МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допущен к защите

Заведующий кафедрой ПМИ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.В. Разова

**Сравнительный анализ алгоритмов устойчивой сортировки**

Курсовой проект по дисциплине  
«Проектная и научно-исследовательская деятельность»

Выполнил студент группы ПМИб-2301-52-00  Жавнерко Максим Адреевич

Руководитель к.пед.н., доцент кафедры ПМИ  Котельникова Анастасия Валерьевна

Работа защищена с оценкой     \_\_\_\_.\_\_\_\_.2024 г.

Члены комиссии:     /     /

    /     /

Киров 2024

**Содержание**

[Введение 2](#_Toc167210486)

[1. Теоретические основы алгоритмов сортировки 5](#_Toc167210487)

[1.1 Постановка задачи 5](#_Toc167210488)

[1.2 Сортировка вставками 7](#_Toc167210489)

[1.3 Сортировка пузырьком 9](#_Toc167210490)

[1.4 Сортировка слиянием 11](#_Toc167210491)

[1.5 Сортировка подсчетом 13](#_Toc167210492)

[1.6 TimSort 15](#_Toc167210493)

[1.7 Выводы по теоретической главе 17](#_Toc167210494)

[2. Реализация и сравнение алгоритмов сортировки 18](#_Toc167210495)

[2.1 Квадратичные сортировки 18](#_Toc167210496)

[2.2 Сортировка подсчетом 19](#_Toc167210497)

[2.3 Логарифмические сортировки 21](#_Toc167210498)

[2.4 Анализ алгоритмов на малых массивах 22](#_Toc167210499)

[2.5 Сравнение количества перестановок, сравнений и перемещений 23](#_Toc167210500)

[2.4 Выводы по практической главе 27](#_Toc167210501)

[Заключение 28](#_Toc167210502)

[Библиографический список 29](#_Toc167210503)

[Приложения 30](#_Toc167210504)

[Приложение 1 30](#_Toc167210505)

[Приложение 2 30](#_Toc167210506)

Введение

Один из первых прототипов современного алгоритма сортировки применялся в конце 19 века для ускорения обработки переписи населения. В США американец Герман Холлерит создал сортировальную машину, работа которой основывалась на методах поразрядной сортировки.

В дальнейшем история алгоритмов связана с развитием [ЭВМ](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%BE-%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0), считается, что программа сортировки стала первой программой для ЭВМ. Некоторые конструкторы называли задачу сортировки данных наиболее характерной нечисловой задачей для вычислительных машин. В 1945 году [Джон фон Нейман](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD,_%D0%94%D0%B6%D0%BE%D0%BD_%D1%84%D0%BE%D0%BD) для тестирования ряда команд разработал программы [сортировки методом слияния](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%BB%D0%B8%D1%8F%D0%BD%D0%B8%D0%B5%D0%BC). В том же году немецкий инженер [Конрад Цузе](https://ru.wikipedia.org/wiki/%D0%A6%D1%83%D0%B7%D0%B5,_%D0%9A%D0%BE%D0%BD%D1%80%D0%B0%D0%B4) разработал программу для [сортировки методом простой вставки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8).

Сортировка ­ это фундаментальная операция в информатике, используемая для упорядочивания элементов в определенном порядке. От эффективности алгоритмов сортировки напрямую зависит быстрота работы многих программ и систем.

Особое внимание заслуживают стабильные сортировки. Алгоритм сортировки считается стабильным, если он сохраняет относительный порядок элементов с одинаковыми значениями. Например, если мы сортируем список студентов по возрасту, стабильная сортировка гарантирует, что студенты с одинаковым возрастом будут расположены в том же порядке, что и в исходном списке.

Объектом исследования являются алгоритмы сортировки данных, их слабые и сильные стороны.

Целью курсового проекта является формирование рекомендации по выбору алгоритма

Для достижения цели курсовой работы были поставлены следующие задачи:

1. Провести обзор научных исследований, посвященных алгоритмам стабильной сортировки. Изучить основные алгоритмы, их характеристики, преимущества и недостатки, а также проанализировать примеры их применения в различных областях.
2. Описание и анализ основных алгоритмов стабильной сортировки.
3. Найти или сгенерировать доступные наборы данных из различных предметных областей, обеспечивая разнообразие по сортируемым элементам.
4. Реализовать выбранные алгоритмы и применить их для сортировки на подготовленных наборах данных.
5. Сравнить эффективность различных алгоритмов на основе выбранных метрик качества.
6. Сформулировать выводы о применимости каждого алгоритма, а также дать рекомендации по выбору оптимального метода.

В первой главе проведен обзор существующих алгоритмов и их теоретическое время работы, а также описана их область применения. Вторая глава посвящена анализу производительности. Приложение содержит листинг программы.

1. Теоретические основы алгоритмов сортировки
   1. Постановка задачи

Пусть ­ входная последовательность элементов, где

Определим отношение порядка на элементах

Сортировка называется стабильной, если для любых двух индексов и где , выполняется следующее условие:

Если , то , где и .

