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 Андреевич

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

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

Члены комиссии:     /     /

    /     /

Киров 2024

**Содержание**

[Введение 4](#_Toc167904196)

[1. Теоретические основы алгоритмов сортировки 6](#_Toc167904197)

[1.1 Постановка задачи 6](#_Toc167904198)

[1.2 Сортировка вставками 8](#_Toc167904199)

[1.3 Сортировка пузырьком 11](#_Toc167904200)

[1.4 Сортировка слиянием 13](#_Toc167904201)

[1.5 Сортировка подсчетом 15](#_Toc167904202)

[1.6 TimSort 17](#_Toc167904203)

[1.7 Библиотечная сортировка 19](#_Toc167904204)

[1.8 Выводы по теоретической главе 22](#_Toc167904205)

[2. Реализация и сравнение алгоритмов сортировки 24](#_Toc167904206)

[2.1 Квадратичные сортировки 24](#_Toc167904207)

[2.2 Сортировка подсчетом 27](#_Toc167904208)

[2.3 Логарифмические сортировки 29](#_Toc167904209)

[2.4 Анализ алгоритмов на малых массивах 31](#_Toc167904210)

[2.5 Сравнение количества перестановок, сравнений и перемещений 32](#_Toc167904211)

[2.4 Выводы по практической главе 36](#_Toc167904212)

[Заключение 38](#_Toc167904213)

[Библиографический список 39](#_Toc167904214)

[Приложения 40](#_Toc167904215)

[Приложение 1 40](#_Toc167904216)

[Приложение 2 46](#_Toc167904217)

[Приложение 3 52](#_Toc167904218)

Введение

В конце 19 века, стремясь ускорить обработку данных переписи населения, американец Герман Холлерит создал один из первых прототипов современного алгоритма сортировки. Его сортировальная машина работала на основе методов поразрядной сортировки и стала важным шагом в развитии вычислительной техники.

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 анализ теоретического времени работы алгоритмов, 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.
8. Library Sort: Алгоритм, основанный на сортировке вставками, но использующий дополнительные структуры данных (например, динамические массивы или бинарные деревья), чтобы ускорить вставку элементов в отсортированный подмассив.
9. Odd–even Sort (Чётно-нечётная сортировка): Итеративный алгоритм, основанный на сравнении и обмене соседних элементов с чётными и нечётными индексами на каждом проходе. Эффективен для параллельных вычислений, но на практике уступает другим алгоритмам по скорости на однопроцессорных системах.
10. Cocktail Shaker Sort (Шейкерная сортировка): Вариация пузырьковой сортировки, где проходы по массиву чередуются в прямом и обратном направлениях, что позволяет "всплывать" большим элементам в конец массива быстрее.

Выбор сортировок вставками, слияниями, подсчётом, пузырьком и TimSort и LibrarySort для программной реализации обусловлен стремлением охватить широкий спектр алгоритмов с разными характеристиками, демонстрируя их сильные и слабые стороны на практике.

* Разная временная сложность: Пузырёк является простым примером алгоритма с квадратичной сложностью , Вставки в худшем случае также имеет , но на практике часто работает быстро. Слияния и TimSort демонстрируют более эффективную, логарифмически линейную сложность , что делает их предпочтительными для больших объёмов данных. Алгоритм "подсчётом" выделяется линейной сложностью , но применим только для целых чисел в ограниченном диапазоне.
* Разная пространственная сложность: сортировка слиянием требует дополнительной памяти для хранения промежуточных результатов (), в то время как остальные алгоритмы работают "на месте" с .
* Сортировка, не основанная на сравнении: подсчёт не использует сравнения элементов, а оперирует количеством встречающихся значений, что делает его уникальным и потенциально быстрым в определённых условиях.
* Сортировки, отличные от пройденных в курсе КАиСД: LibrarySort 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 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: [8, стр. 122]

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: [7, стр. 1389]

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 [6, стр. 12].

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. Библиотечная сортировка

1. Инициализация:
   * Создать два вспомогательных массива, library1 и library2, одинакового размера с arr.
   * Создать массив gaps для хранения индексов элементов, временно находящихся вне основного массива.
   * Создать массив numbered для отслеживания занятых позиций в library1 и library2.
   * Установить размер используемой части library (например, library1) равным 1.
   * Поместить первый элемент arr в первую позицию используемой library.
2. Основной цикл:
   * Для каждого элемента elem в arr (начиная со второго):
     + С помощью бинарного поиска найти позицию pos для вставки elem в отсортированной части используемой library.
     + Если позиция pos свободна (не занята другим элементом):
       - Вставить elem в gaps на позицию pos.
       - Отметить позицию pos в numbered как занятую.
     + Иначе (позиция pos занята):
       - Скопировать все элементы из gaps и используемой library в другую library (например, из library1 в library2), поддерживая отсортированный порядок.
       - Очистить gaps и numbered.
       - Повторить шаги для вставки elem.
3. Слияние:
   * Скопировать все элементы из gaps и используемой library обратно в исходный массив arr, поддерживая отсортированный порядок.

Доказательство корректности сводится к корректности сортировки вставками.

Временная сложность:

Наихудший случай:

* + В наихудшем случае, когда массив уже отсортирован в обратном порядке, каждый новый элемент нужно будет вставлять в начало отсортированного подмассива.
  + Это приведёт к многократному копированию больших блоков данных между library1 и library2.

Средний случай:

* + В среднем случае, если данные распределены случайно, бинарный поиск позиции для вставки займет времени для каждого элемента.
  + Однако, перемещение элементов между library1, library2 и gaps все равно займет в некоторых случаях.

Наилучший случай: – аналогично среднему случаю.

Пространственная сложность: , так как алгоритм использует дополнительные массивы (library1, library2, gaps) того же размера, что и исходный массив.

Пример работы Library Sort на массиве [5, 2, 4, 6, 1, 3]

Инициализация:

* arr = [5, 2, 4, 6, 1, 3]
* library1 = [ , , , , , ]
* library2 = [ , , , , , ]
* gaps = [ , , , , , ]
* numbered = [false, false, false, false, false, false]
* lib\_size = 1 (используем library1)
* library1 = [5, , , , , ]

Шаг 1: elem = 2

* Бинарный поиск позиции для 2 в library1: pos = 0
* pos = 0 свободна:
  + gaps = [2, , , , , ]
  + numbered = [true, false, false, false, false, false]

Шаг 2: elem = 4

* Бинарный поиск позиции для 2 в library1: pos = 0
* pos = 0 занята:
* очищаем pos и numbered
* сливаем gaps и library в один массив library2 = [2, 5, , , , ]

Шаг 3: elem = 6

* Бинарный поиск для 6: pos = 2
* pos = 2 сбоводна:
  + - gaps = [, , 6, , , ]
    - numbered = [false, false, true, false, false, false]

Шаг 4: elem = 1

* Бинарный поиск для 1: pos = 0
* pos = 0 свободна:
  + - gaps = [1, , 6, , , ]
    - numbered = [true, false, true, false, false, false]

Шаг 4: elem = 3

* Бинарный поиск для 3: pos = 1
* pos = 1 свободна:
  + - gaps = [1, 3, 6, , , ]
    - numbered = [true, false, true, false, false, false]

Слияние:

* arr = [1, 2, 3, 4, 5, 6] (объединяем элементы из gaps и library2)
  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).

В идеальном случае, при равномерном распределении данных и правильном выборе размера пробелов, Library sort может достичь временной сложности . Однако, на практике выбор оптимального размера и количества пробелов является сложной задачей. Более того, в худшем случае, при неблагоприятных входных данных, временная сложность алгоритма может деградировать до .

