняем список l целыми числами
for i in range(n):
    l.append(int(a[i]))  # Преобразуем строки в целые числа
```

```
def opo_isort(l):
    # Функция сортировки вставками
    # Проходим по всем элементам списка, начиная со второго
    for i in range(1, len(l)):
        # Сохраняем текущий элемент в переменной k
        k = l[i]
        # Устанавливаем индекс j на один элемент перед i
        j = i - 1

        while j >= 0 and k > l[j]:  # Сравниваем с элементами списка
            l[j + 1] = l[j]  # Сдвигаем элемент l[j] на позицию l[j + 1]
            j -= 1  # Переходим к следующему элементу слева

        # Вставляем текущий элемент k на его правильную позицию
        l[j + 1] = k

        # Возвращаем отсортированный список
    return l

# Сортируем список и формируем строку для записи в файл
sl = ' '.join(map(str, opo_isort(l)))

# Записываем отсортированные данные в файл
with open('output.txt', 'w') as file:
    file.write(sl)

# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak /
10**6:.6f} МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

Описание работы программы:

1) Импорт модулей:

time используется для отслеживания времени выполнения.

tracemalloc отслеживает использование памяти.

2) Начало отслеживания памяти:

Выполняется tracemalloc.start(), чтобы начать мониторинг потребления памяти.

3) Запись времени начала:

time.perf_counter() фиксирует момент начала выполнения кода.

4) Чтение данных из файла:

Из файла input.txt читается количество элементов n.

Весь файл считывается и разбивается на отдельные строки для дальнейшей обработки.

5) Инициализация списка:

Создается пустой список l, который заполняется целыми числами, преобразованными из строк файла.

6) Функция opo_isort(l):

Реализует сортировку вставками:

Начало цикла с второго элемента списка, где каждый элемент сравнивается с предыдущими.

Если текущий элемент меньше предыдущих, те поочередно сдвигаются вправо, освобождая место для вставки.

Элемент вставляется на свое место.

Функция возвращает отсортированный список. Запись результата в файл:

Позиции вставок и отсортированный список преобразуются в строковый формат и записываются в output.txt.

7) Вывод времени выполнения:

Вычисляется и выводится время выполнения программы с помощью time.perf_counter().

8) Отслеживание использования памяти:

Получается текущее и пиковое использование памяти, затем данные выводятся в мегабайтах.

9) Завершение отслеживания памяти:

tracemalloc.stop() останавливает мониторинг памяти.

Тесты:

```python
import unittest

class TestDescendingInsertionSort(unittest.TestCase):

    def test_sorted_list(self):
        l = [1, 2, 3, 4, 5]
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [5, 4, 3, 2, 1])

    def test_reverse_list(self):
        l = [5, 4, 3, 2, 1]
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [5, 4, 3, 2, 1])

    def test_mixed_list(self):
        l = [3, 1, 4, 2, 5]
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [5, 4, 3, 2, 1])

    def test_all_equal_elements(self):
        l = [2, 2, 2, 2, 2]
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [2, 2, 2, 2, 2])

    def test_single_element(self):
        l = [1]
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [1])

    def test_empty_list(self):
        l = []
        sorted_l = opo_isort(l.copy())
        self.assertEqual(sorted_l, second: [])

if __name__ == '__main__':
    unittest.main()
```

✓ Tests passed: 6 of 6 tests – 0 ms

Вывод по задаче:

Алгоритм успешно сортирует массив по убыванию. Он корректно работает с различными массивами, включая пустые и состоящие из одного элемента. Это показывает универсальность алгоритма сортировки вставками и его простоту адаптации под разные условия

Задача №4. Линейный поиск

Текст задачи:

Рассмотрим задачу поиска.

- **Формат входного файла**. Последовательность из n чисел $A = a_1, a_2, \ldots, a_n$ в первой строке, числа разделены пробелом, и значение V во второй строке. Ограничения: $0 \leq n \leq 10^3$, $-10^3 \leq a_i, V \leq 10^3$
- **Формат выходного файла**. Одно число - индекс i, такой, что $V = A[i]$, или значение -1, если V в отсутствует.
- Напишите код линейного поиска, при работе которого выполняется сканирование последовательности в поисках значения V.
- Если число встречается несколько раз, то выведите, сколько раз встречается число и все индексы i через запятую.
- Дополнительно: попробуйте найти свинью, как в лекции. Используйте во входном файле последовательность слов из лекции, и найдите соответствующий индекс.