Для эффективного алгоритма сортировки время выполнения стремится к в лучшем и среднем случае. Менее эффективные алгоритмы демонстрируют время выполнения в худшем случае. Идеальным сценарием является линейная временная сложность . Стоит отметить, что для алгоритмов сортировки, основанных на сравнении элементов, является нижней границей сложности, то есть не существует алгоритма сравнения, способного отсортировать массив быстрее, чем за [5, стр. 364, 365].

Важно учитывать не только временную, но и пространственную сложность алгоритма. Многие алгоритмы сортировки требуют выделения дополнительной памяти, которая, как правило, составляет , то есть не зависит от размера входных данных. При анализе пространственной сложности не учитывается память, занимаемая исходным массивом, а также постоянные затраты, не зависящие от входных данных, такие как объем программного кода, что также соответствует .

Таким образом, при оценке эффективности алгоритма сортировки ключевыми параметрами являются временная сложность, определяющая скорость работы алгоритма, и пространственная сложность, характеризующая объем требуемой дополнительной памяти. Выбор оптимального алгоритма сортировки для конкретной задачи определяется балансом между этими двумя характеристиками с учетом особенностей обрабатываемых данных.

В рамках исследования были рассмотрены несколько алгоритмов, гарантирующие стабильность сортировки:

1. Сортировка вставками (Insertion Sort): Алгоритм, основанный на последовательном перемещении элементов на свои места путем сравнения с уже отсортированной частью массива.
2. Сортировка слиянием (Merge Sort): Алгоритм, основанный на принципе "разделяй и властвуй", рекурсивно разбивающий массив на подмассивы, сортирующий их и затем объединяющий в отсортированный массив.
3. Сортировка подсчетом (Counting Sort): Алгоритм, работающий за линейное время, подсчитывающий количество элементов с каждым значением и на основе этих данных строящий отсортированный массив.
4. Пузырьковая сортировка (Bubble Sort): Алгоритм, основанный на последовательном сравнении соседних элементов и их перестановке, если порядок нарушен, повторяя этот процесс до полной сортировки.
5. Сортировка поразрядно (Radix Sort): Сортирует по цифрам, начиная с младших разрядов.
6. Блочная сортировка (Bucket Sort): Распределяет элементы по "блокам" и сортирует внутри каждого блока.
7. Timsort: Гибридный алгоритм, использует сортировку вставками и слиянием, разработан для работы с реальными данными.

Для программной реализации были выбраны сортировки вставками, слияниями, подсчетом, пузырьком и TimSort.

Далее необходимо рассмотреть алгоритмы вышеназванных сортировок, показать их принцип работы на конкретных примерах.

* 1. Сортировка вставками

1. Инициализация:
   * Считаем, что подпоследовательность уже отсортирована.
2. Итерации:
   * Для каждого от до :
     + Пока и :

На каждой итерации внешнего цикла подпоследовательность $A[1..i]$ отсортирована.

Доказательство корректности:

* Базовый случай: При подпоследовательность тривиально отсортирована.
* Индуктивный шаг: Предположим, что для подпоследовательность отсортирована. На шаге алгоритм вставляет элемент на правильную позицию в уже отсортированной подпоследовательности , сравнивая его с элементами в обратном порядке и сдвигая большие элементы вправо. После завершения внутреннего цикла оказывается на правильной позиции, и подпоследовательность становится отсортированной.
* Завершение: По завершении цикла , вся последовательность отсортирована.

Временная сложность:

* Худший случай: (когда входная последовательность отсортирована в обратном порядке)
* Средний случай:
* Лучший случай: (когда входная последовательность уже отсортирована)

Пространственная сложность: (алгоритм сортирует на месте)

Стабильность: Сортировка вставками является стабильной, так как внутренний цикл перемещает элементы только в том случае, если они строго больше, сохраняя порядок равных элементов.

Пример работы сортировки вставками на массиве

1. Итерация 1:
   * Отсортированная часть:
   * Берем элемент 1.
   * Сравниваем 1 с 5 и меняем их местами:
2. Итерация 2:
   * Отсортированная часть:
   * Берем элемент 4.
   * Сравниваем 4 с 5, они в правильном порядке.
   * Сравниваем 4 с 1 и вставляем 4 перед 5:
3. Итерация 3:
   * Отсортированная часть:
   * Берем элемент 2.
   * Сравниваем 2 с 5, 2 меньше.
   * Сравниваем 2 с 4, 2 меньше.
   * Сравниваем 2 с 1 и вставляем 2 перед 4:
4. Итерация 4:
   * Отсортированная часть:
   * Берем элемент 8.
   * Сравниваем 8 с 5, 8 больше.
   * 8 уже находится в правильном месте:

Результат: .

* 1. Сортировка пузырьком

1. Инициализация:
2. Итерации:
   * Пока :
     + Для каждого от 1 до :
       - Если :
         * Поменять местами и

На каждой итерации внешнего цикла наибольший неотсортированный элемент перемещается на свою окончательную позицию в конце последовательности.

Доказательство корректности:

* Базовый случай: Если , последовательность уже отсортирована.
* Индуктивный шаг: Предположим, что после итераций наибольших элементов находятся на своих окончательных позициях в конце последовательности. На -ой итерации алгоритм сравнивает соседние элементы и меняет их местами, если порядок нарушен. В результате наибольший из оставшихся неотсортированных элементов перемещается на -ю позицию, занимая своё окончательное место.
* Завершение: Цикл завершается, когда , что означает, что на последней итерации не было произведено ни одной перестановки. Это гарантирует, что все элементы находятся на своих окончательных позициях, и последовательность отсортирована.