Кратко весь анализ можно свести к таблице ниже:

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Средняя временная сложность | Худшая | Пространственная  сложность |
| InsertionSort |  |  |  |
| MergeSort |  |  |  |
| CountingSort |  |  |  |
| BubbleSort |  |  |  |
| TimSort |  |  |  |
| LibrarySort |  |  |  |

1. Реализация и сравнение алгоритмов сортировки

Для разработки и анализа методов сортировки мы будем использовать язык программирования C++ (стандарт 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.

Для тестирования алгоритмов подходит не так много данных (из-за сортировки подсчетом), поэтому был сгенерирован один набор данных случайных целых чисел.

Все графики в этой главе были построены по таблицам из приложения 1. Реализация алгоритмов находится в приложении 2, бенчмарки находятся в приложении 3.

* 1. Квадратичные сортировки

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

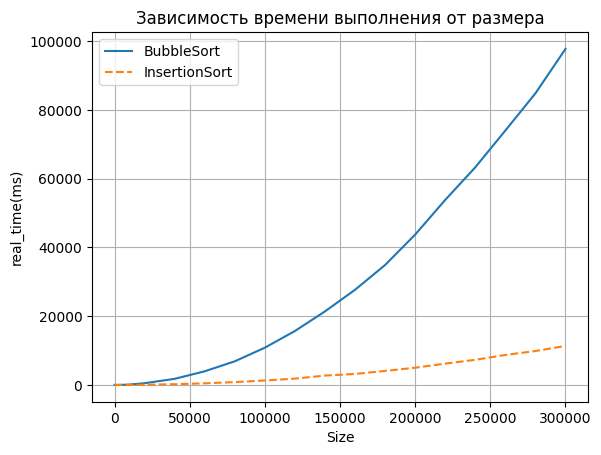


Рисунок 1 Зависимость времени выполнения от размера массива в среднем случае для сортировки вставками, пузырьком

График показывает правильность временной сложности, рассчитанной теоретически.

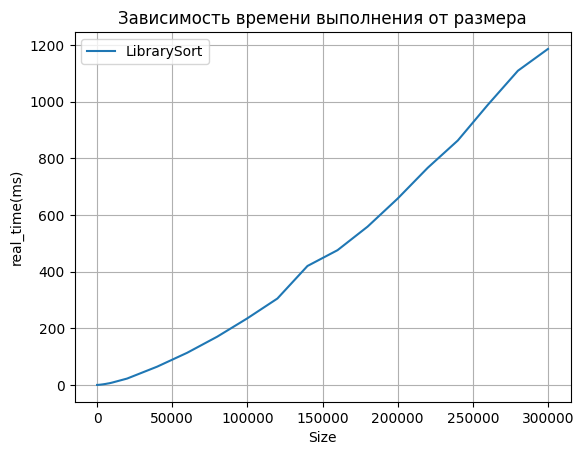


Рисунок ‑2 Скорость выполнения LibrarySort

График LibrarySort в целом соответствует графику . При этом на фоне квадратичных сортировок заметна хорошая прибавка к скорости. Рисунок ниже иллюстрирует это.

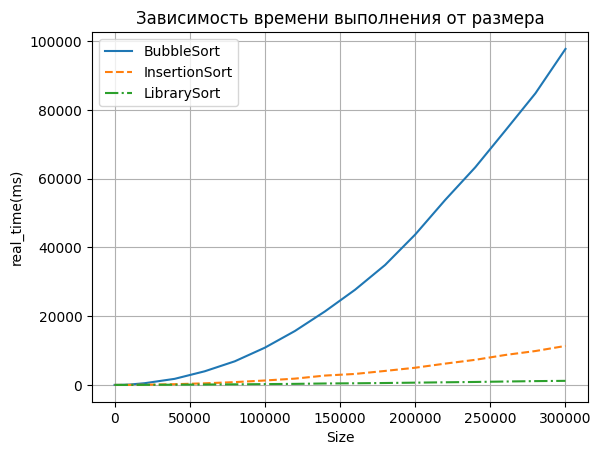


Рисунок 3 Скорость работы LibrarySort на фоне логарифмических сортировок

* 1. Сортировка подсчетом

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

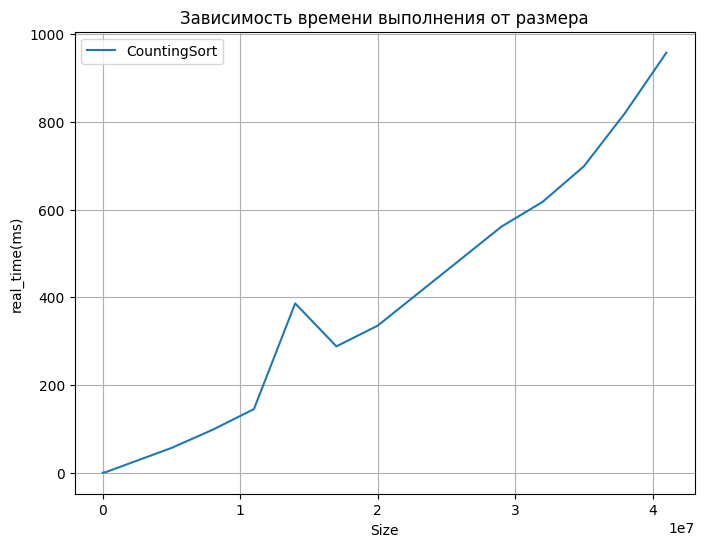


Рисунок 4 Зависимость времени выполнения сортировки подсчетом от времени в случае 1% уникальных элементов

Время работы алгоритма демонстрирует хорошую производительность, что видно на графике, особенно на рисунке вместе с логарифмическими сортировками.

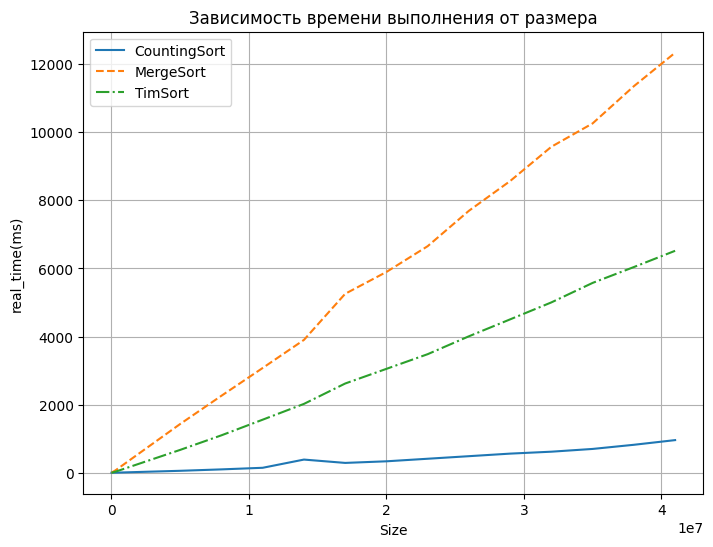


Рисунок 5 Зависимость времени выполнения от размера логарифмических сортировок и сортировки подсчетом

* 1. Логарифмические сортировки

Проанализируем средний случай (случайный массив):

