МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Пензенский государственный технологический университет»

(ПензГТУ)

Факультет автоматизированных информационных технологий

Кафедра «Информационные технологии и системы»

Дисциплина «Введение в программирование»

КУРСОВАЯ РАБОТА

на тему: «Реализация и исследование алгоритмов сортировок перемешиванием, Шелла, вставками, интросорт»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

ПензГТУ 3.09.03.02.23.57ПЗ

Выполнил: студент группы 23ИС1б Сидоров А.А.

Проверил: к.т.н., доцент каф. ИТС Пискаев К.Ю.

Работа защищена с оценкой: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Пенза 2024

Утверждаю

зав. каф. ИТС Михеев М.Ю.

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_

ЗАДАНИЕ

на курсовую работу по дисциплине

«Введение в программирование»

Студента Сидорова А.А. Группы 23ИС1б

Тема работы: «Реализация и исследование алгоритмов сортировок перемешиванием, Шелла, вставками, интросорт»

Исходные данные (технические требования на работу)

1. Назначение программы:

Сортировка поданного массива различными алгоритмами сортировок и исследования работоспособности и корректности их выполнения;

2. Состав и структура входных данных: Массив числовых данных.

3. Состав и структура выходных данных: Сортированный массив и результаты исследования и корректности выполнения программы;

4. Требования к функционалу разрабатываемого программного обеспечения:

– запуск по требованию пользователя

– вывод результатов исследований алгоритмов сортировок

5. Прочие требования: тестирование работоспособности реализованных алгоритмов с помощью библиотеки *pytest*;

6. Содержание расчетной части:

– исследование зависимости времени и количества операций от длины массива;

– проверка соответствия полученным результатам теоретической сложности алгоритма.

7. Графическая часть:

– схема программы (1 лист формата А4);

8. Экспериментальная часть:

– составление и отладка программы в *Visual Studio Code* *v.* 1.89;

9. Срок выполнения работы по разделам:

Постановка задачи до

Разработка алгоритмов до

Подготовка текста программы до

Написание программы до

Отладка программы до

Оформление графической части до

Оформление пояснительной записки до

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_ /

Задание получил «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_г.

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_ /

Задание выдал «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_г.

Дата защиты работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_ /

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_ /

СОДЕРЖАНИЕ

Изм.

Лист

№ докум.

Подпись

Дата

Лист

2

ПензГТУ 3.09.03.02.57ПЗ

Разраб.

Сидоров А.А.

Провер.

Пискаев М.Ю.

Т. контр

Н.контр.

Утвержд.

аАв

Реализация и исследование… (см. Титульный лист)

Лит.

Листов

47

гр. 23ИС1б

[ВВЕДЕНИЕ 4](#_Toc168489985)

[ГЛАВА 1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 7](#_Toc168489986)

[1.1 Общие сведения 7](#_Toc168489987)

[1.2 Теоретические сведения об алгоритмах 7](#_Toc168489988)

[1.3 Характеристики алгоритмов 10](#_Toc168489989)

[ГЛАВА 2 РЕАЛИЗАЦИЯ АЛГОРИТМОВ И РАЗРАБОТКА УПРАВЛЯЮЩЕЙ ПРОГРАММЫ 13](#_Toc168489990)

[2.1 Общие сведения 13](#_Toc168489991)

[2.2 Выбор языка программирования 13](#_Toc168489992)

[2.3 Реализация функций алгоритмов 15](#_Toc168489993)

[2.4 Тестирование работоспособности и корректности разработанных алгоритмов 20](#_Toc168489994)

[2.5 Разработка программы исследования алгоритмов 23](#_Toc168489995)

[2.6 Выводы по главе 27](#_Toc168489996)

[ГЛАВА 3 ИССЛЕДОВАНИЕ АЛГОРИТМОВ 28](#_Toc168489997)

[3.1 Общие сведения 28](#_Toc168489998)

[3.2 Исследование зависимости количества операций от длины массива 28](#_Toc168489999)

[3.3 Проверка соответствия полученным результатам теоретической сложности алгоритма 31](#_Toc168490000)

[3.4 Исследование устойчивости алгоритма 34](#_Toc168490001)

[3.5 Выводы по главе 35](#_Toc168490002)

[ЗАКЛЮЧЕНИЕ 37](#_Toc168490003)

[БИБЛИОГРАФИЧЕСКИЙ СПИСОК 38](#_Toc168490004)

[ПРИЛОЖЕНИЕ А - СХЕМА ПРОГРАММЫ 40](#_Toc168490005)

[ПРИЛОЖЕНИЕ Б – КОД ПРОГРАММЫ СОРТИРОВОК 41](#_Toc168490006)

[ПРИЛОЖЕНИЕ В - КОД ПРОГРАММЫ 44](#_Toc168490007)

Введение

Тема курсовой работы «Реализация и исследование алгоритмов сортировок» является очень актуальной на сегодняшний день. Объём использования информационных технологий постоянно растет, как и количество информации на всей планете. Информационные технологии решают такую задачу, как правильное, эффективное и удобное сохранение всей имеющейся информации в мире. Для этого были созданы различные хранители данных, которые сохраняют информацию непосредственно в памяти ЭВМ, например, *CD*-диски, внешние и внутренние жёсткие диски, флеш-накопители. Кроме этого, в наше время, всю необходимую информацию, пользователи, также могут сохранять и в интернете. Но просто сохранить всю имеющуюся информацию недостаточно, ее необходимо организовать таким образом, чтобы искать необходимые данные было быстро и удобно. Именно поэтому, изучение алгоритмов сортировок очень полезно в настоящее время.

Из предыдущего абзаца следует, что для работы с большими данными используются алгоритмы сортировок, чтобы упростить их дальнейшую обработку. Также для веб-разработки сортировки являются одним из основных элементов. Ряд математических операций и работа со строками не обходятся без упорядочивания данных. Различные игры, такие как знаменитая головоломка французского математика Эдуарда Люка «Ханойская башня», основываются на понимании алгоритмов сортировок. [1].

Цель: Реализовать и исследовать такие алгоритмы сортировок, как: сортировка перемешиванием, сортировка Шелла, сортировка выбором, сортировка интросорт.

Задачи:

1. Рассмотреть теоретические сведения, относящиеся к теме курсовой работы и описать работу алгоритмов сортировок и подкрепить их схемами.
2. Сравнить язык программирования *Python* с другими языками программирования и выявить его преимущества.
3. Разработать программу для реализации алгоритмов сортировок и провести тестирование работоспособности первой версии программы.
4. Провести исследование алгоритмов сортировок на зависимость количества операций от длины массива, соответствия полученным результатам теоретической сложности алгоритма и устойчивость алгоритмов.

Глава 1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Общие сведения

Алгоритм - это последовательность инструкций, представляющая собой решение задачи или выполнение определенной операции.

Структуры данных - организованные формы хранения данных, которые предоставляют эффективный доступ и манипуляцию с ними.

Сложность алгоритма - это оценка количества ресурсов (времени, памяти) необходимых для выполнения алгоритма в зависимости от размера входных данных.

Поиск - операция нахождения нужного элемента в наборе данных.

Сортировка – операция упорядочивания элементов данных по определенному критерию.

Стек – структура данных, реализующая принцип "последним вошел, первым вышел" (*Last In First Out, LIFO*).

Очередь – структура данных, реализующая принцип "первым вошел, первым вышел" (*First In First Out, FIFO*).

Дерево – структура данных, состоящая из узлов, соединенных ребрами, причем каждый узел имеет ровно одного предка (кроме корневого) и может иметь несколько потомков.