Временная сложность:

* Худший случай: (когда входная последовательность отсортирована в обратном порядке)
* Средний случай:
* Лучший случай: (когда входная последовательность уже отсортирована)

Пространственная сложность: (алгоритм сортирует на месте)

Стабильность: Сортировка вставками является стабильной, так как внутренний цикл перемещает элементы только в том случае, если они строго больше, сохраняя порядок равных элементов.

Пример работы сортировки пузырьком на массиве :

1. Итерация 1:
   * Сравниваем 5 и 1, меняем их местами:
   * Сравниваем 5 и 4, меняем их местами:
   * Сравниваем 5 и 2, меняем их местами:
   * Сравниваем 5 и 8, они уже в правильном порядке.
2. Итерация 2:
   * Сравниваем 1 и 4, они уже в правильном порядке.
   * Сравниваем 4 и 2, меняем их местами:
   * Сравниваем 4 и 5, они уже в правильном порядке.
   * Сравниваем 5 и 8, они уже в правильном порядке.
3. Итерация 3:
   * Сравниваем 1 и 2, они уже в правильном порядке.
   * Сравниваем 2 и 4, они уже в правильном порядке.
   * Сравниваем 4 и 5, они уже в правильном порядке.
   * Сравниваем 5 и 8, они уже в правильном порядке.

Результат: .

* 1. Сортировка слиянием

Сортировка слиянием реализуется с помощью двух основных процедур:

1. Merge(L, R): Слияние двух отсортированных последовательностей и в одну отсортированную последовательность .
   * Инициализация: , , .
   * Пока и :
     + Если :
     + Иначе:
   * Скопировать оставшиеся элементы (если ) или (если ) в .
2. MergeSort(A): Рекурсивная сортировка последовательности .
   * Если длина() :
     + Вернуть
   * Иначе:
     + Вернуть

Доказательство корректности:

* Корректность Merge: Процедура Merge корректно сливает две отсортированные последовательности, так как на каждом шаге выбирается наименьший элемент из оставшихся в и и добавляется в , гарантируя возрастающий порядок элементов в .
* Корректность MergeSort:
  + Базовый случай: Если длина() , последовательность тривиально отсортирована.
  + Индуктивный шаг: Предполагаем, что корректно сортирует последовательности длины меньше . Для последовательности длины алгоритм разбивает ее на две подпоследовательности длины не более , рекурсивно сортирует их с помощью (что корректно по предположению) и затем сливает отсортированные подпоследовательности с помощью (что также корректно).
* Завершение: По завершении рекурсии возвращает отсортированную последовательность .

Временная сложность:

* Худший случай:
* Средний случай:
* Лучший случай:

Пространственная сложность: (inplace реализация)

Стабильность: Сортировка слиянием является стабильной, так как процедура при равных элементах и выбирает , сохраняя порядок элементов из исходной последовательности.

Пример работы сортировки слиянием на массиве :

1. Разделение:
2. Слияние:

Результат: .

* 1. Сортировка подсчетом

Алгоритм:

1. Подсчет:
   * Создать массив размера , инициализированный нулями.
   * Для каждого элемента в :
2. Суммирование:
   * Для каждого от 1 до :
3. Распределение:
   * Создать массив размера .
   * Для каждого от до 1 (в обратном порядке):

Доказательство корректности:

* Подсчет: Массив хранит количество вхождений каждого значения из диапазона элементов .
* Суммирование: После суммирования содержит количество элементов в , меньших или равных .
* Распределение: Используя информацию из , алгоритм помещает каждый элемент на его окончательную позицию в , гарантируя сортировку.

Временная сложность:

Пространственная сложность: (для массивов и )

Стабильность: Сортировка подсчетом является стабильной, так как элементы с одинаковым значением обрабатываются в том же порядке, в котором они встречаются во входной последовательности, благодаря обратному порядку обработки в шаге "Распределение".

Пример работы сортировки подсчётом на массиве

1. Создание массива подсчёта:
   * Создаем массив counts, размер которого равен максимальному значению в исходном массиве + 1 (в нашем случае 9, так как максимальное значение - 8). Инициализируем все элементы нулями.
2. Подсчёт элементов:
   * Проходим по исходному массиву и увеличиваем значение соответствующего элемента в counts на 1.
   * Например, для элемента 5 увеличиваем counts[5] на 1.
   * В итоге counts будет выглядеть так:
3. Построение отсортированного массива:
   * Проходим по массиву counts и добавляем в новый массив sorted\_array столько элементов, сколько указано в counts.
   * Например, counts[1] равен 1, поэтому добавляем один элемент 1 в sorted\_array.

Результат: .

* 1. TimSort

Алгоритм:

1. Разбиение на прогоны:
   * Входной вектор делится на подмассивы фиксированного размера RUN (ран).
   * Каждый подмассив сортируется с использованием insertionSortTim.
2. Слияние прогонов:
   * Последовательно сливаются пары соседних отсортированных прогонов (mergeTim), размер которых удваивается на каждой итерации (size).
   * Процесс слияния продолжается до тех пор, пока не будет получен единый отсортированный вектор.