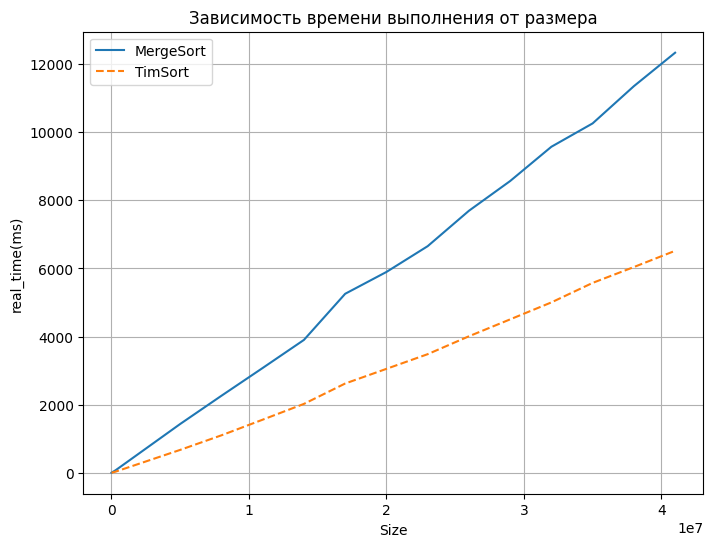
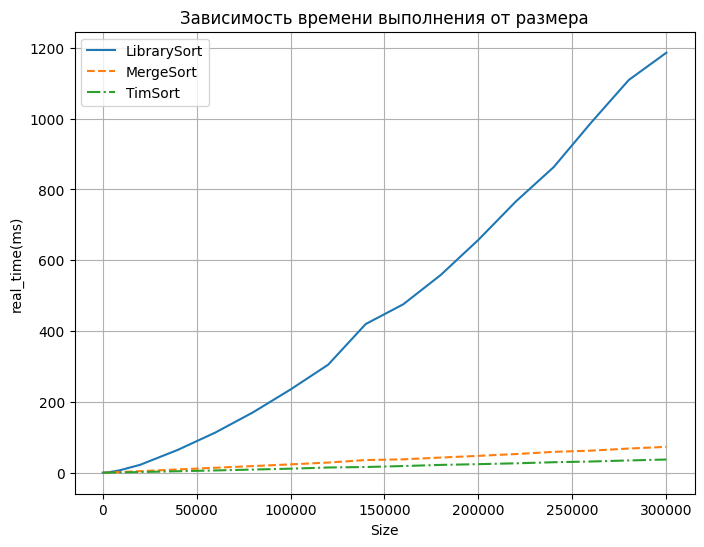


Рисунок 6 Зависимость времени выполнения от размера массива для логарифмических сортировок

График иллюстрирует эффективность TimSort над сортировкой слиянием в среднем случае.

Однако LibrarySort требует по сравнению с ними гораздо больше времени:



* 1. Анализ алгоритмов на малых массивах

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

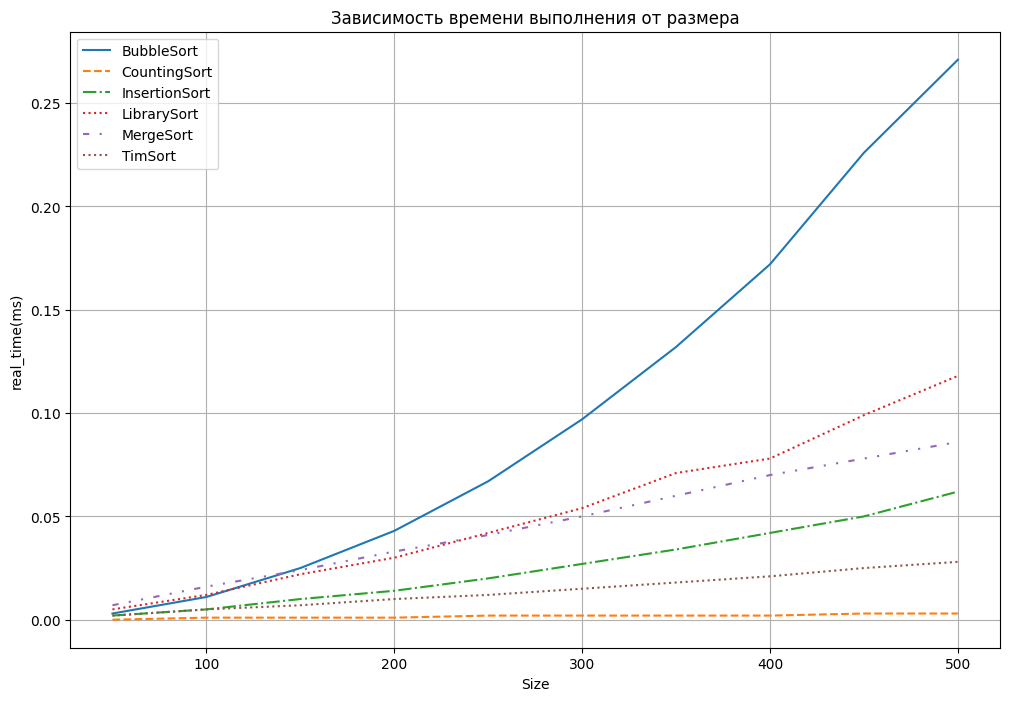


Рисунок 7 Зависимость времени выполнения от размера массива на малых графах

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 этом параграфе исследовались сортировки, основанные на сравнении.

Для подсчета сравнений, перестановок и перемещений внутри реализуемых функций были созданы соответствующие счетчики, значения которых возвращались вместе с отсортированным массивом. Рассмотрим несколько графиков, наглядно в сравнении показывающих особенности алгоритмов

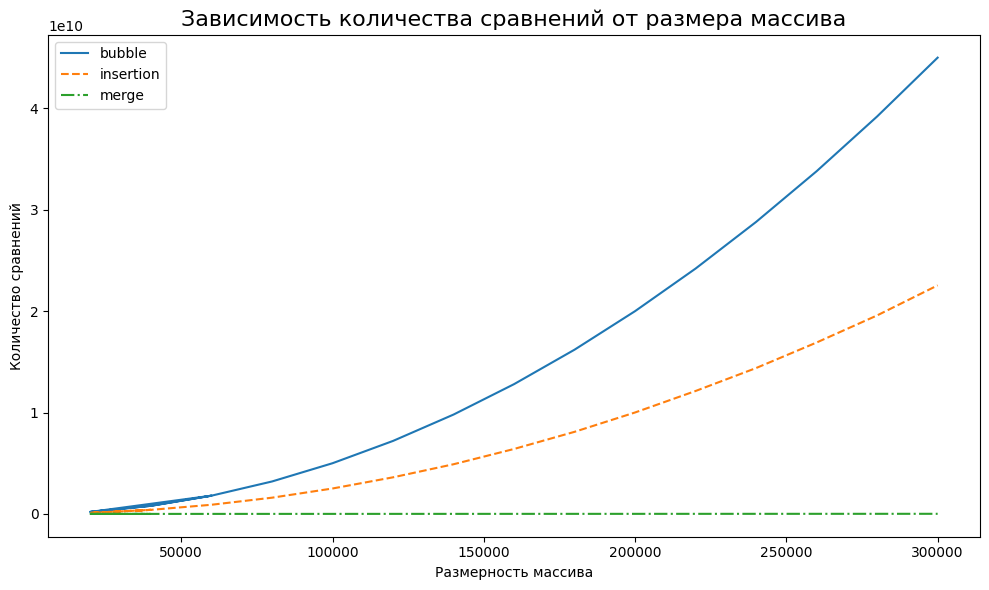


Рисунок 8 Зависимость количества сравнений от размера массива (кроме TimSort)