1.2 Теоретические сведения об алгоритмах

Сортировка перемешиванием (также известная как шейкерная сортировка или коктейльная сортировка) - это улучшенный алгоритм сортировки пузырьком. Он работает путем прохода по массиву поочередно справа налево и слева направо, меняя местами соседние элементы, если они стоят в неправильном порядке. Этот процесс повторяется до тех пор, пока массив не будет отсортирован [1].

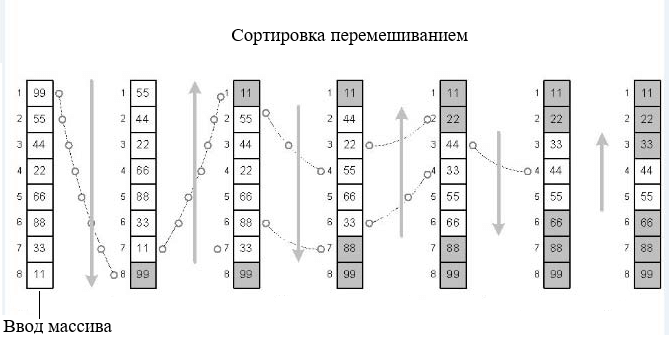


Рисунок 1 – Схема работы сортировки перемешиванием

Сортировка Шелла - это алгоритм сортировки, который является улучшенной версией сортировки вставками. Он работает путем сравнения элементов, находящихся на определенном расстоянии друг от друга, и перемещения их на правильное место. Далее расстояние между элементами уменьшается, и процесс повторяется, пока массив не будет отсортирован [1].

Сортировка выбором - это алгоритм сортировки, который работает путем нахождения минимального (или максимального) элемента в массиве и перемещением его на соответствующее место. Затем этот элемент исключается из дальнейших рассмотрений, и процесс повторяется до тех пор, пока весь массив не будет отсортирован [1].

Сортировка интросорт (*Introsort*) – это алгоритм сортировки, который сочетает в себе быструю сортировку и пирамидальную сортировку, чтобы обеспечить высокую производительность [1].

Ниже представлены схемы работы этих алгоритмов сортировок:

Ниже представлена схема работы сортировки Шелла:

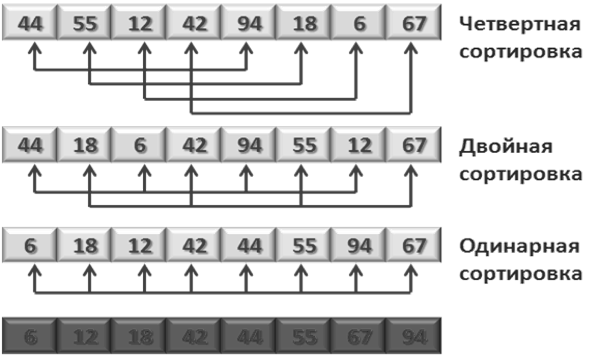


Рисунок 2 – Схема работы сортировки Шелла

Далее представлена схема работы сортировки выбором:

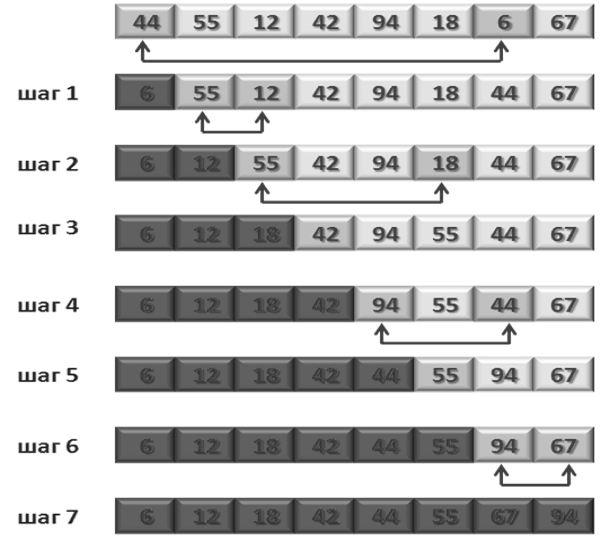


Рисунок 3 – Схема сортировки выбором

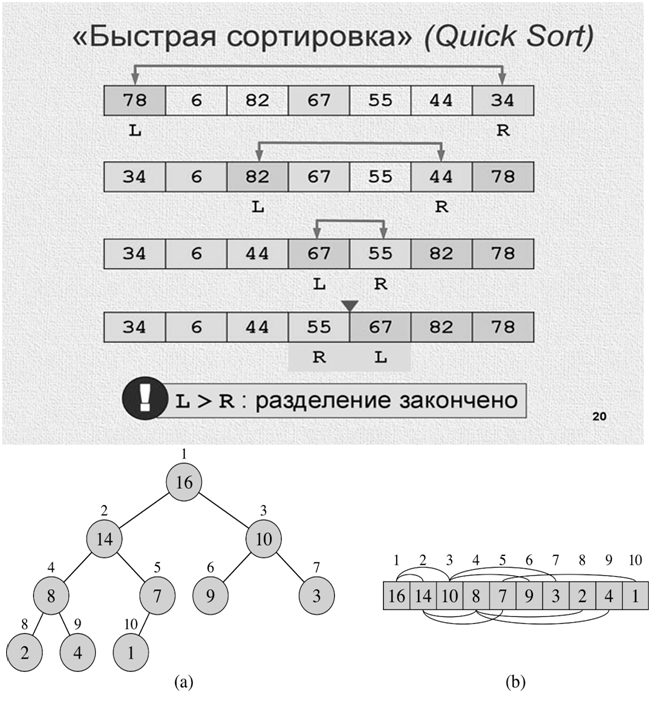
Так как, сортировка интросорт включает в себя быструю, пирамидальную сортировки, приведём схемы этих двух сортировок: 

Рисунок 4 – Схема работы быстрой и пирамидальной сортировок

1.3 Характеристики алгоритмов

1. Сортировка перемешиванием (*Cocktail sort*):

Характеристики:

* Алгоритм сортировки перемешиванием основан на разделении массива на две части, каждая из которых сортируется отдельно, после чего происходит их слияние.
* Использует метод сортировки слиянием для упорядочивания элементов массива.
* Один из простых и эффективных методов сортировки.
* Свойства:
* Временная сложность сортировки перемешиванием в лучшем, среднем и худшем случаях составляет *O(n•log(n)).*
* Применяется на практике для сортировки больших массивов, где эффективность алгоритма становится заметной.

1. Сортировка Шелла (*Shell sort*):

Характеристики:

* Гибридный метод сортировки, который объединяет простые методы сортировки в более эффективный алгоритм.
* Использует последовательность шагов для сортировки элементов, которая уменьшается с каждым проходом.

В основе алгоритма лежит метод сортировки вставками.

Свойства:

* Временная сложность алгоритма зависит от выбора последовательности шагов, но в среднем составляет *O(n•log(n)).*
* Хорошо подходит для сортировки средних и больших массивов данных.
* Имеет более высокую эффективность по сравнению с простыми сортировками, такими как сортировка пузырьком или вставками.

1. Сортировка выбором (*Selection sort*):

Характеристики:

* Простой метод сортировки, который на каждом шаге выбирает минимальный элемент и ставит его на первое место.
* Алгоритм эффективен для небольших массивов данных.
* Не требует дополнительной памяти для сортировки.

Свойства:

* Временная сложность сортировки выбором в любом случае составляет *O(n2).*
* Не является оптимальным выбором для сортировки больших массивов данных из-за его квадратичной сложности.
* Может быть полезен в случаях, когда необходимо отсортировать небольшой массив элементов.