Доказательство корректности:

* insertionSortTim: Гарантирует сортировку каждого прогона, так как использует классический алгоритм сортировки вставками.
* mergeTim: Обеспечивает корректное слияние отсортированных подмассивов, сохраняя относительный порядок элементов.

Временная сложность:

* insertionSortTim: в худшем случае, но эффективна для небольших массивов (прогонов).
* mergeTim: для слияния двух подмассивов размера n.
* Общее время: В среднем и худшем случае Timsort имеет сложность , как и другие эффективные алгоритмы сортировки. Благодаря использованию сортировки вставками для небольших прогонов, Timsort может показывать лучшую производительность на частично отсортированных данных.

Пространственная сложность:

* mergeTim: для временных векторов left и right.
* Общее пространство: в худшем случае.

Стабильность:

* insertionSortTim: Стабилен, так как не меняет порядок элементов с одинаковыми ключами.
* mergeTim: Также стабилен, так как сохраняет относительный порядок элементов при слиянии.

Пример работы Timsort на массиве с размером рана 4:

1. Разбиение на раны:

* Исходный массив:
* Делим на раны размером 4:
  + Ран 1:
  + Ран 2:
  + Ран 3: (последний ран может быть меньше 4)

2. Сортировка ранов:

* Сортируем каждый ран с помощью сортировки вставками:
  + Ран 1:
  + Ран 2:
  + Ран 3:

3. Слияние ранов:

* + Сливаем ран 1 и ран 2:
    - и
  + Сливаем полученный ран с ран 3:
    - и

Результат: .

* 1. Выводы по теоретической главе

Сортировка вставками и пузырьковая сортировка, обладая временной сложностью в общем случае, демонстрируют низкую эффективность при работе с большими объемами данных. Однако, их простота реализации и минимальные требования к дополнительной памяти () делают их подходящим выбором для сортировки небольших массивов.

Сортировка слиянием отличается стабильной временной сложностью , гарантируя предсказуемое время работы независимо от характера входных данных.

Timsort ­ гибридный алгоритм, сочетающий в себе сортировку вставками и сортировку слиянием. Он разработан для эффективной работы с реальными данными, которые часто имеют частично отсортированные подпоследовательности.

Сортировка подсчётом является наиболее эффективным алгоритмом из рассмотренных, демонстрируя линейную временную сложность , где k ­ диапазон значений сортируемых элементов. Однако, эффективность этого алгоритма напрямую зависит от диапазона сортируемых значений: при большом значении k потребление памяти может стать неприемлемым. Кроме того этот алгоритм накладывает существенные ограничения на сортируемые объекты (у каждого объекта должно быть отображение в множество целых чисел с нулем).

1. Реализация и сравнение алгоритмов сортировки

Для разработки и анализа методов сортировки мы будем использовать язык программирования C++20 и библиотеку Google Benchmark (<https://github.com/google/benchmark> ).

Выбор C++ для анализа алгоритмов сортировки в данной курсовой работе обусловлен стремлением к максимальной точности и эффективности исследования. C++, будучи компилируемым языком, демонстрирует непревзойденную производительность и минимальные накладные расходы ("zero overhead"), что позволяет получать достоверные данные о временных характеристиках алгоритмов.

Несмотря на низкоуровневую природу, C++ предоставляет обширную стандартную библиотеку (STL), включающую в себя высокоуровневые структуры данных и алгоритмы. STL обеспечивает необходимый уровень абстракции, позволяя исследователю сосредоточиться на анализе алгоритмов, не отвлекаясь на рутинные задачи реализации базовых функций.

* 1. Квадратичные сортировки

Проанализируем средний случай для сортировки вставками и пузырьком (случайный массив):

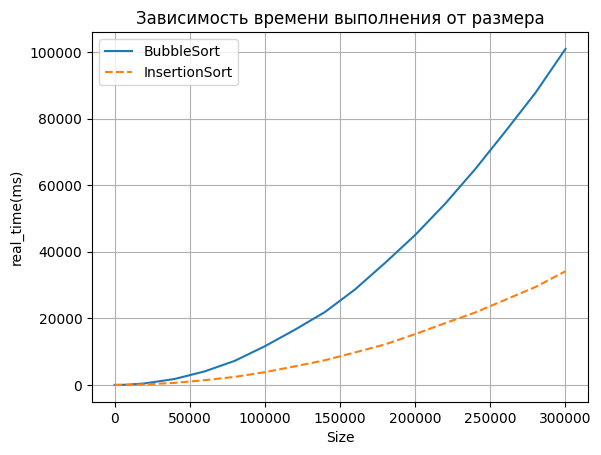


Рисунок 1 Зависимость времени выполнения от размера массива в среднем случае для сортировки вставками и пузырьком

График показывает правильность временной сложности, рассчитанной теоретически.