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

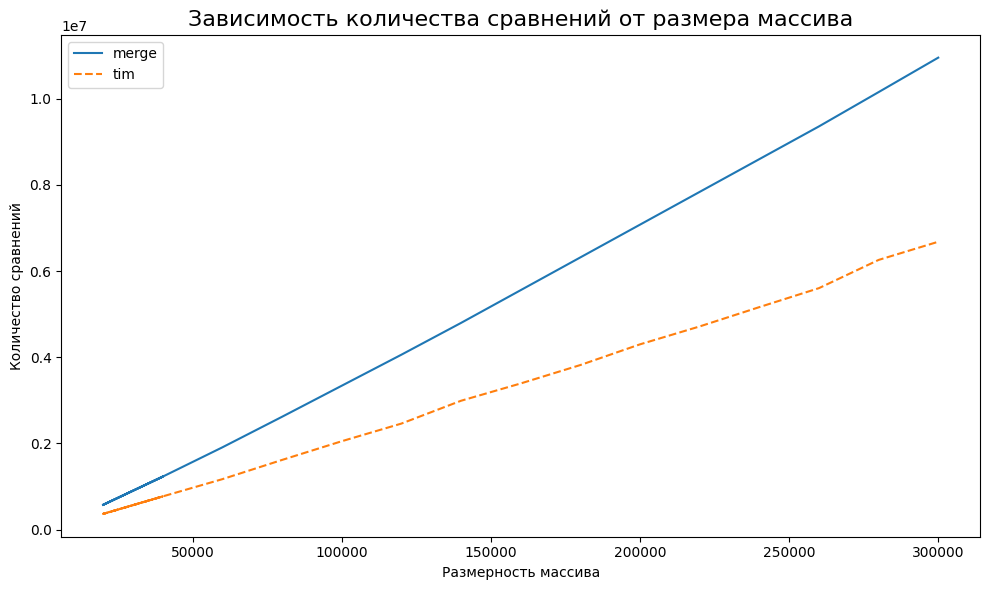


Рисунок 9 Зависимость количества сравнений от размера массива (TimSort и сортировка слиянием)

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.

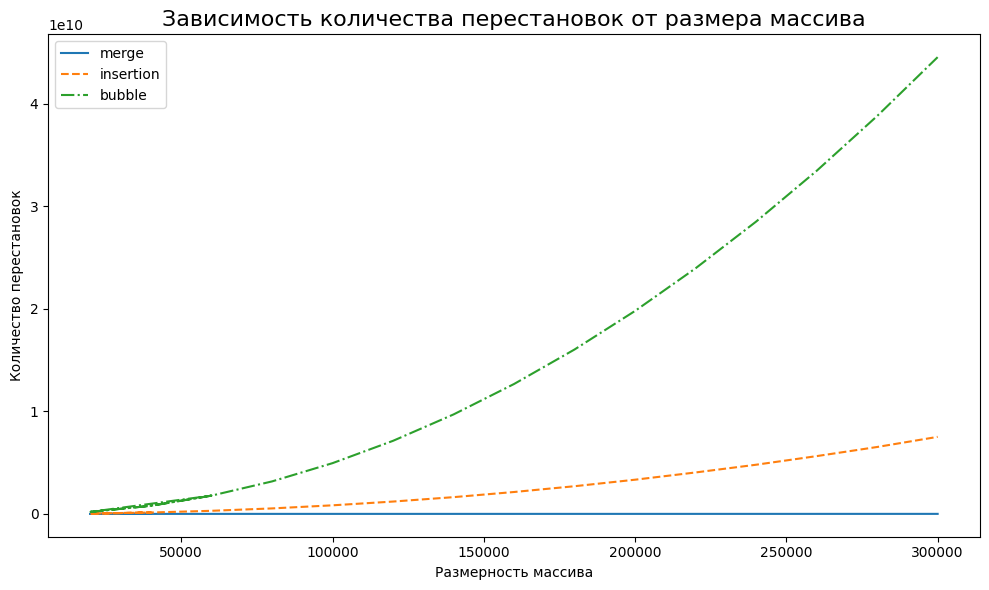


Рисунок 10 Зависимость количества перестановок от размера массива (кроме TimSort)

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

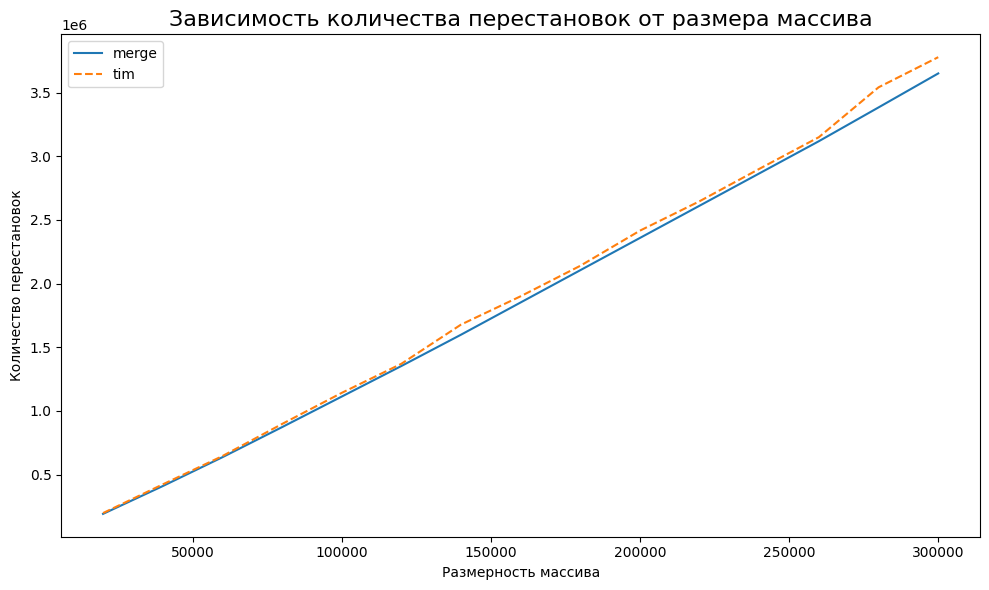


Рисунок ‑ 11 Зависимость количества перестановок от размера массива (TimSort и сортировка слиянием)

TimSort, имея в своей реализации сортировку вставками, имеет почти такое же количество перестановок как у сортировки слиянием. Дело в том, что для поиска места для вставки использовался бинарный поиcка.

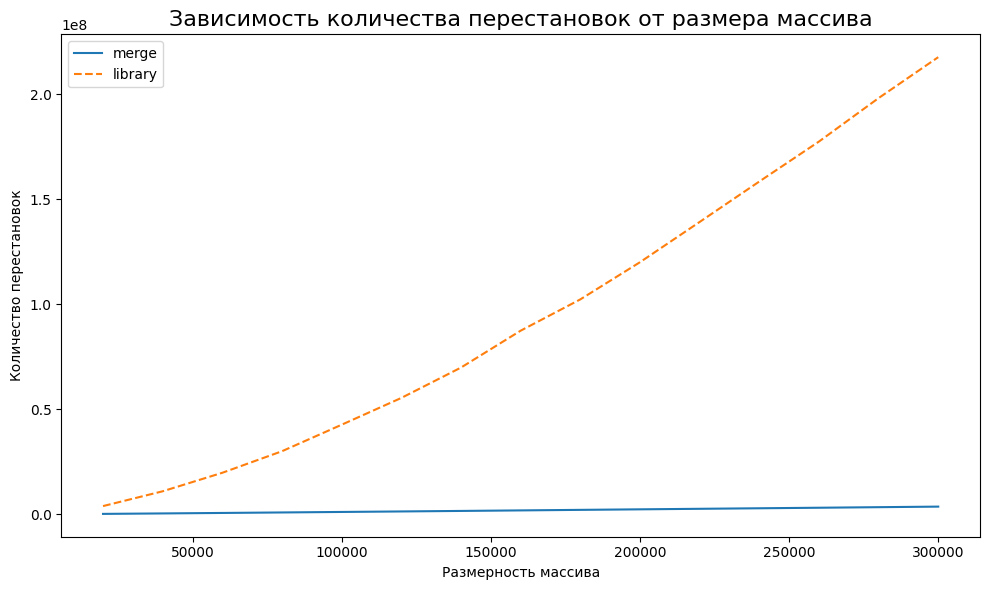


Рисунок 12 Количество перестановок в зависимости от размера массива для LibrarySort в сравнении с сортировкой слиянием.