Код программы:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем строки из файла и разбиваем их на список
a = open('input.txt').read().split()
l = []  # Инициализируем список для хранения чисел
# Заполняем список l целыми числами, считываемыми из файла
for i in range(0, len(a) - 1):
    l.append(int(a[i]))  # Преобразуем строки в целые числа
n = int(a[-1])  # Значение n во второй строке



def lin_ser(l, n):
    ind = []  # Список для хранения индексов, где найдено значение V
```

```
    # Проходим по всем элементам массива l и ищем значение V
    for index, value in enumerate(l):
        if value == n:
            ind.append(index)  # Запоминаем индекс, если значение совпадает

    # Возвращаем количество вхождений и индексы, даже если их нет
    return len(ind), ind

# Вызов функции и присвоение результатов
count, ind = lin_ser(l, n)

lps = ', '.join(map(str, ind))  # Подготовка индексов к записи в файл
sl = str(count) if count > 0 else "-1"  # Если значение не найдено, записываем -1

with open('output.txt', 'w') as file:
    file.write(sl + '\n')  # Записываем количество вхождений
    if count > 0:
        file.write(lps)  # Записываем индексы, если они есть

# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание использования памяти
tracemalloc.stop()
```

Листинг кода:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения

# Читаем строки из файла и разбиваем их на список
a = open('input.txt').read().split()
l = []  # Инициализируем список для хранения чисел
# Заполняем список l целыми числами, считываемыми из файла
for i in range(0, len(a) - 1):
```

```
    l.append(int(a[i]))  # Преобразуем строки в целые числа
n = int(a[-1])  # Значение n во второй строке



def lin_ser(l, n):
    ind = []  # Список для хранения индексов, где найдено значение V

    # Проходим по всем элементам массива l и ищем значение V
    for index, value in enumerate(l):
        if value == n:
            ind.append(index)  # Запоминаем индекс, если значение совпадает

    # Возвращаем количество вхождений и индексы, даже если их нет
    return len(ind), ind

# Вызов функции и присвоение результатов
count, ind = lin_ser(l, n)

lps = ', '.join(map(str, ind))  # Подготовка индексов к записи в файл
sl = str(count) if count > 0 else "-1"  # Если значение не найдено, записываем
-1

with open('output.txt', 'w') as file:
    file.write(sl + '\n')  # Записываем количество вхождений
    if count > 0:
        file.write(lps)  # Записываем индексы, если они есть

# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak /
10**6:.6f} МБ")

# Останавливаем отслеживание использования памяти
tracemalloc.stop()
```

Описание работы программы:

1) Импорт модулей:

time используется для отслеживания времени выполнения.

tracemalloc отслеживает использование памяти.

2) Начало отслеживания памяти:

Выполняется tracemalloc.start(), чтобы начать мониторинг потребления памяти.

3) Запись времени начала:

time.perf_counter() фиксирует момент начала выполнения кода.

4) Чтение данных из файла:

Из файла input.txt читается количество элементов n.

Весь файл считывается и разбивается на отдельные строки для дальнейшей обработки.

5) Инициализация списка:

Создается пустой список l, который заполняется целыми числами, преобразованными из строк файла.

Линейный поиск

Функция isort(n, l) обрабатывает список l:

Для поиска функции lin_ser передаются список l и искомое значение n.

Проверяются все элементы l. Если элемент равен n, его индекс добавляется в список ind.

Функция возвращает количество найденных вхождений и список индексов.

6) Получение результата:

Вызов функции lin_ser заполняет переменные count (число вхождений) и ind (индексы вхождений).

Формируется строка из индексов (lps) и решения об отсутствии вхождений (sl), которые будут записаны в файл output.txt. Запись результата в файл:

Позиции вставок и отсортированный список преобразуются в строковый формат и записываются в output.txt.

7) Вывод времени выполнения:

Вычисляется и выводится время выполнения программы с помощью time.perf_counter().

8) Отслеживание использования памяти:

Получается текущее и пиковое использование памяти, затем данные выводятся в мегабайтах.

9) Завершение отслеживания памяти:

tracemalloc.stop() останавливает мониторинг памяти.