Подробные результаты бенчмарков представлены в таблицах в приложении (ЗДЕСЬ БУДЕТ ССЫЛКА НА ПРИЛОЖЕНИЕ)

* 1. Сортировка подсчетом

Проанализируем случай, когда уникальных элементов около 1% (в целом такой случай можно назвать средним):

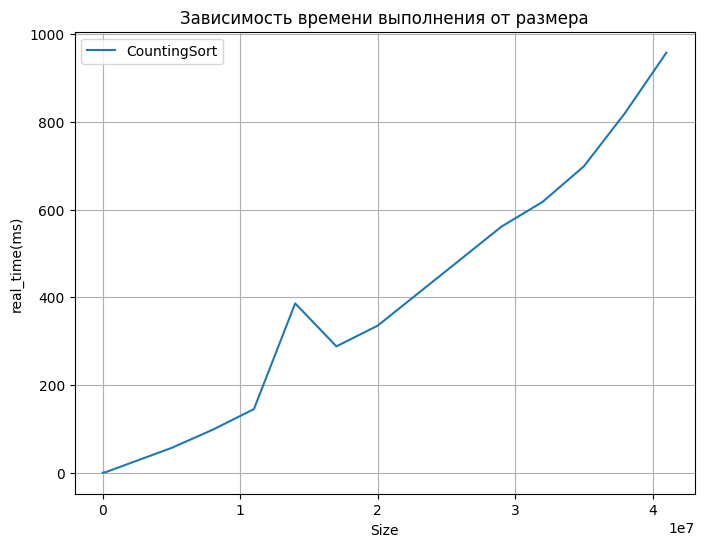


Рисунок 2 Зависимость времени выполнения сортировки подсчетом от времени в случае 1% уникальных элементов

Мы видим, что время работы алгоритма хорошее. Это хорошо видно на графике с логарифмическими соритровками.

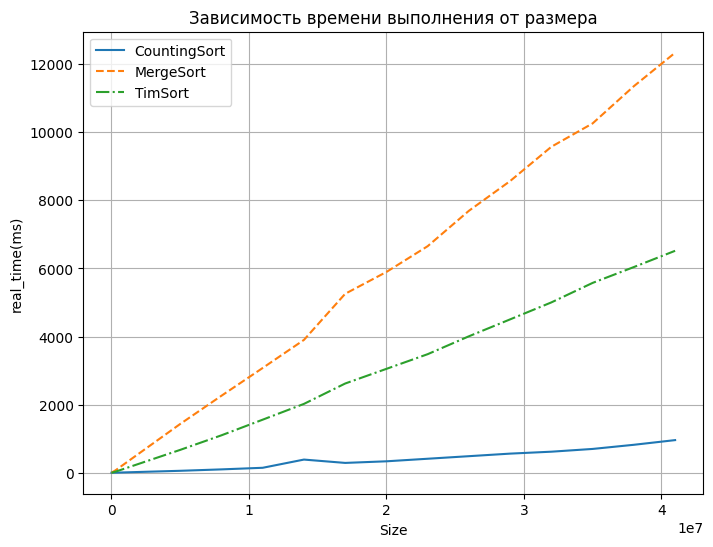


Рисунок 3 Зависимость времени выполнения от размера логарифмических сортировок и сортировки подсчетом

* 1. Логарифмические сортировки

Проанализируем средний случай (случайный массив):

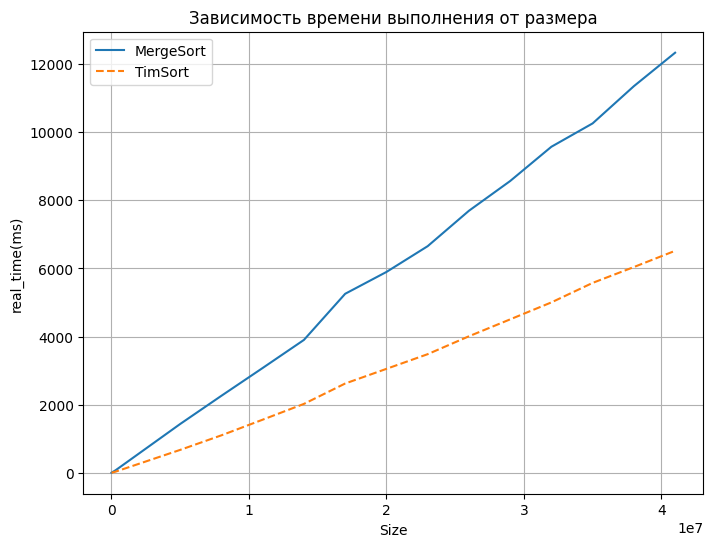


Рисунок 4 Зависимость времени выполнения от размера массива для логарифмических сортировок

График иллюстрирует эффективность TimSort над сортировкой слиянием в среднем случае.