1. Сортировка интросорт (*Introsort*):

Характеристики:

* Комбинированный метод сортировки, который объединяет сортировку быструю и сортировку пирамидальную.
* Если глубина быстрой сортировки становится слишком большой, алгоритм переключается на сортировку пирамидальную для предотвращения худшего случая.
* Обеспечивает быструю сортировку для больших массивов данных.

Свойства:

* Временная сложность в среднем случае составляет *O(n•log(n))*, что делает этот алгоритм эффективным для сортировки больших объемов данных.
* Имеет меньшую вероятность приведения к худшему случаю по сравнению с чистой быстрой сортировкой.
* Широко применяется в практике из-за своей эффективности и надежности.

1.4 Выводы по главе

В данной главе были приведены теоретические сведения, относящиеся к главе. Также была подробна описана работа таких алгоритмов сортировок как: сортировка перемешиванием, сортировка Шелла, сортировка выбором, сортировка интросорт. Каждый алгоритм сортировки был подкреплён схемой его работы. Были описаны характеристики и свойства данных алгоритмов.

Глава 2 Реализация алгоритмов и разработка управляющей программы

2.1 Общие сведения

Язык программирования *Python* разработал голландец Гвидо ван Россум. *Python* - интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической семантикой. Встроенные высокоуровневые структуры данных в сочетании с динамической типизацией и связыванием делают язык привлекательным для быстрой разработки приложений (*RAD. Rapid Application Development*). Кроме того, его можно использовать в качестве сценарного языка для связи программных компонентов [2].

Интерпретатор – это программа, которая воспринимает входную программу на исходном языке, переводит каждый оператор или строку в машинный язык и выполняет их [2].

Операция - выполнение каких-либо действий над данными (операндами). Для выполнения конкретных действий требуются специальные инструменты – операторы [2].

**Оператор в *Python*** – это специальная конструкция для операций над данными и управления логикой программы [2].

В данной главе мы рассмотрим преимущества *Python* в сравнении с другими языками программирования. Также разработаем программу для работы алгоритмов сортировок и проведём её тестирование.

2.2 Выбор языка программирования

Был выбран язык программирования *Python*.

Причины:

1. На протяжении 1-го курса нами изучался язык программирования *Python*.
2. Понятность кода. Синтаксическая особенность *Python* – выделение блоков кода отступами, что значительно упрощает зрительное восприятие программ, написанных на этом языке.
3. Интерпретируемость. Программы, написанные на языке программирования *Python*, не переводятся в машинный код, а сразу выполняются программой-интерпретатором. Это позволяет запускать код на любой платформе с установленным заранее интерпретатором.
4. Открытый код. У *Python* открытый и бесплатный исходный код [3].
5. Большое и активное сообщество. Язык программирования *Python* используют на протяжении десятилетий, за это время язык Python приобрёл огромное количество пользователей [3].
6. Богатые библиотеки и экосистема. В библиотеках уже есть огромный массив заранее написанного кода [3].
7. Встраиваемость. *Python* можно встраивать в различные приложения.
8. Быстрая разработка. Чтобы написать программу, нужно меньше кода, чем для других языков программирования, например, как *Java* и *C/C++.*

Сравним Python с другими языками программирования:

*Python* в сравнении с *Java*. *Python* — это интерпретируемый язык с динамической типизацией. *Java* же — компилируемый язык язык со статической типизацией. Эти различия делают *Python* и *Java* полными противоположностями друг друга в плане скорости запуска и выполнения программ. Код, написанный на *Python*, быстрее запускается и дольше выполняется. В то время как программы на *Java* медленнее запускаются, но гораздо быстрее выполняются.

С помощью *Java* можно разрабатывать кроссплатформенные приложения, однако и *Python* также совместим со многими операционными системами. Также с помощью этих языков программирования разработчики могут создавать сетевые приложения. Если же говорить о сложности этих двух языков, то *Java* безусловно уступает *Python* в простоте изучения [4].

*Python* в сравнении *C/C++*. Все преимущества и недостатки языка *Java*, описанные выше, можно отнести и к *C/C++*. Однако всё же есть существенное отличие *C/C++* от *Python* – это стоимость разработки и качество программного обеспечения. Для того чтобы разрабатывать на *C/C++*, программист должен обладать более высокой квалификацией. Из этого можно сделать вывод, что программное обеспечение, разработанное на *C/C++*, будет более качественным.

Ещё одно отличие *Python* от *C/C++* – длина кода и простота его восприятия. Программы, написанные на языке *Python*, могут быть до 10 раз короче, чем на *C/C++*. Конечно же, это напрямую влияет на скорость разработки [4].

2.3 Реализация функций алгоритмов

Для выполнения поставленной задачи мною был написана программа для реализации алгоритмов сортировок (см. Приложение Б – код программы).

Описание работы программы:

1. Сортировка перемешиванием (*Cocktail Sort*)

Описание: Сортировка перемешиванием — это разновидность пузырьковой сортировки, которая работает в двух направлениях на каждом проходе через список. Этот метод помогает устранить проблемы пузырьковой сортировки, когда небольшие элементы "всплывают" слишком медленно, особенно если они находятся в конце списка [5].

Алгоритм:

* Инициализируется переменная *swapped* как *True* для начала работы цикла.
* Определяются начальные и конечные индексы массива: *start = 0* и *end = n - 1.*
* Вход в цикл *while*, который будет работать до тех пор, пока есть обмены (*swapped* равно *True*).
* Внутренний цикл проходит от *start* до *end*, сравнивая соседние элементы и меняя их местами, если они идут в неправильном порядке. Если произошел обмен, *swapped* устанавливается в *True*.
* Если в предыдущем проходе не было обменов (*swapped* остается *False*), сортировка завершена.
* Уменьшается значение *end* на 1, так как последний элемент уже на своем месте.
* Внутренний цикл проходит от *end* - 1 до *start*, выполняя обратный проход, сравнивая и меняя местами элементы в противоположном направлении.
* Увеличивается значение *start* на 1, так как первый элемент уже на своем месте.
* Повторяется процесс, пока *swapped* не станет *False*.

Код программы представлен в листинге 1:

Листинг 1

def cocktail\_sort(arr):

n = len(arr)

swapped = True

start = 0

end = n - 1

while swapped:

swapped = False

# По возрастанию

for i in range(start, end):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

swapped = True

if not swapped:

break

swapped = False

end -= 1

# По убыванию

for i in range(end - 1, start - 1, -1):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

swapped = True

start += 1

return arr

2. Сортировка Шелла (*Shell Sort*)

Описание: Сортировка Шелла – это улучшение сортировки вставками. Идея состоит в сравнении элементов, находящихся на определенном расстоянии (*gap*) друг от друга, а не только соседних элементов. Это позволяет значительно уменьшить количество перестановок для дальних элементов [5].

Алгоритм:

* Инициализируется *gap* (интервал между элементами, которые будут сравниваться) как половина длины массива (n // 2).
* Цикл *while* работает до тех пор, пока *gap* больше нуля.
* Для каждого элемента массива, начиная с индекса gap, выполняется:
* Сохранение текущего элемента *arr[i]* во временной переменной *temp*.
* Внутренний цикл перемещает элементы, которые находятся на расстоянии *gap*, пока не найдется правильное место для *temp*.
* Уменьшение *gap* вдвое на каждой итерации.