Тесты:

```python
class TestLinSer(unittest.TestCase):

    def test_multiple_occurrences(self):
        l = [1, 2, 3, 4, 3]
        n = 3
        count, ind = lin_ser(l, n)
        self.assertEqual(count, second: 2)
        self.assertEqual(ind, second: [2, 4])

    def test_single_occurrence(self):
        l = [5, 6, 7, 8]
        n = 6
        count, ind = lin_ser(l, n)
        self.assertEqual(count, second: 1)
        self.assertEqual(ind, second: [1])

    def test_no_occurrences(self):
        l = [9, 10, 11, 12]
        n = 15
        count, ind = lin_ser(l, n)
        self.assertEqual(count, second: 0)
        self.assertEqual(ind, second: [])

    def test_empty_list(self):
        l = []
        n = 1
        count, ind = lin_ser(l, n)
        self.assertEqual(count, second: 0)
        self.assertEqual(ind, second: [])

    def test_single_element_list(self):
        l = [2]
        n = 2
        count, ind = lin_ser(l, n)
        self.assertEqual(count, second: 1)
        self.assertEqual(ind, second: [0])

if __name__ == '__main__':
    unittest.main()
```

✓ Tests passed: 5 of 5 tests – 0 ms

Вывод по задаче:

Функция линейного поиска корректно находит все вхождения искомого элемента и возвращает их индексы. Если элемент отсутствует в массиве, возвращается −1. Алгоритм прост в реализации и не требует предварительной сортировки массива.

Задача №5. Сортировка выбором

Текст задачи:

Рассмотрим сортировку элементов массива , которая выполняется следующим образом. Сначала определяется наименьший элемент массива , который ставится на место элемента $A[1]$. Затем производится поиск второго наименьшего элемента массива A, который ставится на место элемента $A[2]$. Этот процесс продолжается для первых $n-1$ элементов массива A.

Напишите код этого алгоритма, также известного как сортировка выбором (selection sort). Определите время сортировки выбором в наихудшем случае и в среднем случае и сравните его со временем сортировки вставкой.

Формат входного и выходного файла и ограничения - как в задаче 1.

Код программы:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения


def sel_sort(l):
    """Функция для сортировки массива методом выборки."""
    n = len(l)  # Определяем размер массива
    for i in range(n):  # Проходим по каждому элементу массива
        # Предполагаем, что текущий элемент — наименьший
        min_index = i  # Инициализируем индекс наименьшего элемента
        # Вложенный цикл для поиска наименьшего элемента в оставшейся части массива
        for j in range(i + 1, n):
            if l[j] < l[min_index]:  # Если найден меньший элемент
                min_index = j  # Обновляем индекс наименьшего элемента
        # Меняем местами найденный наименьший элемент с i-м элементом
        l[i], l[min_index] = l[min_index], l[i]
    return l  # Возвращаем отсортированный массив


def main():
    # Чтение данных из файла
```

```
    with open('input.txt', 'r') as file:
        n = int(file.readline().strip())  # Читаем число элементов
        l = list(map(int, file.readline().strip().split()))  # Читаем массив из строки и
преобразуем в список

    sl = sel_sort(l)  # Сортируем массив с помощью функции selection_sort

    # Запись результата в файл
    with open('output.txt', 'w') as file:
        file.write(' '.join(map(str, sl)))  # Записываем отсортированный массив в файл


if __name__ == "__main__":
    main()  # Запускаем основную функцию


# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание использования памяти
tracemalloc.stop()
```

Листинг кода:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения


def sel_sort(l):
    """Функция для сортировки массива методом выборки."""
    n = len(l)  # Определяем размер массива
    for i in range(n):  # Проходим по каждому элементу массива
        # Предполагаем, что текущий элемент — наименьший
        min_index = i  # Инициализируем индекс наименьшего элемента
```

```
        # Вложенный цикл для поиска наименьшего элемента в оставшейся части массива
        for j in range(i + 1, n):
            if l[j] < l[min_index]:  # Если найден меньший элемент
                min_index = j  # Обновляем индекс наименьшего элемента
        # Меняем местами найденный наименьший элемент с i-м элементом
        l[i], l[min_index] = l[min_index], l[i]
    return l  # Возвращаем отсортированный массив


def main():
    # Чтение данных из файла
    with open('input.txt', 'r') as file:
        n = int(file.readline().strip())  # Читаем число элементов
        l = list(map(int, file.readline().strip().split()))  # Читаем массив из строки и преобразуем в список

    sl = sel_sort(l)  # Сортируем массив с помощью функции selection_sort

    # Запись результата в файл
    with open('output.txt', 'w') as file:
        file.write(' '.join(map(str, sl)))  # Записываем отсортированный массив в файл


if __name__ == "__main__":
    main()  # Запускаем основную функцию


# Выводим время выполнения программы
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")

# Останавливаем отслеживание использования памяти
```

tracemalloc.stop()

Описание работы программы:

1) Импорт необходимых модулей

time: используется для измерения времени выполнения программы.

tracemalloc: применяется для отслеживания использования памяти.