Полное сравнение в таблицах находится в приложении (ЗДЕСЬ БУДЕТ ССЫЛКА НА ПРИЛОЖЕНИЕ)

* 1. Анализ алгоритмов на малых массивах

Средний случай (тесты были запущены достаточное количество раз для стабильного разультата) :

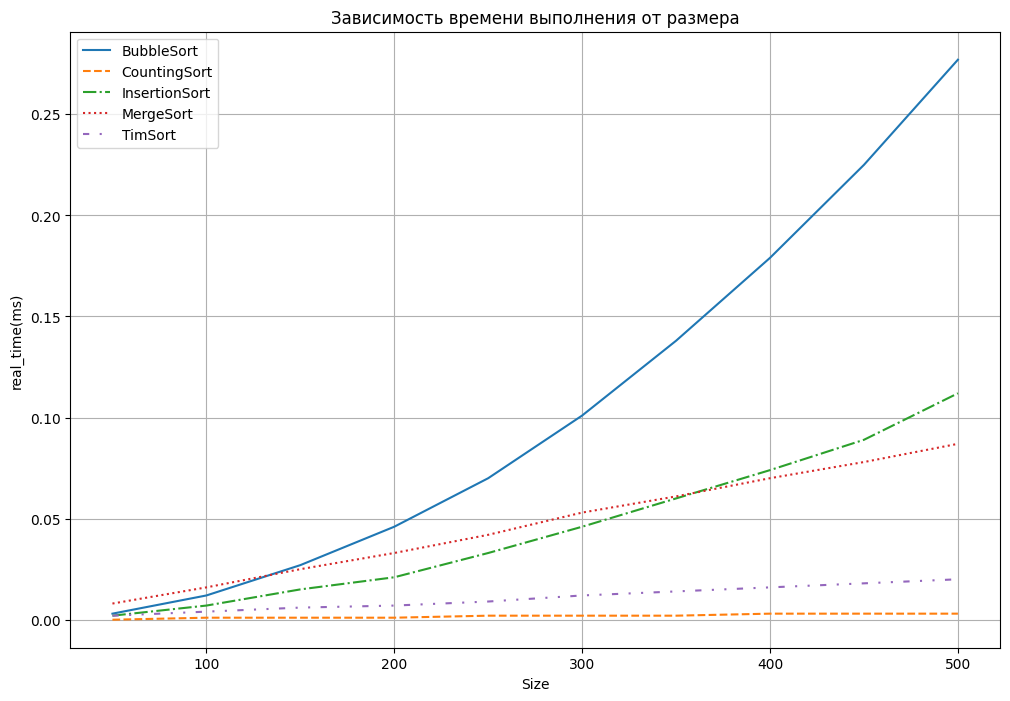


Рисунок 5 Зависимость времени выполнения от размера массива на малых графах

По графику можно сделать несколько выводов:

* + 1. На малых графах квадратичные сортировки могут показывать себя лучше, чем сортировка слиянием
    2. TimSort являясь ансамблем из двух алгоритмов, показывает лучшее время среди сортировок, основанных на сравнении.
  1. Сравнение количества перестановок, сравнений и перемещений

В этой главе исследовались сортировки, основанные на сравнении.

Рассмотрим несколько графиков, наглядно в сравнении показывающих особенности алгоритмов

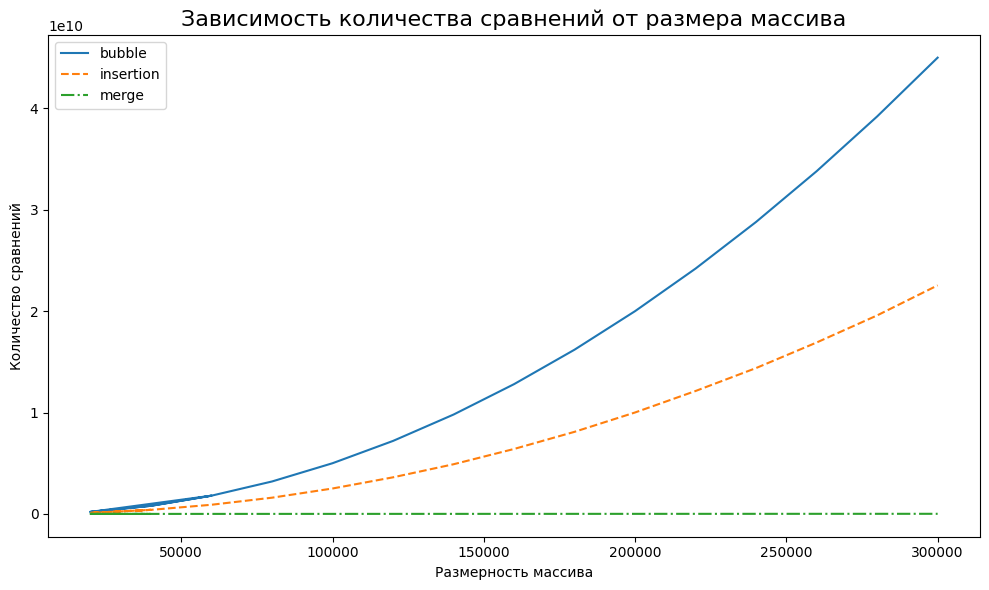


Рисунок 6 Зависимость количества сравнений от размера массива (кроме TimSort)

По рисунку выше хорошо заметно, что количество операций сравнения для сортировки пузырьком и сортировка вставками в среднем случае растет квадратично.