Листинг 2 – Код сортировки Шелла

def shell\_sort(arr):

n = len(arr)

gap = n // 2

while gap > 0:

for i in range(gap, n):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

gap //= 2

return arr

3. Сортировка выбором (Selection Sort)

**Описание:** Сортировка выбором последовательно находит минимальный элемент из неотсортированной части массива и перемещает его в начало. Процесс повторяется для подмассива, начиная с следующей позиции [5].

**Алгоритм:**

* + Цикл проходит по всему массиву, начиная с первого элемента.
  + Инициализируется *min\_idx* как текущий индекс.
  + Внутренний цикл находит минимальный элемент в оставшейся части массива.
  + После нахождения минимального элемента, он меняется местами с элементом, находящимся на позиции i.
  + Процесс повторяется для следующего элемента.

**Листинг 3 – Код сортировки выбором**

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_idx = i

for j in range(i + 1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

4. Сортировка интросорт (*Introsort*)

**Описание:** *Introsort* сочетает быструю сортировку и пирамидальную сортировку. Начинается с быстрой сортировки, а при достижении определенной глубины рекурсии переключается на пирамидальную сортировку. Это предотвращает деградацию производительности быстрой сортировки в худших случаях [5].

**Алгоритм:**

Вспомогательные функции

* 1. ***Heapify*:** Функция для обеспечения свойства кучи, необходимого для пирамидальной сортировки:

Листинг 4 – Вспомогательная программа для пирамидальной сортировки

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[largest] < arr[l]:

largest = l

if r < n and arr[largest] < arr[r]:

largest = r

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

* 1. ***Heapsort*:** Функция для выполнения пирамидальной сортировки:

*Python*

Листинг 5 – Программа сортировки

def heapsort(arr):

n = len(arr)

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

* 1. ***Partition*:** Функция для выполнения разбиения в быстрой сортировке:

*Python*

Листинг 6 – Программа быстрой сортировки

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

* 1. ***Introsort*:** Основная функция интросорта, которая контролирует глубину рекурсии и переключается на пирамидальную сортировку при необходимости:

Листинг 7 – Релизация сортировки интросорт

def introsort(arr, low, high, depth\_limit):

while high > low:

if depth\_limit == 0:

heapsort(arr[low:high + 1])

return

pivot = partition(arr, low, high)

introsort(arr, pivot + 1, high, depth\_limit - 1)

high = pivot - 1

* 1. *Introsort\_sort*: Инициализирует интросорт с расчетом предельной глубины рекурсии:

def introsort\_sort(arr):

depth\_limit = 2 \* (len(arr)).bit\_length()

introsort(arr, 0, len(arr) - 1, depth\_limit)

return arr

Выполнение сортировок

Код выполняет каждую сортировку на копии исходного массива и выводит результат:

Листинг 9 – Создание копий для каждой сортировки и вывод результата

cocktail\_sorted\_arr = cocktail\_sort(arr.copy())

print("Отсортированный массив (Перемешиванием):", cocktail\_sorted\_arr, "\n")

shell\_sorted\_arr = shell\_sort(arr.copy())

print("Отсортированный массив (Шеллом):", shell\_sorted\_arr, "\n")

selection\_sorted\_arr = selection\_sort(arr.copy())

print("Отсортированный массив (Выбором):", selection\_sorted\_arr, "\n")

introsort\_sorted\_arr = introsort\_sort(arr.copy())

print("Отсортированный массив (Интросортом):", introsort\_sorted\_arr, "\n")

Каждая функция сортирует копию исходного массива *arr*, что позволяет избежать изменений в исходном массиве между сортировками. Это важно для корректного сравнения результатов разных алгоритмов сортировки.

2.4 Тестирование работоспособности и корректности разработанных алгоритмов

При тестировании работоспособности и корректности разработанных алгоритмов для удобства и быстроты использовался фреймворк *pytest.*

*pytest* – это мощный и удобный в использовании фреймворк для тестирования кода на языке программирования *Python*. Он поддерживает автоматизацию тестов, простую и масштабируемую организацию тестов и множество других полезных функций для разработки и тестирования.

Основные особенности *pytest*:

1. Простота использования:

* *pytest* позволяет писать тесты быстро и просто, без избыточного кода.
* Синтаксис тестов очень похож на обычный *Python*-код, что делает его легким для освоения.

1. Автоматическое обнаружение тестов:

* *pytest* автоматически обнаруживает все тестовые функции, классы и методы, которые начинаются с префикса *test*\_ или имеют его в названии.
* Не требуется специальных аннотаций для тестов, что упрощает их написание.

1. Мощные возможности отчетности:

* *pytest* предоставляет подробные отчеты о тестах, включая информацию о неудачных тестах, стек-трейсы и многое другое.
* Поддерживает различные плагины для расширения возможностей отчетности.

1. Плагины и расширяемость:

* *pytest* имеет развитую экосистему плагинов, которые добавляют множество полезных функций (например, параллельное выполнение тестов, интеграция с *CI/CD*, и т.д.).
* Можно легко писать свои собственные плагины для специфичных нужд.

1. Совместимость с другими тестовыми фреймворками:

* *pytest* может запускать тесты, написанные для *unittest* и *nose*, что делает переход на *pytest* менее болезненным.

Была разработана программа для тестирования работоспособности и корректности разработанных алгоритмов с применением *pytest*.

Листинг 10 – Программа тестирования с использованием фреймворка *pytest*

import pytest

from code\_cocktail\_sort import cocktail\_sort

from code\_shell\_sort import shell\_sort

from code\_selection\_sort import selection\_sort

from code\_introsort import introsort\_sort

testing\_arr = [

([1, 0, 3, -9, 7, -5, 2, -4, -1, 8], [-9.0, -5.0, -4.0, -1.0, 0.0, 1.0, 2.0, 3.0, 7.0, 8.0]),

([2.3, 0.5, -5.5, -7.1, 2.8, -4.1, 1.8, 1.1, -9.9, -2.1],[-9.9, -7.1, -5.5, -4.1, -2.1, 0.5, 1.1, 1.8, 2.3, 2.8]),

([998.998, 0.12345, -111.1111, 535.35353, 2.145, -7.4949, -0.5421, 651.345, -87.001, -106.2325],[-111.1111, -106.2325, -87.001, -7.4949, -0.5421, 0.12345, 2.145, 535.35353, 651.345, 998.998]),

([],[]),

([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0, 5.0, 5.0, 6.0, 6.0, 7.0, 7.0, 8.0, 8.0, 9.0, 9.0])]

@pytest.mark.parametrize("sort\_func", [cocktail\_sort, shell\_sort, selection\_sort, introsort\_sort])

@pytest.mark.parametrize("input\_list, expected", testing\_arr)

def test\_sorts(sort\_func, input\_list, expected):

    assert sort\_func(input\_list) == expected

print(" ")

if \_\_name\_\_ == "\_\_main\_\_":

    pytest.main()

Тестирование алгоритмов сортировки будет проводится следующим образом:

* 1. Проверка на работу с положительными и отрицательными значениями элементов
  2. Проверка на работу с нецелочисленными значениями элементов
  3. Проверка на работу с числами, в которых не менее 2 знаков в целочисленной и нецелочисленной частях.
  4. Проверка на работу с пустым массивом
  5. Проверка на работу с повторяющимися элементами