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 1). Анализ результатов показал, что сортировка слиянием и Timsort являются наиболее эффективными методами для больших массивов данных. Сортировка пузырьком оказалась наименее эффективнее 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.
6. Auger, Nicolas, Cyril Nicaud, and Carine Pivoteau. "Merge strategies: from merge sort to Timsort." (2015).
7. Elkahlout, Ahmad H., and Ashraf YA Maghari. "A comparative study of sorting algorithms comb, cocktail and counting sorting." *International Research Journal of Engineering and Technology (IRJET)* 4.01 (2017).
8. Al-Kharabsheh, Khalid Suleiman, et al. "Review on sorting algorithms a comparative study." *International Journal of Computer Science and Security (IJCSS)* 7.3 (2013): 120-126.

Приложения

Приложение 1

\\ Bubble Sort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0,003 | 0,002 | 344615 | 50 |
| 0,011 | 0,007 | 112000 | 100 |
| 0,025 | 0,012 | 49778 | 150 |
| 0,043 | 0,03 | 32000 | 200 |
| 0,067 | 0,047 | 16186 | 250 |
| 0,097 | 0,056 | 16186 | 300 |
| 0,132 | 0,084 | 7467 | 350 |
| 0,172 | 0,095 | 6400 | 400 |
| 0,226 | 0,113 | 6921 | 450 |
| 0,271 | 0,173 | 4978 | 500 |
| 1,099 | 0,872 | 896 | 1000 |
| 4,3 | 2,847 | 236 | 2000 |
| 9,614 | 6,138 | 112 | 3000 |
| 17,021 | 11,042 | 75 | 4000 |
| 26,665 | 22,485 | 41 | 5000 |
| 38,528 | 23,438 | 32 | 6000 |
| 52,913 | 30,649 | 26 | 7000 |
| 68,52 | 35,59 | 18 | 8000 |
| 86,987 | 50 | 10 | 9000 |
| 107,687 | 59,375 | 10 | 10000 |
| 496,538 | 234,375 | 3 | 20000 |
| 1770,022 | 1031,25 | 3 | 40000 |
| 3967,948 | 1890,625 | 3 | 60000 |
| 6871,655 | 3562,5 | 3 | 80000 |
| 10836,41 | 5546,875 | 3 | 100000 |
| 15689,7 | 9500 | 3 | 120000 |
| 21393,47 | 13843,75 | 3 | 140000 |
| 27690,84 | 18984,38 | 3 | 160000 |
| 34905,43 | 22921,88 | 3 | 180000 |
| 43754,69 | 26046,88 | 3 | 200000 |
| 53848,2 | 32421,88 | 3 | 220000 |
| 63340,33 | 37156,25 | 3 | 240000 |
| 74039,32 | 43156,25 | 3 | 260000 |
| 84861,19 | 52546,88 | 3 | 280000 |
| 97781,84 | 61484,38 | 3 | 300000 |

\\ CoutingSort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0 | 0 | 2357895 | 50 |
| 0,001 | 0 | 1600000 | 100 |
| 0,001 | 0,001 | 1120000 | 150 |
| 0,001 | 0,001 | 1115023 | 200 |
| 0,002 | 0,001 | 640000 | 250 |
| 0,002 | 0,001 | 746667 | 300 |
| 0,002 | 0,001 | 746667 | 350 |
| 0,002 | 0,001 | 560000 | 400 |
| 0,003 | 0,002 | 497778 | 450 |
| 0,003 | 0,002 | 344615 | 500 |
| 0,006 | 0,005 | 100000 | 1000 |
| 0,012 | 0,007 | 89600 | 2000 |
| 0,017 | 0,013 | 64000 | 3000 |
| 0,023 | 0,016 | 44800 | 4000 |
| 0,029 | 0,022 | 28000 | 5000 |
| 0,035 | 0,025 | 32000 | 6000 |
| 0,04 | 0,027 | 29867 | 7000 |
| 0,046 | 0,026 | 22400 | 8000 |
| 0,052 | 0,028 | 28000 | 9000 |
| 0,057 | 0,036 | 19478 | 10000 |
| 0,119 | 0,066 | 10000 | 20000 |
| 0,242 | 0,126 | 4978 | 40000 |
| 0,373 | 0,181 | 3446 | 60000 |
| 0,48 | 0,271 | 2133 | 80000 |
| 0,597 | 0,353 | 2036 | 100000 |
| 0,733 | 0,401 | 1792 | 120000 |
| 0,928 | 0,377 | 1948 | 140000 |
| 0,964 | 0,837 | 747 | 160000 |
| 1,107 | 0,795 | 1120 | 180000 |
| 1,249 | 0,841 | 929 | 200000 |
| 1,364 | 0,903 | 640 | 220000 |
| 1,558 | 0,75 | 1000 | 240000 |
| 1,632 | 0,837 | 747 | 260000 |
| 1,962 | 1,36 | 448 | 280000 |
| 2,108 | 1,067 | 498 | 300000 |
| 61,619 | 25,67 | 28 | 5000000 |
| 109,824 | 52,557 | 11 | 8000000 |
| 154,31 | 85,938 | 12 | 11000000 |
| 211,046 | 98,958 | 6 | 14000000 |
| 249,183 | 140,625 | 9 | 17000000 |
| 303,315 | 253,906 | 4 | 20000000 |
| 391,517 | 213,542 | 3 | 23000000 |
| 428,184 | 203,125 | 3 | 26000000 |
| 513,8 | 296,875 | 3 | 29000000 |
| 612,463 | 398,438 | 3 | 32000000 |
| 711,286 | 359,375 | 3 | 35000000 |
| 743,648 | 515,625 | 3 | 38000000 |
| 993,56 | 531,25 | 3 | 41000000 |

\\ InsertionSort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0,002 | 0,001 | 647432 | 50 |
| 0,005 | 0,003 | 280000 | 100 |
| 0,01 | 0,005 | 100000 | 150 |
| 0,014 | 0,01 | 89600 | 200 |
| 0,02 | 0,01 | 64000 | 250 |
| 0,027 | 0,016 | 49778 | 300 |
| 0,034 | 0,021 | 28000 | 350 |
| 0,042 | 0,024 | 34462 | 400 |
| 0,05 | 0,034 | 23579 | 450 |
| 0,062 | 0,037 | 17920 | 500 |
| 0,193 | 0,151 | 6400 | 1000 |
| 0,672 | 0,508 | 1600 | 2000 |
| 1,361 | 0,837 | 747 | 3000 |
| 2,316 | 1,831 | 512 | 4000 |
| 3,566 | 2,404 | 299 | 5000 |
| 5,237 | 4,011 | 187 | 6000 |
| 6,822 | 5,243 | 149 | 7000 |
| 8,68 | 4,464 | 112 | 8000 |
| 11,119 | 6,557 | 112 | 9000 |
| 13,393 | 7,639 | 90 | 10000 |
| 52,952 | 33,203 | 24 | 20000 |
| 210,634 | 117,188 | 6 | 40000 |
| 463,204 | 273,438 | 3 | 60000 |
| 819,873 | 593,75 | 3 | 80000 |
| 1288,163 | 812,5 | 3 | 100000 |
| 1831,684 | 1046,875 | 3 | 120000 |
| 2722,964 | 1125 | 3 | 140000 |
| 3195,98 | 2515,625 | 3 | 160000 |
| 4073,531 | 2843,75 | 3 | 180000 |
| 5002,783 | 3828,125 | 3 | 200000 |
| 6203,812 | 3390,625 | 3 | 220000 |
| 7308,974 | 3843,75 | 3 | 240000 |
| 8717,779 | 5250 | 3 | 260000 |
| 9863,965 | 5500 | 3 | 280000 |
| 11349,91 | 6843,75 | 3 | 300000 |