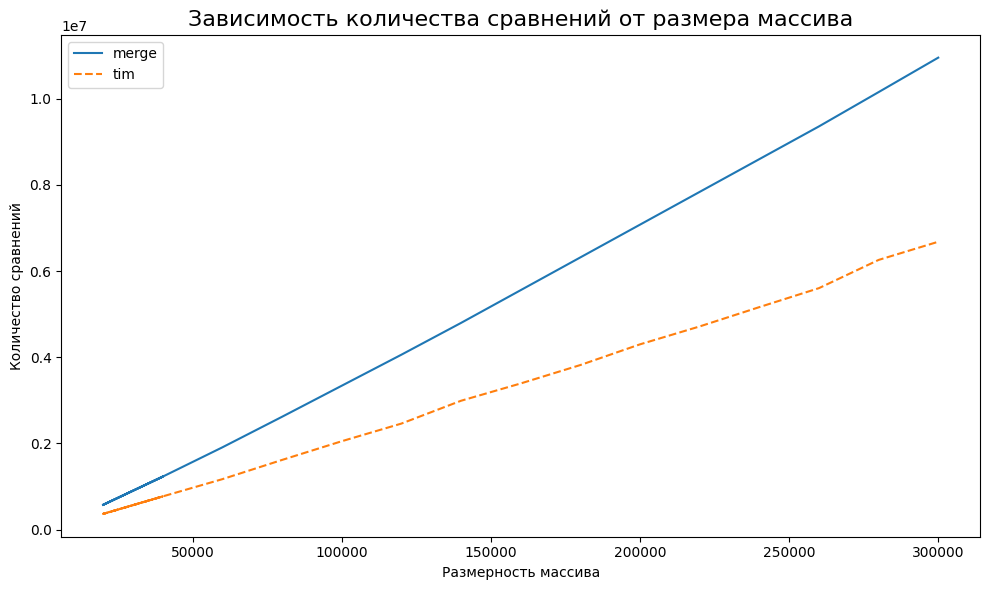


Рисунок 7 Зависимость количества сравнений от размера массива (TimSort и сортировка слиянием)

В TimSort в среднем меньше сравнений

Анализ перестановок является более сложным, так как в сортировке слияниями нет перестановок (только перемещения элементов), а в сортировке вставками количество перемещений равно количеству элементов. Поэтому имеет смысл 3 перемещения считать за одну перестановку. Разберемся, почему это справедливо. Если есть 2 элемента A и B, которые нужно поменять местами, чтобы сделать это необходимо прочитать элемент A, записать в его промежуточную переменную, прочитать элемент B, записать его на место A, прочитать из временного хранилища элемент A, записать его на место B ­ итого 3 чтения и 3 записи. Перемещение представляет собой только 1 чтение и 1 запись в другую ячейку памяти.

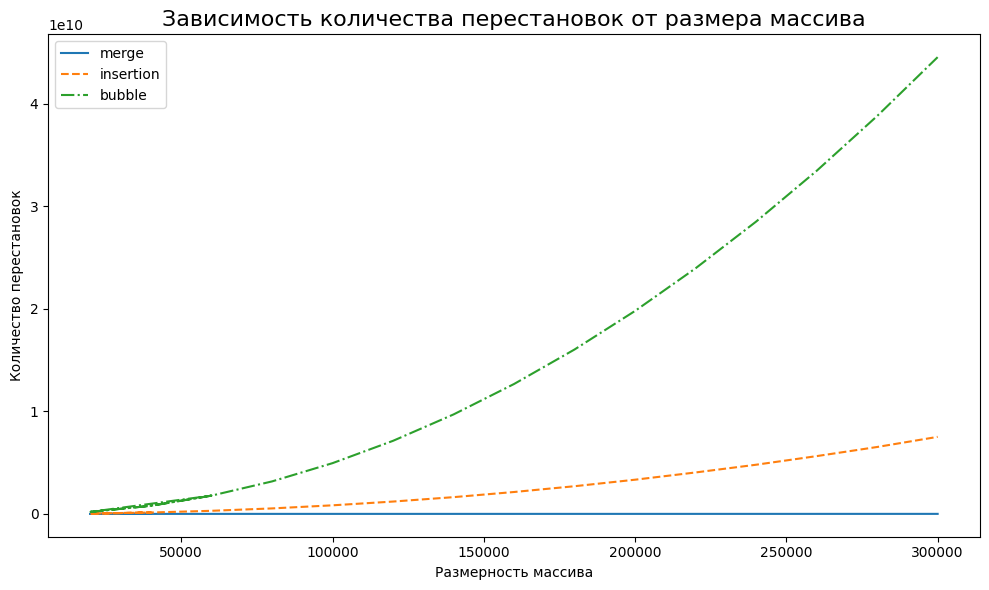


Рисунок 8 Зависимость количества перестановок от размера массива (кроме TimSort)

На рисунке выше снова наблюдается квадратичная зависимость количества перестановок от времени.