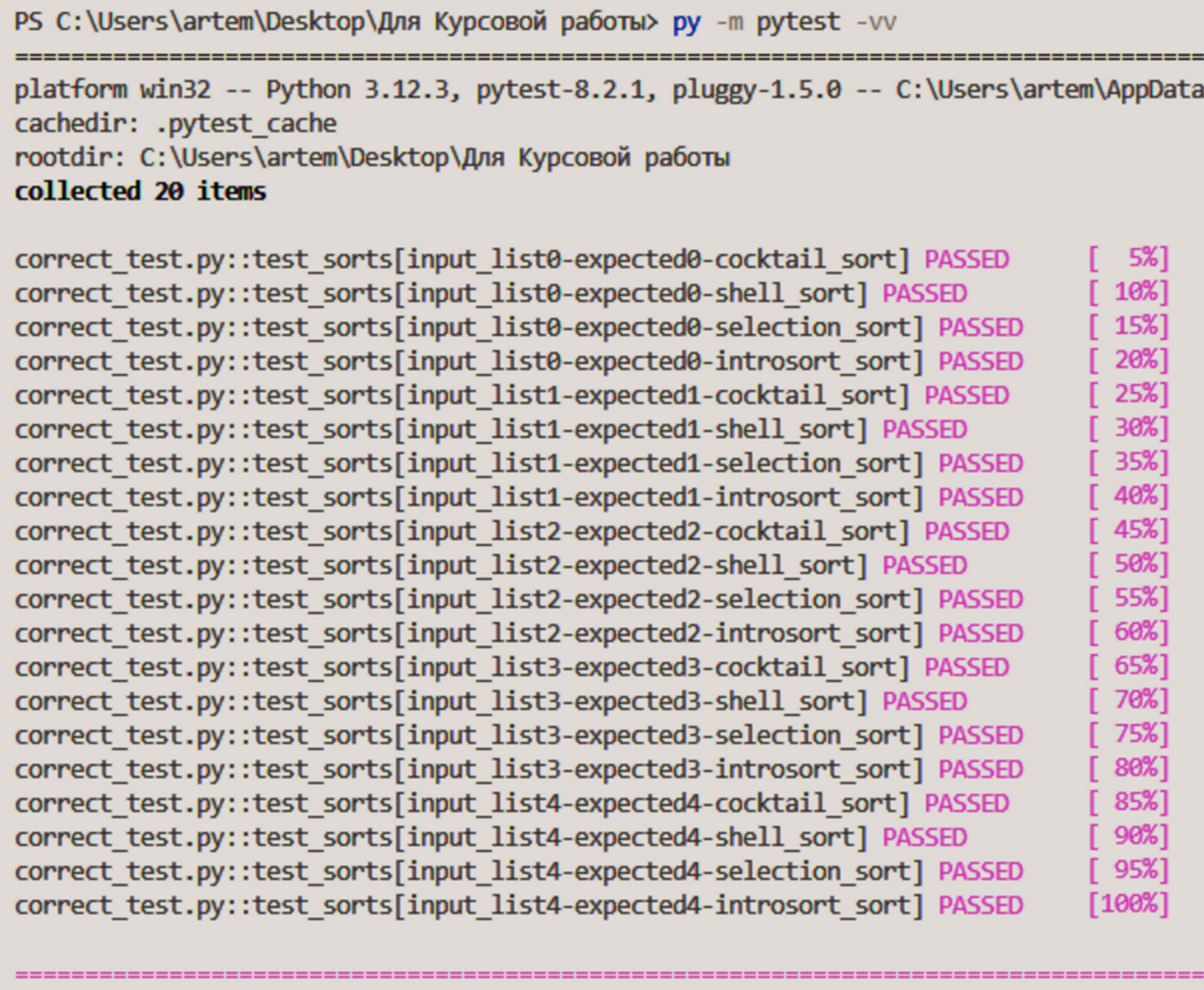


Рисунок 5 – Результат работы программы тестирования

2.5 Разработка программы исследования алгоритмов

Был разработан код для исследования алгоритмов на зависимости количества операций от длины массива, соответствия полученным результатам теоретической сложности алгоритма, устойчивости алгоритма (см. Приложение Б).

В данной программе я использовал такие библиотеки как:

* 1. *random* используется для генерации случайных чисел и выполнения различных операций, связанных со случайностью;
  2. *time* предоставляет различные функции для работы с временем и измерения времени;
  3. math предоставляет доступ к различным математическим функциям и константам, что позволяет выполнять сложные математические операции и вычисления.
  4. Проверка устойчивости сортировки

Цель: Проверить, сохраняет ли алгоритм сортировки порядок равных элементов в исходном массиве.

Реализация показана в листинге 11:  
Листинг 11

def check\_stability(sort\_fn):

class StableElement:

def \_\_init\_\_(self, value, original\_index):

self.value = value

self.original\_index = original\_index

def \_\_lt\_\_(self, other):

return self.value < other.value

data = [StableElement(value, idx) for idx, value in enumerate([5, 3, 8, 3, 5])]

sorted\_data, \_ = sort\_fn(data.copy())

stable = True

for i in range(len(sorted\_data) - 1):

if sorted\_data[i].value == sorted\_data[i + 1].value and sorted\_data[i].original\_index > sorted\_data[i + 1].original\_index:

stable = False

break

return stable

Класс *StableElement*:

Содержит значение *value* и индекс исходного массива *original\_index*.

Реализует метод \_\_*lt*\_\_, чтобы сравнивать только значения, игнорируя индексы.

Создание тестовых данных:

Массив *data* содержит элементы *StableElement*, где значение повторяется (например, два элемента со значением 3 и два элемента со значением 5).

Сортировка и проверка:

*sort\_fn* – функция сортировки, применяется к копии массива *data*.

Проверяется, находятся ли равные значения в правильном порядке по оригинальным индексам после сортировки.

Если нет, сортировка не является стабильной.

* 1. Измерение времени и количества операций

Цель: Измерить производительность сортировки, в частности, время выполнения и количество операций сравнения/перестановки.

Реализация показана в листинге 12:

Листинг 12

def measure\_sort(sort\_fn, arr):

start\_time = time.time()

\_, operations = sort\_fn(arr.copy())

end\_time = time.time()

elapsed\_time = end\_time - start\_time

return operations, elapsed\_time

Начало измерения времени:

Функция *time.time*() фиксирует текущее время перед началом сортировки (*start\_time*).

Сортировка массива:

Вызывается функция сортировки *sort\_fn* на копии массива *arr*, чтобы не изменять исходный массив.

Возвращаются отсортированный массив и количество операций *operations*.

Конец измерения времени:

Фиксируется время после завершения сортировки (*end\_time*).

Вычисление затраченного времени:

Разница между *end\_time* и *start\_time* дает *elapsed\_time* – время, затраченное на сортировку.

Возврат результатов:

Функция возвращает количество операций и затраченное время.

* 1. Анализ алгоритмов сортировки

Цель: Сравнить производительность различных алгоритмов сортировки на массивах разной длины и проверить их устойчивость.

Реализация показана в листинге 13:

Листинг 13

def analyze\_sorting\_algorithms():

lengths = [10, 50, 100, 500, 1000, 5000, 10000]

algorithms = [

("Сортировка перемешиванием", cocktail\_sort, "O(n^2)"),

("Сортировка Шелла", shell\_sort, "O(n^1.5)"),

("Сортировка выбором", selection\_sort, "O(n^2)"),

("Сортировка Интросорт", introsort\_sort, "O(n log n)")

]

for name, sort\_fn, complexity in algorithms:

print(f"\n{name} ({complexity}):")

for length in lengths:

arr = [random.random() for \_ in range(length)]

operations, elapsed\_time = measure\_sort(sort\_fn, arr)

print(f"Длина массива: {length}, Кол-во операций: {operations}, Время: {elapsed\_time:.6f} секунд")

is\_stable = check\_stability(sort\_fn)

print(f"Стабильность: {'Стабильна' если is\_stable else 'Не стабильна'}")

Определение длин массивов:

*lengths* — список длин массивов для тестирования: от 10 до 10 000 элементов.