\\ LibrarySort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0,005 | 0,002 | 263529 | 50 |
| 0,012 | 0,006 | 89600 | 100 |
| 0,022 | 0,014 | 64000 | 150 |
| 0,03 | 0,022 | 40727 | 200 |
| 0,042 | 0,022 | 32000 | 250 |
| 0,054 | 0,036 | 24889 | 300 |
| 0,071 | 0,052 | 10000 | 350 |
| 0,078 | 0,047 | 11200 | 400 |
| 0,099 | 0,055 | 15206 | 450 |
| 0,118 | 0,072 | 10240 | 500 |
| 0,287 | 0,195 | 3446 | 1000 |
| 0,794 | 0,614 | 1120 | 2000 |
| 1,439 | 0,984 | 1000 | 3000 |
| 2,1 | 1,507 | 560 | 4000 |
| 2,964 | 2,404 | 299 | 5000 |
| 3,775 | 2,715 | 236 | 6000 |
| 4,858 | 3,081 | 213 | 7000 |
| 5,698 | 3,14 | 209 | 8000 |
| 6,853 | 4,464 | 112 | 9000 |
| 8,132 | 5,625 | 100 | 10000 |
| 22,428 | 11,719 | 64 | 20000 |
| 64,48 | 33,654 | 26 | 40000 |
| 113,573 | 50 | 10 | 60000 |
| 170,457 | 91,146 | 6 | 80000 |
| 235,123 | 136,161 | 7 | 100000 |
| 305,016 | 195,312 | 4 | 120000 |
| 419,936 | 218,75 | 4 | 140000 |
| 475,744 | 328,125 | 3 | 160000 |
| 558,877 | 367,188 | 3 | 180000 |
| 657,929 | 453,125 | 3 | 200000 |
| 766,624 | 546,875 | 3 | 220000 |
| 862,963 | 515,625 | 3 | 240000 |
| 989,1 | 515,625 | 3 | 260000 |
| 1109,263 | 593,75 | 3 | 280000 |
| 1186,27 | 656,25 | 3 | 300000 |

\\ MergeSort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0,007 | 0,004 | 165926 | 50 |
| 0,016 | 0,011 | 56000 | 100 |
| 0,024 | 0,012 | 55139 | 150 |
| 0,033 | 0,02 | 40727 | 200 |
| 0,041 | 0,028 | 26353 | 250 |
| 0,05 | 0,033 | 22400 | 300 |
| 0,06 | 0,039 | 17231 | 350 |
| 0,07 | 0,036 | 20364 | 400 |
| 0,078 | 0,049 | 11200 | 450 |
| 0,086 | 0,047 | 16186 | 500 |
| 0,181 | 0,107 | 4978 | 1000 |
| 0,377 | 0,335 | 2800 | 2000 |
| 0,587 | 0,392 | 1792 | 3000 |
| 0,783 | 0,61 | 896 | 4000 |
| 1,008 | 0,767 | 1120 | 5000 |
| 1,217 | 0,816 | 747 | 6000 |
| 1,411 | 0,931 | 772 | 7000 |
| 1,637 | 0,875 | 1000 | 8000 |
| 1,859 | 1,156 | 1000 | 9000 |
| 2,079 | 1,562 | 640 | 10000 |
| 4,36 | 2,832 | 320 | 20000 |
| 9,18 | 4,424 | 166 | 40000 |
| 14,003 | 6,944 | 90 | 60000 |
| 18,778 | 14,101 | 41 | 80000 |
| 23,582 | 13,438 | 50 | 100000 |
| 28,663 | 17,361 | 45 | 120000 |
| 35,775 | 15,278 | 45 | 140000 |
| 37,784 | 27,344 | 20 | 160000 |
| 42,852 | 27,5 | 25 | 180000 |
| 47,596 | 35,938 | 20 | 200000 |
| 52,769 | 33,053 | 26 | 220000 |
| 58,918 | 29,375 | 25 | 240000 |
| 62,409 | 35,807 | 24 | 260000 |
| 68,306 | 34,375 | 15 | 280000 |
| 73,101 | 40,799 | 18 | 300000 |
| 1432,37 | 671,875 | 3 | 5000000 |
| 2302,906 | 1109,375 | 3 | 8000000 |
| 3305,365 | 1437,5 | 3 | 11000000 |
| 4047,885 | 2125 | 3 | 14000000 |
| 4971,77 | 2609,375 | 3 | 17000000 |
| 5841,569 | 3328,125 | 3 | 20000000 |
| 6893,137 | 3500 | 3 | 23000000 |
| 7646,533 | 4421,875 | 3 | 26000000 |
| 8705,145 | 4578,125 | 3 | 29000000 |
| 9425,341 | 4781,25 | 3 | 32000000 |
| 10432,14 | 5984,375 | 3 | 35000000 |
| 11532,66 | 6640,625 | 3 | 38000000 |
| 12184,65 | 7500 | 3 | 41000000 |

\\ TimSort

|  |  |  |  |
| --- | --- | --- | --- |
| **real\_time(ms)** | **cpu\_time(ms)** | **iterations** | **Size** |
| 0,002 | 0,001 | 640000 | 50 |
| 0,005 | 0,003 | 203636 | 100 |
| 0,007 | 0,003 | 165926 | 150 |
| 0,01 | 0,006 | 112000 | 200 |
| 0,012 | 0,008 | 89600 | 250 |
| 0,015 | 0,009 | 74667 | 300 |
| 0,018 | 0,013 | 74667 | 350 |
| 0,021 | 0,012 | 56000 | 400 |
| 0,025 | 0,012 | 49778 | 450 |
| 0,028 | 0,015 | 56000 | 500 |
| 0,064 | 0,055 | 10000 | 1000 |
| 0,15 | 0,092 | 7467 | 2000 |
| 0,233 | 0,163 | 4614 | 3000 |
| 0,32 | 0,262 | 2800 | 4000 |
| 0,414 | 0,273 | 1948 | 5000 |
| 0,504 | 0,333 | 2489 | 6000 |
| 0,606 | 0,352 | 1600 | 7000 |
| 0,691 | 0,42 | 1600 | 8000 |
| 0,804 | 0,53 | 1445 | 9000 |
| 0,893 | 0,61 | 896 | 10000 |
| 1,916 | 1,283 | 560 | 20000 |
| 4,134 | 2,352 | 299 | 40000 |
| 6,471 | 4,327 | 195 | 60000 |
| 8,968 | 5 | 100 | 80000 |
| 11,388 | 5,156 | 100 | 100000 |
| 14,768 | 5,781 | 100 | 120000 |
| 16,12 | 13,889 | 45 | 140000 |
| 18,872 | 11,875 | 75 | 160000 |
| 22,098 | 10,324 | 56 | 180000 |
| 24,212 | 15,278 | 45 | 200000 |
| 26,583 | 15,625 | 32 | 220000 |
| 29,634 | 18,555 | 64 | 240000 |
| 31,704 | 15,625 | 50 | 260000 |
| 34,671 | 19,531 | 32 | 280000 |
| 37,277 | 22,917 | 45 | 300000 |
| 788,942 | 304,688 | 3 | 5000000 |
| 1268,439 | 578,125 | 3 | 8000000 |
| 1823,59 | 843,75 | 3 | 11000000 |
| 2404,509 | 1250 | 3 | 14000000 |
| 3008,143 | 1406,25 | 3 | 17000000 |
| 3459,959 | 1812,5 | 3 | 20000000 |
| 4051,785 | 1671,875 | 3 | 23000000 |
| 4406,369 | 2703,125 | 3 | 26000000 |
| 5036,293 | 2859,375 | 3 | 29000000 |
| 5662,809 | 2546,875 | 3 | 32000000 |
| 6154,978 | 3890,625 | 3 | 35000000 |
| 6826,062 | 3750 | 3 | 38000000 |
| 7293,602 | 3984,375 | 3 | 41000000 |