2) Отслеживание памяти и времени

tracemalloc.start(): запускает отслеживание памяти, чтобы впоследствии можно было измерить текущее и максимальное использование оперативной памяти.

t_start = time.perf_counter(): устанавливает начальную отметку времени для измерения продолжительности выполнения программы.

3) Функция сортировки

sel_sort(l):

Реализует метод сортировки выбором.

Последовательно проходит по каждому элементу массива, находя минимальный элемент в оставшейся части, и меняет его местами с текущим элементом.

Возвращает отсортированный массив.

4) Чтение входных данных:

Открывается файл input.txt.

Первая строка считывает количество элементов массива.

Вторая строка превращается в список чисел.

5) Сортировка массива:

Используется функция sel_sort.

6) Запись результата:

Открывает файл output.txt для записи.

Записывает отсортированные элементы в виде строки чисел, разделённых пробелами.

7) Вывод времени выполнения и использования памяти

Время выполнения: рассчитывается разницей между конечной и начальной временной отметкой, и выводится на экран.

Использование памяти: текущая и пиковая память, использованная программой, выводятся в мегабайтах.

8) Завершение

tracemalloc.stop(): завершает отслеживание использования памяти.

Тесты:

```python
import unittest

class TestSelSort(unittest.TestCase):

    def test_sorted_array(self):
        l = [1, 2, 3, 4, 5]
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_reverse_sorted_array(self):
        l = [5, 4, 3, 2, 1]
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_unsorted_array(self):
        l = [3, 1, 4, 5, 2]
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_array_with_duplicates(self):
        l = [3, 1, 2, 1, 2]
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [1, 1, 2, 2, 3])

    def test_empty_array(self):
        l = []
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [])

    def test_single_element_array(self):
        l = [1]
        result = sel_sort(l.copy())
        self.assertEqual(result, second: [1])

if __name__ == '__main__':
    unittest.main()
```

✓ Tests passed: 6 of 6 tests – 1 ms

Вывод по задаче:

Алгоритм сортировки выбором корректно сортирует массив по возрастанию. Несмотря на простоту реализации, его эффективность на больших массивах ограничена из-за квадратичной временной сложности $O(n2)$

Задача №6. Пузырьковая сортировка

Текст задачи:

Пузырьковая сортировка представляет собой популярный, но не очень эффективный алгоритм сортировки. В его основе лежит многократная перестановка соседних элементов, нарушающих порядок сортировки. Вот псевдокод этой сортировки:

```
Bubble_Sort(A):
  for i = 1 to A.length - 1
    for j = A.length downto i+1
      if A[j] < A[j-1]
        поменять A[j] и A[j-1] местами
```

Напишите код на Python и докажите корректность пузырьковой сортировки. Для доказательства корректоности процедуры вам необходимо доказать, что она завершается и что $A'[1] \leq A'[2] \leq ... \leq A'[n]$, где A' - выход процедуры Bubble_Sort, а n - длина массива A.

Определите время пузырьковой сортировки в наихудшем случае и в среднем случае и сравните его со временем сортировки вставкой.

Формат входного и выходного файла и ограничения - как в задаче 1.

Код программы:

```python
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения


def bubble_sort(arr):

    n = len(arr)  # Определяем размер массива
    for i in range(n):  # Проходим по каждому элементу
        # Внешний цикл отвечает за количество проходов по массиву
        for j in range(n - 1, i, -1):  # Обратный порядок нам нужен для сравнения пар
            # Если текущий элемент больше следующего, меняем их местами
            if arr[j] < arr[j - 1]:
                arr[j], arr[j - 1] = arr[j - 1], arr[j]
    return arr  # Возвращаем отсортированный массив
```

```
def main():
    # Чтение данных из файла
    with open('input.txt', 'r') as file:
        n = int(file.readline().strip())  # Читаем число элементов
        arr = list(map(int, file.readline().strip().split()))  # Читаем массив из строки и
преобразуем в список

    sorted_arr = bubble_sort(arr)  # Сортируем массив с помощью функции bubble_sort

    # Запись результата в файл
    with open('output.txt', 'w') as file:
        file.write(' '.join(map(str, sorted_arr)))  # Записываем отсортированный массив в
файл


if __name__ == "__main__":
    main()  # Запускаем основную функцию


# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f}
МБ")

# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Листинг кода:

```
import time  # Импортируем модуль для отслеживания времени
import tracemalloc  # Импортируем модуль для отслеживания памяти