Определение алгоритмов сортировки:

*algorithms* — список кортежей, где каждый содержит название алгоритма, функцию сортировки и асимптотическую сложность.

Проход по каждому алгоритму:

Для каждого алгоритма выводится его название и сложность.

Проход по каждому размеру массива:

Для каждого размера массива создается новый массив случайных чисел.

Измеряется количество операций и время сортировки с помощью *measure\_sort*.

Выводятся результаты измерений: длина массива, количество операций и время выполнения.

Проверка устойчивости:

Проверяется, является ли алгоритм стабильным с помощью *check\_stability*.

Выводится результат проверки устойчивости.

2.6 Выводы по главе

В данной главе были описаны причины выбора языка программирования *Python*. Были описаны алгоритмы сортировок и написан код для их реализации на языке *Python*, а также был разработан код для исследования алгоритмов сортировок. В результате тестирования программы было выявлено, что она может работать с любыми числами. А также был разработана конечная программа.

Глава 3 ИССЛЕДОВАНИЕ АЛГОРИТМОВ

3.1 Общие сведения

В данной главе будут проведены исследования алгоритмов сортировки на зависимости количества операций от длины массива, соответствия полученным результатам теоретической сложности алгоритма и устойчивость алгоритмов.

Будут представлены графики зависимостей количества операций от длины массива и результаты работы программы для каждого алгоритма сортировки.

3.2 Исследование зависимости количества операций от длины массива

1. Исследование зависимости количества операций от длины массива на основе алгоритма сортировки перемешиванием:

Ниже представлены результаты исследования зависимости количества операций от длины массива в виде графика для сортировки перемешиванием:

Рисунок 6 –График зависимости количества операций от длины массива

Результат работы программы:

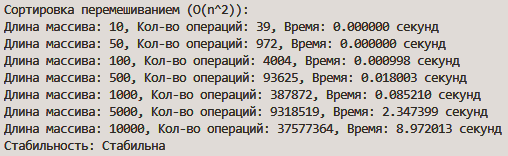


Рисунок 7 – Результат исследования сортировки перемешивания

1. Исследование зависимости количества операций от длины массива на основе алгоритма сортировки Шелла:

Ниже представлены результаты исследования зависимости количества операций от длины массива в виде графика для сортировки Шелла:

Рисунок 8 – График зависимости количества операций от длины массива

Результат работы программы:

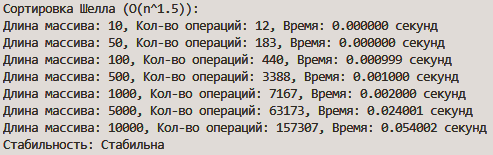


Рисунок 9 – Результат исследования сортировки Шелла

1. Исследование зависимости количества операций от длины массива на основе алгоритма сортировки выбором:

Ниже представлены результаты исследования зависимости количества операций от длины массива в виде графика для сортировки выбором:

Рисунок 10 – График зависимости количества операций от длины массива

Результат работы программы:

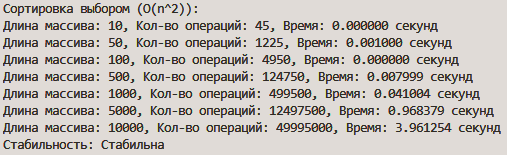


Рисунок 11 – Результат исследования сортировки выбором

1. Исследование зависимости количества операций от длины массива на основе алгоритма сортировки интросорт:

Ниже представлены результаты исследования зависимости количества операций от длины массива для сортировки Интросорт:

Рисунок 12 – График зависимости количества операций от длины массива

Результат работы программы:

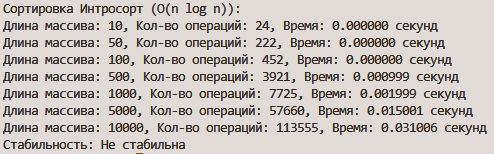


Рисунок 13 – Результат исследования сортировки Интросорт

Благодаря графикам количества операций и длины массива и их зависимости мы можем увидеть, что количество операций растёт в геометрической прогрессии, по сравнению с длиной массива, хоть и в сортировках Шелла и интросорт это менее сильно выражено.

3.3 Проверка соответствия полученным результатам теоретической сложности алгоритма

* 1. Сортировка перемешиванием

В худшем случае количество операций (сравнений и обменов) для шейкерной сортировки будет примерно таким же, как и для пузырьковой сортировки, то есть *O(n2)*, где *n* – длина массива.

Для пузырьковой сортировки в худшем случае количество операций можно оценить, как: *n(n-1)/2*. Для шейкерной сортировки, это число также можно принять за оценку, поскольку она в худшем случае тоже выполняет большое количество сравнений и обменов. Таким образом, можно считать, что количество операций в худшем случае составляет *n(n−1)/2.*

* 1. Сортировка Шелла

Сортировка Шелла с шагом 0,5 подразумевает использование последовательности приращений *hk=[n/2k]*. В этом случае сложность в худшем случае можно оценить, как: *O(n3/2)*, где *n* – длина массива.

Для оценки количества операций в худшем случае можно воспользоваться формулой: *T(n)=k⋅n3/2*, где *k* – некоторый коэффициент, который зависит от деталей реализации. В данном случае для оценки мы примем *k=1*.

* 1. Сортировка выбором

Сортировка выбором имеет временную сложность *O(n2)* в худшем случае, так как она выполняет *n(n−1)/2* сравнений и *n−1* обменов (хотя число обменов не влияет существенно на порядок сложности).

Количество операций в худшем случае можно оценить по формуле: *T(n)=n(n−1)/2*

* 1. Сортировка Интросорт

Интроспективная сортировка (*introsort*) сочетает в себе быструю сортировку (*quicksort*), сортировку кучей (*heapsort*) и сортировку вставками (*insertion* *sort*). В худшем случае, когда быстрая сортировка деградирует, интросорт переключается на сортировку кучей, что обеспечивает временную сложность *O(n log (n))*.

Результаты вычисления и дальнейшего сравнения теоретической и практической сложностей представлены в таблице 1:

Таблица 1 – Соответствие теоретической и практической сложностей алгоритмов сортировки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Длина массива | Теоретическая сложность | Кол-во операций | Соответствие теоретич. и практич. сложностей |
| Сортировка  перемешиванием *O(n2)* | 10 | 45 | 39 | - |
| 50 | 1225 | 972 | - |
| 100 | 4950 | 4004 | - |
| 500 | 124750 | 93625 | - |
| 1000 | 499500 | 387872 | - |
| 5000 | 12497500 | 9318519 | - |
| 10000 | 49995000 | 37577364 | - |
| Сортировка  Шелла *O(n3/2)* | 10 | 32 | 12 | - |
| 50 | 354 | 183 | - |
| 100 | 1000 | 440 | - |
| 500 | 5590 | 3388 | - |
| 1000 | 31623 | 7167 | - |
| 5000 | 353553 | 63173 | - |
| 10000 | 1000000 | 157307 | - |
| Сортировка вставками *O(n2)* | 10 | 45 | 45 | = |
| 50 | 1225 | 1225 | = |
| 100 | 4950 | 4950 | = |
| 500 | 124750 | 124750 | = |
| 1000 | 499500 | 499500 | = |
| 5000 | 12497500 | 12497500 | = |
| 10000 | 49995000 | 49995000 | = |
| Интроспективная сортировка *O(n\*log(n))* | 10 | 33 | 24 | - |
| 50 | 282 | 222 | - |
| 100 | 664 | 452 | - |
| 500 | 4485 | 3921 | - |
| 1000 | 9970 | 7725 | - |
| 5000 | 61450 | 57660 | - |
| 10000 | 132900 | 113555 | - |

Условные обозначения в таблице 1: “+” – превосходит теоретич. сложность, “-” – не превосходит теоретич. сложность, “=” – равно теоретич. сложности.