Приложение 2

\\ BubbleSort

#pragma once

#include <vector>

#include <functional>

#include <tuple>

/// <summary>

///

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="arr"></param>

/// <param name="comp"></param>

/// <returns>tuple of soreted array, count comapare and count swap</returns>

template<typename T>

std::tuple<std::vector<T>, long long, long long> bubbleSort(std::vector<T> arr, std::function<bool(const T&, const T&)> comp = std::less<T>()) {

int n = arr.size();

bool swapped;

long long count\_swap = 0;

long long count\_compare = 0;

for (int i = 0; i < n - 1; i++) {

swapped = false;

for (int j = 0; j < n - i - 1; j++) {

count\_compare++;

if (!comp(arr[j], arr[j + 1])) {

count\_swap++;

std::swap(arr[j], arr[j + 1]);

swapped = true;

}

}

// Если внутренний цикл не произвел обмена, выходим

if (!swapped)

break;

}

return { arr, count\_compare, count\_swap };

}

\\ Counting Sort

#pragma once

#include <vector>

#include <functional>

using namespace std;

template <typename T>

vector<T> countingSort(vector<T>& inputArray, std::function<int (const T&)> intExtractor)

{

int N = inputArray.size();

// Finding the maximum element of array inputArray[].

int M = 0;

for (int i = 0; i < N; i++)

M = max(M, intExtractor(inputArray[i]));

// Initializing countArray[] with 0

vector<T> countArray(M + 1, 0);

// Mapping each element of inputArray[] as an index

// of countArray[] array

for (int i = 0; i < N; i++)

countArray[intExtractor(inputArray[i])]++;

// Calculating prefix sum at every index

// of array countArray[]

for (int i = 1; i <= M; i++)

countArray[i] += countArray[i - 1];

// Creating outputArray[] from countArray[] array

vector<T> outputArray(N);

for (int i = N - 1; i >= 0; i--)

{

outputArray[countArray[intExtractor(inputArray[i])] - 1] = inputArray[i];

countArray[intExtractor(inputArray[i])]--;

}

return outputArray;

}

\\ Insertion Sort

#pragma once

#include <vector>

#include <functional>

#include <tuple>

/// <summary>

///

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="list"></param>

/// <param name="comp"></param>

/// <returns>tuple of sorted vector, count compares, count swaps, count moves</returns>

template<typename T>

std::tuple<std::vector<T>, long long, long long, long long>

insertionSort(std::vector<T> list, std::function<bool(const T&, const T&)> comp = std::less<T>()) {

long long countCompare = 0;

long long countSwap = 0;

long long countMove = 0;

// Iterate through the list

for (unsigned int i = 1; i < list.size(); i++) {

T elem = list[i];

auto it = std::lower\_bound(list.begin(), list.begin() + i, elem, comp);

size\_t j = std::distance(list.begin(), it);

countCompare += static\_cast<long long>(log2(i)) + 1;

if (j < i) {

T temp = list[i];

for (size\_t k = i; k > j; --k) {

list[k] = list[k - 1];

countMove++;

}

countSwap++;

list[j] = temp;

}

}

return { list, countCompare, countSwap, countMove };

}

\\ LibrarySort

#pragma once

#include <vector>

#include <algorithm>

#include <cmath>

#include <tuple>

#include <functional>

double log2(int x) {

return log(x) / log(2);

}

template <typename T>

std::tuple<std::vector<T>, long long, long long> librarySort(std::vector<T>& vec,

std::function<bool(const T&, const T&)> comp = std::less<T>()) {

if (vec.size() <= 1) {

return { vec, 0, 0 };

}

long long countComapare = 0, countMove = 0;

auto index = vec;

int n = index.size();

std::vector<std::vector<T>> library(2, std::vector<T>(n));

std::vector<T> gaps(n + 1, false);

std::vector<bool> numbered(n + 1, false);

int lib\_size = 1;

int index\_pos = 1;

bool target\_lib = 0;

library[target\_lib][0]= index[0];

int index\_last\_el\_in\_gaps = 0;

while (index\_pos < n) {

// binary search

countComapare += static\_cast<long long>(log2(lib\_size)) + 1;

int insert = std::distance(

std::begin(library[target\_lib]),

std::lower\_bound(std::begin(library[target\_lib]),

std::next(std::begin(library[target\_lib]), lib\_size), index[index\_pos], comp));

// if there is no gap to insert a new index ...

if (numbered[insert] == true) {

int prov\_size = 0, next\_target\_lib = !target\_lib;

// update library and clear gaps

for (int i = 0; i <= std::max(index\_last\_el\_in\_gaps, lib\_size); i++) {

if (numbered[i] == true) {

countMove++;

library[next\_target\_lib][prov\_size] = gaps[i];

prov\_size++;

numbered[i] = false;

}

if (i <= lib\_size) {

countMove++;

library[next\_target\_lib][prov\_size] =

library[target\_lib][i];

prov\_size++;

}

}

target\_lib = next\_target\_lib;

lib\_size = prov\_size - 1;

}

else {

countMove++;

numbered[insert] = true;

gaps[insert] = index[index\_pos];

index\_last\_el\_in\_gaps = insert;

index\_pos++;

}

}

int index\_pos\_for\_output = 0;

for (int i = 0; index\_pos\_for\_output < n; i++) {

if (numbered[i] == true) {

countMove++;

// std::cout << gaps[i] << std::endl;

index[index\_pos\_for\_output] = gaps[i];

index\_pos\_for\_output++;

}

if (i < lib\_size) {

countMove++;

// std::cout << library[target\_lib][i] << std::endl;

index[index\_pos\_for\_output] = library[target\_lib][i];

index\_pos\_for\_output++;

}

}

return { index, countComapare, countMove };

}

\\ Merge Sort

#pragma once

#include <vector>

#include <functional>

template<typename T>

void merge(std::vector<T>& list, int lo, int mi, int hi, std::function<bool(const T&, const T&)> comp,

long long& countCompare, long long& countMove) {

int n1 = mi - lo + 1; // размер первого подмассива

int n2 = hi - mi; // размер второго подмассива

// Создаем временные векторы для хранения левого и правого подмассивов

std::vector<T> left(n1);

std::vector<T> right(n2);

countMove += (n1 + n2);

// Копируем данные во временные векторы

for (int i = 0; i < n1; i++)

left[i] = list[lo + i];

for (int j = 0; j < n2; j++)

right[j] = list[mi + 1 + j];

// Индексы для обхода временных векторов

int i = 0, j = 0;

// Индекс для обхода исходного вектора

int k = lo;

countMove += (n1 + n2);

// Объединяем временные векторы обратно в исходный вектор

while (i < n1 && j < n2) {

countCompare++;

if (comp(left[i], right[j])) {

list[k] = left[i];

i++;

}

else {

list[k] = right[j];

j++;

}

k++;

}

// Копируем оставшиеся элементы из left[], если такие есть

while (i < n1) {

list[k] = left[i];

i++;

k++;

}

// Копируем оставшиеся элементы из right[], если такие есть

while (j < n2) {

list[k] = right[j];

j++;

k++;