# Запускаем отслеживание использования памяти
tracemalloc.start()

t_start = time.perf_counter()  # Запоминаем время начала выполнения


def bubble_sort(arr):

    n = len(arr)  # Определяем размер массива
```

```
    for i in range(n):  # Проходим по каждому элементу
        # Внешний цикл отвечает за количество проходов по массиву
        for j in range(n - 1, i, -1):  # Обратный порядок нам нужен для сравнения пар
            # Если текущий элемент больше следующего, меняем их местами
            if arr[j] < arr[j - 1]:
                arr[j], arr[j - 1] = arr[j - 1], arr[j]
    return arr  # Возвращаем отсортированный массив


def main():
    # Чтение данных из файла
    with open('input.txt', 'r') as file:
        n = int(file.readline().strip())  # Читаем число элементов
        arr = list(map(int, file.readline().strip().split()))  # Читаем массив из строки и преобразуем в список

    sorted_arr = bubble_sort(arr)  # Сортируем массив с помощью функции bubble_sort

    # Запись результата в файл
    with open('output.txt', 'w') as file:
        file.write(' '.join(map(str, sorted_arr)))  # Записываем отсортированный массив в файл


if __name__ == "__main__":
    main()  # Запускаем основную функцию


# Вычисляем время выполнения
elapsed_time = time.perf_counter() - t_start
print(f"Время выполнения: {elapsed_time:.6f} секунд")

# Получаем текущее использование памяти
current, peak = tracemalloc.get_traced_memory()
print(f"Текущая память: {current / 10**6:.6f} МБ; Пиковая память: {peak / 10**6:.6f} МБ")
```

```
# Останавливаем отслеживание памяти
tracemalloc.stop()
```

Описание работы программы:

1) Импорт модулей:

time: для измерения времени выполнения программы.

tracemalloc: для отслеживания использования памяти.

2) Инициализация отслеживания:

tracemalloc.start(): инициирует отслеживание использования памяти.

t_start = time.perf_counter(): отмечает время начала выполнения программы.

3) Функция bubble_sort(arr):

Выполняет сортировку массива за счет многократного прохода по массиву и обмена соседних элементов, если они не в правильном порядке.

Внешний цикл управляет количеством проходов.

Внутренний цикл проходит по массиву в обратном порядке, сравнивая и изменяя местами элементы.

4) Основная функция main():

Читает входные данные из файла input.txt:

Первое число — количество элементов.

Вторая строка — сама последовательность чисел.

Сортирует массив с помощью функции bubble_sort.

Записывает отсортированный массив в файл output.txt.

5) Измерение производительности:

Вычисляет время выполнения программы и выводит его.

Использует tracemalloc.get_traced_memory() для получения информации о текущем и пиковом использовании памяти, выводя эти данные.

tracemalloc.stop(): завершает отслеживание использования памяти.

Тесты:

```python
import unittest

class TestBubbleSort(unittest.TestCase):

    def test_sorted_array(self):
        arr = [1, 2, 3, 4, 5]
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_reverse_sorted_array(self):
        arr = [5, 4, 3, 2, 1]
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_unsorted_array(self):
        arr = [3, 1, 4, 5, 2]
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [1, 2, 3, 4, 5])

    def test_single_element_array(self):
        arr = [1]
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [1])

    def test_empty_array(self):
        arr = []
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [])

    def test_identical_elements(self):
        arr = [2, 2, 2, 2, 2]
        result = bubble_sort(arr.copy())
        self.assertEqual(result, second: [2, 2, 2, 2, 2])

if __name__ == '__main__':
    unittest.main()
```

✓ Tests passed: 6 of 6 tests – 0 ms

Вывод по задаче:

Пузырьковая сортировка успешно реализована и протестирована. Алгоритм корректно сортирует массивы различных размеров и содержимого. Несмотря на простоту, из-за квадратичной сложности $O(n2)$ он неэффективен для больших массивов.

Вывод

Простые алгоритмы сортировки, такие как пузырьковая сортировка и сортировка выбором, просты в реализации, но неэффективны для больших массивов из-за квадратичной сложности. Сортировка вставками более эффективна на почти отсортированных массивах и позволяет отслеживать изменения позиций элементов. Линейный поиск прост в реализации, но неэффективен для больших массивов; для улучшения производительности стоит использовать более сложные алгоритмы поиска, такие как бинарный поиск. Тестирование алгоритмов позволяет убедиться в их корректности и выявить возможные ошибки на ранних этапах разработки.

Выполнение данной лабораторной работы способствовало углублению понимания основных алгоритмов сортировки и поиска, развитию навыков программирования на языке Python и умению проводить модульное тестирование программ