3.4 Исследование устойчивости алгоритма

Устойчивость алгоритма сортировки означает, что равные элементы сохраняют свой относительный порядок после сортировки. В программе для проверки устойчивости используется специальная функция *check*\_*stability*.

Вот как происходит проверка устойчивости:

**Создание тестовых данных**:

Создается список объектов *StableElement*, каждый из которых имеет значение и индекс.

**Сортировка данных**:

Список объектов передается в функцию сортировки, которая сортирует его по значениям.

**Проверка порядка**:

После сортировки проверяется, что для всех пар равных элементов их исходный порядок (по индексу) сохранился.

Если находится пара равных элементов, где порядок изменился, алгоритм признается нестабильным.

Код функции проверки устойчивости показан в листинге 14:

Листинг 14

def check\_stability(sort\_fn):

class StableElement:

def \_\_init\_\_(self, value, original\_index):

self.value = value

self.original\_index = original\_index

def \_\_lt\_\_(self, other):

return self.value < other.value

data = [StableElement(value, idx) for idx, value in enumerate([5, 3, 8, 3, 5])]

sorted\_data, \_ = sort\_fn(data.copy())

stable = True

for i in range(len(sorted\_data) - 1):

if sorted\_data[i].value == sorted\_data[i + 1].value and sorted\_data[i].original\_index > sorted\_data[i + 1].original\_index:

stable = False

break

return stable

3. Измерение производительности

Для каждого алгоритма измеряется количество операций и время выполнения сортировки на массивах разной длины. Используются массивы длиной от 10 до 10,000 элементов, заполненные случайными значениями.

Код для измерения производительности преставлен в листинге 15:

Листинг 15

def measure\_sort(sort\_fn, arr):

start\_time = time.time()

\_, operations = sort\_fn(arr.copy())

end\_time = time.time()

elapsed\_time = end\_time - start\_time

return operations, elapsed\_time

4. Анализ результатов

Результаты измерений выводятся на экран для каждого алгоритма. Показаны количество операций и время выполнения для каждого размера массива. Также выводится информация о том, является ли алгоритм устойчивым.

3.5 Выводы по главе

В данной главе были проведены исследования алгоритмов сортировки на зависимости количества операций от длины массива, соответствия полученным результатам теоретической сложности алгоритма и устойчивость алгоритмов.

Согласно полученным из работы программы данным, можно сделать вывод о том, что она работает корректно, так как значения, нужные в ходе исследования на практике не превосходят их теоретических значений.

Заключение

В данной курсовой работе была разработана программа, целью которой было реализация таких алгоритмов сортировки, как: сортировка перемешиванием, сортировка Шелла, сортировка выбором, сортировка интросорт. Также были проведены исследования этих алгоритмов сортировок на зависимость количества операций от длины массива, на их устойчивость, и были проведена проверка полученных данных теоретической сложности этих алгоритмов. Для этого было проведено исследование некоторых языков программирования на то, какой их них будет лучше всего подходить под поставленные задачи, таким языком оказался *Python*. Также были рассмотрены алгоритмы сортировки, чтобы стало возможным реализовать их на данном языке программирования.

Библиографический список

* 1. Каширская, Е. Н. Программирование алгоритмов сортировки: учебное пособие / Е. Н. Каширская, М. А. Макаров, С. Е. Харьковский. — Москва: РТУ МИРЭА, 2021. — 102 с. — Текст: электронный // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/218591 (дата обращения: 14.05.2024).
  2. Никитина, Т. П. Программирование. Основы Python / Т. П. Никитина, Л. В. Королев. — Санкт-Петербург: Лань, 2023. — 156 с. — ISBN 978-5-507-45283-5. — Текст: электронный // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/302714 (дата обращения: 15.05.2024).
  3. Грачев В.М., Марченко С.С. Перспективы использования языка «Python» в решении прикладных задач [Электронный ресурс] // XIV Международная студенческая научная конференция. Студенческий научный форум – 2022. URL: <https://scienceforum.ru/2022/article/2018031232> (дата обращения: 15.06.2024).
  4. Борзунов, С. В. Языки программирования. Python: решение сложных задач / С. В. Борзунов, С. Д. Кургалин. — Санкт-Петербург: Лань, 2023. — 192 с. — ISBN 978-5-507-45923-0. — Текст: электронный // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/319394 (дата обращения: 15.05.2024).
  5. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных: учебное пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова. — Москва: РТУ МИРЭА, 2022 — Часть 1: Сложность алгоритмов. Сортировки. Линейные структуры данных. Поиск в таблице — 2022. — 110 с. — ISBN 978-5-7339-1612-5. — Текст: электронный // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/256592 (дата обращения: 16.05.2024). — Режим доступа: для авториз. пользователей.
  6. Введение в программирование и структуры данных / К. Фислер, Ш. Кришнамурти, Б. С. Лернер, Д. Г. Политц; перевод с английского А. В. Снастина. — Москва: ДМК Пресс, 2022. — 440 с. — ISBN 978-5-93700-137-5. — Текст: электронный // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/314846 (дата обращения: 16.05.2024)

Приложение А - Схема программы

Изм.

Лист

№ докум.

Подпись

Дата

Лист

40

ПензГТУ 3.09.03.02.57ПЗ

Разраб.

Сидоров А.А.

Провер.

Пискаев К.Ю.

Т. контр

Н.контр.

Утвержд.

.

аАв

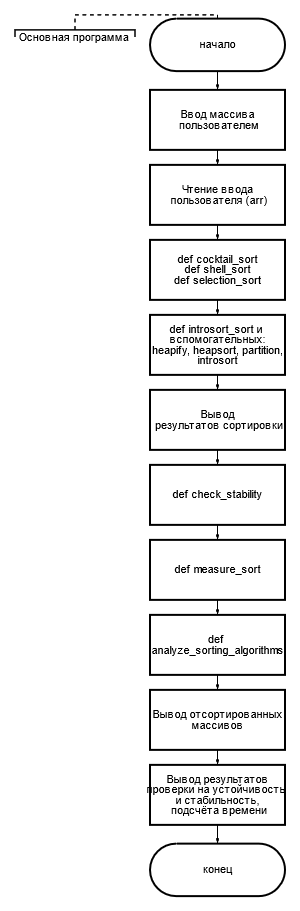
Блок-Схема

Лит.