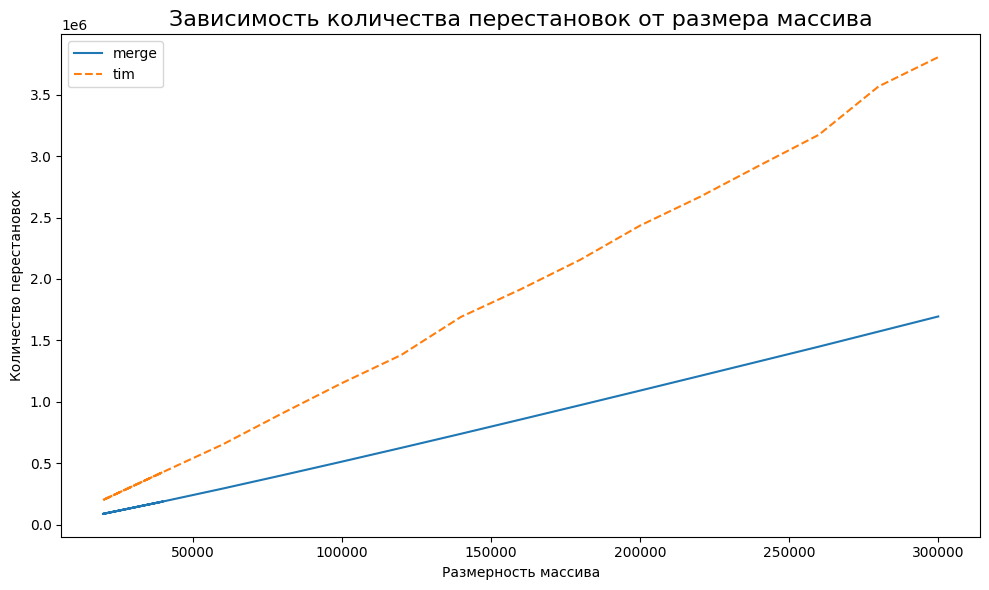


Рисунок 9 Зависимость количества перестановок от размера массива (TimSort и сортировка слиянием)

Как и ожидалось TimSort, имея в своей реализации сортировку вставками имеет большее количество перестановок, чем сортировка слияниями.

2.4 Выводы по практической главе

Сортировка вставками продемонстрировала более высокую скорость работы по сравнению с пузырьковой сортировкой на всех протестированных наборах данных. Это подтверждает теоретические предпосылки о большей эффективности сортировки вставками, особенно на частично отсортированных массивах.

В случаях, когда данные соответствовали ограничениям сортировки подсчетом (целочисленные значения в ограниченном диапазоне), этот алгоритм значительно превзошел по скорости все остальные рассмотренные алгоритмы. Это доказывает, что сортировка подсчетом является оптимальным выбором при наличии подходящих условий.

Timsort – лидер среди универсальных алгоритмов. Timsort продемонстрировал наилучшую производительность среди алгоритмов с логарифмической сложностью, опередив сортировку слиянием. Это подчеркивает эффективность гибридного подхода Timsort, который использует преимущества сортировки вставками для оптимизации работы с реальными данными.

На малых объемах данных квадратичные алгоритмы (сортировка вставками и пузырьком) показали себя конкурентоспособными по сравнению с более сложными алгоритмами. Это объясняется низкими накладными расходами, связанными с их реализацией, что делает их эффективными для небольших массивов.

Заключение

В данном курсовом проекте были рассмотрены пять различных методов стабильной сортировки данных: сортировка вставками, сортировка пузырьком, сортировка слиянием, сортировка подсчетом и Timsort. Выбор этих алгоритмов обусловлен их распространенностью, различными подходами к сортировке и широким спектром применения. Особое внимание было уделено практической стороне вопроса: все алгоритмы были реализованы вручную (Приложение 1), что позволило глубже понять принципы их работы.

Были получены практические результаты времени работы алгоритмов в зависимости от размера входного массива (Приложение 2). Анализ результатов показал, что сортировка слиянием и Timsort являются наиболее эффективными методами для больших массивов данных. Сортировка пузырьком оказалась наименее эффективной в большинстве случаев, что подтверждает ее теоретическую сложность.

Дальнейшее исследование может быть направлено на оптимизацию реализованных алгоритмов, добавление в сравнение других методов стабильной сортировки, а также на изучение адаптации алгоритмов к различным типам данных и параллельной реализации для многоядерных систем. Также представляется перспективным исследование влияния аппаратных особенностей на производительность алгоритмов.

Библиографический список

1. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск. – Litres, 2018.
2. Ефимов С. С. Обзор методов распараллеливания алгоритмов решения некоторых задач вычислительной дискретной математики // МСиМ. 2007. №1 (17). URL: https://cyberleninka.ru/article/n/obzor-metodov-rasparallelivaniya-algoritmov-resheniya-nekotoryh-zadach-vychislitelnoy-diskretnoy-matematiki (дата обращения: 15.12.2022).
3. Список рассылки разработчиков Python : сайт. – URL: https://mail.python.org/pipermail/python-dev/2002-July/026837.html (дата обращения: 17.12.2022)
4. Nicolas Auger, Vincent Jugé, Cyril Nicaud, Carine Pivoteau. On the Worst-Case Complexity of TimSort. 26th Annual European Symposium on Algorithms (ESA 2018), Aug 2018, Helsinki, Finland. pp.4:1–4:13, 10.4230/LIPIcs.ESA.2018.4. hal-01798381
5. Mishra, Aditya Dev, and Deepak Garg. "Selection of best sorting algorithm." *International Journal of intelligent information Processing* 2.2 (2008): 363-368.

Приложения

Приложение 1

ТУТ БУДЕТ КОД

Приложение 2

ТУТ БУДЕТ КОД