Листов

47

гр. 23ИС1б



Приложение Б – Код программы сортировок

import random

# Вводим массив с клавиатуры

arr = list(map(float, input('Введите элементы списка через пробел: ').split()))

# Сортировка перемешиванием (Cocktail Sort)

def cocktail\_sort(arr):

n = len(arr)

swapped = True

start = 0

end = n - 1

while swapped:

swapped = False

# По возрастанию

for i in range(start, end):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

swapped = True

if not swapped:

break

swapped = False

end -= 1

# По убыванию

for i in range(end - 1, start - 1, -1):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

swapped = True

start += 1

return arr

# Сортировка Шелла (Shell Sort)

def shell\_sort(arr):

n = len(arr)

gap = n // 2

while gap > 0:

for i in range(gap, n):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

gap //= 2

return arr

# Сортировка выбором (Selection Sort)

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_idx = i

for j in range(i+1, n):

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

# Сортировка интросорт (Introsort)

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[largest] < arr[l]:

largest = l

if r < n and arr[largest] < arr[r]:

largest = r

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

def heapsort(arr):

n = len(arr)

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

for i in range(n-1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

def introsort(arr, low, high, depth\_limit):

while high > low:

if depth\_limit == 0:

heapsort(arr[low:high + 1])

return

pivot = partition(arr, low, high)

introsort(arr, pivot + 1, high, depth\_limit - 1)

high = pivot - 1

def introsort\_sort(arr):

depth\_limit = 2 \* (len(arr)).bit\_length()

introsort(arr, 0, len(arr) - 1, depth\_limit)

return arr

# Создание копий массива для каждой сортировки

cocktail\_sorted\_arr = cocktail\_sort(arr.copy())

print("Отсортированный массив (Перемешиванием):", cocktail\_sorted\_arr, "\n"

shell\_sorted\_arr = shell\_sort(arr.copy())

print("Отсортированный массив (Шеллом):", shell\_sorted\_arr, "\n")

selection\_sorted\_arr = selection\_sort(arr.copy())

print("Отсортированный массив (Выбором):", selection\_sorted\_arr, "\n")

introsort\_sorted\_arr = introsort\_sort(arr.copy())

print("Отсортированный массив (Интросортом):", introsort\_sorted\_arr, "\n"

Приложение В - Код программы

import random

import time

import math

# Сортировка перемешиванием (Cocktail Sort)

def cocktail\_sort(arr):

    n = len(arr)

    swapped = True

    start = 0

    end = n - 1

    operations = 0

    while swapped:

        swapped = False

        # По возрастанию

        for i in range(start, end):

            operations += 1

            if arr[i] > arr[i + 1]:

                arr[i], arr[i + 1] = arr[i + 1], arr[i]

                swapped = True

        if not swapped:

            break

        swapped = False

        end -= 1

        # По убыванию

        for i in range(end - 1, start - 1, -1):

            operations += 1

            if arr[i] > arr[i + 1]:

                arr[i], arr[i + 1] = arr[i + 1], arr[i]

                swapped = True

        start += 1

    return arr, operations

# Сортировка Шелла (Shell Sort)

def shell\_sort(arr):

    n = len(arr)

    gap = n // 2

    operations = 0

    while gap > 0:

        for i in range(gap, n):

            temp = arr[i]

            j = i

            while j >= gap and arr[j - gap] > temp:

                operations += 1

                arr[j] = arr[j - gap]

                j -= gap

            arr[j] = temp

        gap //= 2

    return arr, operations

# Сортировка выбором (Selection Sort)

def selection\_sort(arr):

    n = len(arr)

    operations = 0

    for i in range(n):

        min\_idx = i

        for j in range(i+1, n):

            operations += 1

            if arr[j] < arr[min\_idx]:

                min\_idx = j

        arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

    return arr, operations

# Сортировка интросорт (Introsort)

def heapify(arr, n, i, operations):

    largest = i

    l = 2 \* i + 1

    r = 2 \* i + 2

    if l < n: operations

+= 1  # Comparison

        if arr[largest] < arr[l]:

            largest = l

    if r < n:

        operations += 1  # Comparison

        if arr[largest] < arr[r]:

            largest = r

    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        operations += 1  # Swap

        operations = heapify(arr, n, largest, operations)

    return operations

def heapsort(arr, operations):

    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):

        operations = heapify(arr, n, i, operations)

    for i in range(n - 1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]

        operations += 1  # Swap

        operations = heapify(arr, i, 0, operations)

    return operations

def partition(arr, low, high, operations):

    pivot = arr[high]

    i = low - 1

    for j in range(low, high):

        operations += 1  # Comparison

        if arr[j] < pivot:

            i += 1

            arr[i], arr[j] = arr[j], arr[i]

            operations += 1  # Swap

    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    operations += 1  # Swap

    return i + 1, operations

def introsort(arr, low, high, depth\_limit, operations):

    while high > low:

if depth\_limit == 0:

            operations = heapsort(arr[low:high + 1], operations)

            return operations

        pivot, operations = partition(arr, low, high, operations)

        operations = introsort(arr, pivot + 1, high, depth\_limit - 1, operations)

        high = pivot - 1

    return operations

def introsort\_sort(arr):

    depth\_limit = 2 \* (len(arr)).bit\_length()

    operations = introsort(arr, 0, len(arr) - 1, depth\_limit, 0)

    return arr, operations

# Проверка устойчивости сортировки

def check\_stability(sort\_fn):

    class StableElement:

        def \_\_init\_\_(self, value, original\_index):

            self.value = value

            self.original\_index = original\_index

        def \_\_lt\_\_(self, other):

            return self.value < other.value

    data = [StableElement(value, idx) for idx, value in enumerate([5, 3, 8, 3, 5])]

    sorted\_data, \_ = sort\_fn(data.copy())

    stable = True

    for i in range(len(sorted\_data) - 1):

        if sorted\_data[i].value == sorted\_data[i + 1].value and sorted\_data[i].original\_index > sorted\_data[i + 1].original\_index:

            stable = False

            break

    return stable

# Функция для измерения времени и количества операций

def measure\_sort(sort\_fn, arr):

    start\_time = time.time()

    \_, operations = sort\_fn(arr.copy())

    end\_time = time.time()

elapsed\_time = end\_time - start\_time

    return operations, elapsed\_time

# Исследование зависимости количества операций и времени выполнения от длины массива

def analyze\_sorting\_algorithms():

    lengths = [10, 50, 100, 500, 1000, 5000, 10000]

    algorithms = [

        ("Сортировка перемешиванием", cocktail\_sort, "O(n^2)"),

        ("Сортировка Шелла", shell\_sort, "O(n^1.5)"),

        ("Сортировка выбором", selection\_sort, "O(n^2)"),

        ("Сортировка Интросорт", introsort\_sort, "O(n log n)")

    ]

    for name, sort\_fn, complexity in algorithms:

        print(f"\n{name} ({complexity}):")

        for length in lengths:

            arr = [random.random() for \_ in range(length)]

            operations, elapsed\_time = measure\_sort(sort\_fn, arr)

            print(f"Длина массива: {length}, Кол-во операций: {operations}, Время: {elapsed\_time:.6f} секунд")

        # Проверка устойчивости

        is\_stable = check\_stability(sort\_fn)

        print(f"Стабильность: {'Стабильна' if is\_stable else 'Не стабильна'}")

# Ввод массива с клавиатуры и сортировка

def main():

    arr = list(map(float, input('Введите элементы списка через пробел: ').split()))

    print(" ")

    sorted\_arr, \_ = cocktail\_sort(arr.copy())

    print("Отсортированный массив (Перемешиванием):", sorted\_arr, "\n")

    sorted\_arr, \_ = shell\_sort(arr.copy())

    print("Отсортированный массив (Шеллом):", sorted\_arr, "\n")

    sorted\_arr, \_ = selection\_sort(arr.copy())

print("Отсортированный массив (Выбором):", sorted\_arr, "\n")

    sorted\_arr, \_ = introsort\_sort(arr.copy())

    print("Отсортированный массив (Интросортом):", sorted\_arr, "\n")

    # Анализ алгоритмов

    analyze\_sorting\_algorithms()

if \_\_name\_\_ == "\_\_main\_\_":

    main()