}

}

template<typename T>

void \_mergeSort(std::vector<T>& list, int lo, int hi, std::function<bool(const T&, const T&)> comp,

long long& countCompare, long long& countMove) {

if (lo < hi) {

int mi = lo + (hi - lo) / 2;

\_mergeSort(list, lo, mi, comp, countCompare, countMove);

\_mergeSort(list, mi + 1, hi, comp, countCompare, countMove);

merge(list, lo, mi, hi, comp, countCompare, countMove);

}

}

/// <summary>

///

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="list"></param>

/// <param name="comp"></param>

/// <returns>tuple of soreted list, count comapare and count move</returns>

template<typename T>

std::tuple<std::vector<T>, long long, long long>

mergeSort(std::vector<T> list, std::function<bool(const T&, const T&)> comp = std::less<T>()) {

long long countCompare = 0;

long long countMove = 0;

\_mergeSort(list, 0, list.size() - 1, comp, countCompare, countMove);

return { list, countCompare, countMove };

}

\\ TimSort

#pragma once

#include <vector>

#include <algorithm>

#include <tuple>

#include <functional>

// Функция сортировки вставками для части вектора

template <typename T>

void insertionSortTim(std::vector<T>& arr, int left, int right,

std::function<bool(const T&, const T&)> comp,

long long& comparisons, long long& swaps, long long& moves) {

for (int i = left + 1; i <= right; ++i) {

T elem = arr[i];

auto it = std::lower\_bound(arr.begin() + left, arr.begin() + i, elem, comp);

size\_t j = std::distance(arr.begin(), it);

comparisons += static\_cast<long long>(log2(i)) + 1;

if (j < i) {

T temp = arr[i];

for (size\_t k = i; k > j; --k) {

arr[k] = arr[k - 1];

moves++;

}

swaps++;

arr[j] = temp;

}

}

}

// Функция слияния двух отсортированных частей вектора

template <typename T>

void mergeTim(std::vector<T>& arr, int l, int m, int r,

std::function<bool(const T&, const T&)> comp, long long& comparisons, long long& moves) {

int len1 = m - l + 1;

int len2 = r - m;

std::vector<T> left(len1);

std::vector<T> right(len2);

moves += (len1 + len2);

for (int i = 0; i < len1; ++i) {

left[i] = arr[l + i];

}

for (int i = 0; i < len2; ++i) {

right[i] = arr[m + 1 + i];

}

int i = 0;

int j = 0;

int k = l;

moves += (len1 + len2);

while (i < len1 && j < len2) {

++comparisons;

if (comp(left[i], right[j])) {

arr[k] = left[i];

++i;

}

else {

arr[k] = right[j];

++j;

}

++k;

}

while (i < len1) {

arr[k] = left[i];

++k;

++i;

}

while (j < len2) {

arr[k] = right[j];

++k;

++j;

}

}

/// <summary>

///

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="arr"></param>

/// <param name="comp"></param>

/// <returns>tuple of soreted list, count comapares, count swaps and count moves</returns>

template <typename T>

std::tuple<std::vector<T>, long long, long long, long long>

timSort(std::vector<T> arr, std::function<bool(const T&, const T&)> comp = std::less<T>()) {

const int RUN = 32;

long long comparisons = 0;

long long swaps = 0;

long long moves = 0;

int n = arr.size();

// Сортировка подмассивов размером RUN

for (int i = 0; i < n; i += RUN) {

insertionSortTim(arr, i, std::min(i + RUN - 1, n - 1), comp, comparisons, swaps, moves);

}

// Слияние отсортированных подмассивов

for (int size = RUN; size < n; size \*= 2) {

for (int left = 0; left < n; left += 2 \* size) {

int mid = left + size - 1;

int right = std::min(left + 2 \* size - 1, n - 1);

if (mid < right) {

mergeTim(arr, left, mid, right, comp, comparisons, moves);

}

}

}

return { arr, comparisons, swaps, moves };

}

Приложение 3

#include <Benchmark/benchmark.h>

#include <memory>

#include <iostream>

#include <string>

#include <format>

#include <stdexcept>

#include "../SortsAlgoriphm/BubbleSort.h"

#include "../SortsAlgoriphm/InsertionSort.h"

#include "../SortsAlgoriphm/MergeSort.h"

#include "../SortsAlgoriphm/SupportFunction.h"

#include "../SortsAlgoriphm/CountingSort.h"

#include "../SortsAlgoriphm/TimSort.h"

#include "../SortsAlgoriphm/LibrarySort.h"

using namespace std;

#define STRINGIFY(x) #x

#define EXPAND(x) STRINGIFY(x)

string solutionDir;

string getSolutionDir() {

if (!solutionDir.empty())

return solutionDir;

string s = EXPAND(SOLDIR);

s.erase(0, 1); // erase the first quote

s.erase(s.size() - 2); // erase the last quote and the dot

size\_t found = s.find\_last\_of('\\');

if (found != std::string::npos) {

s = s.substr(0, found);

}

solutionDir = s;

return solutionDir;

}

static void BM\_MergeSort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = mergeSort(array);

benchmark::DoNotOptimize(arr);

}

}

static void BM\_InsertionSort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = insertionSort(array);

benchmark::DoNotOptimize(arr);

}

}

static void BM\_BubbleSort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = bubbleSort(array);

benchmark::DoNotOptimize(arr);

}

}

static void BM\_CountingSort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = countingSort<int>(array, [](auto p) { return p; });

benchmark::DoNotOptimize(arr);

}

}

static void BM\_TimSort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = timSort<int>(array);

benchmark::DoNotOptimize(arr);

}

}

static void BM\_LibrarySort(benchmark::State& state, const std::string& arrayFileName) {

auto fullPath = getSolutionDir() + "\\" + arrayFileName;

std::vector<int> array = readIntegersFromFile(fullPath);

for (auto \_ : state) {

auto arr = librarySort<int>(array);

benchmark::DoNotOptimize(arr);

}

}

int main(int argc, char\*\* argv) {

for (int i = 5'000'000; i <= 41'000'000; i += 3'000'000) {

auto intUniformFileName = std::format("arrays\\int\_uniform\_{}.txt", i);

auto intUniformName = std::format("int\_uniform\_{}.txt", i);

//benchmark::RegisterBenchmark("BM\_LibrarySort/" + intUniformName, BM\_LibrarySort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_MergeSort/" + intUniformName, BM\_MergeSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_TimSort/" + intUniformName, BM\_TimSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_CountingSort/" + intUniformName, BM\_CountingSort, intUniformFileName)->Unit(benchmark::kMillisecond);

}

for (int i = 20'000; i <= 300'000; i += 20'000) {

auto intUniformFileName = std::format("arrays\\int\_uniform\_{}.txt", i);

auto intUniformName = std::format("int\_uniform\_{}.txt", i);

benchmark::RegisterBenchmark("BM\_LibrarySort/" + intUniformName, BM\_LibrarySort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_MergeSort/" + intUniformName, BM\_MergeSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_CountingSort/" + intUniformName, BM\_CountingSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_InsertionSort/" + intUniformName, BM\_InsertionSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_BubbleSort/" + intUniformName, BM\_BubbleSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_TimSort/" + intUniformName, BM\_TimSort, intUniformFileName)->Unit(benchmark::kMillisecond);

}

for (int i = 1'000; i <= 10'000; i += 1'000) {

auto intUniformFileName = std::format("arrays\\int\_uniform\_{}.txt", i);

auto intUniformName = std::format("int\_uniform\_{}.txt", i);

benchmark::RegisterBenchmark("BM\_LibrarySort/" + intUniformName, BM\_LibrarySort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_MergeSort/" + intUniformName, BM\_MergeSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_CountingSort/" + intUniformName, BM\_CountingSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_InsertionSort/" + intUniformName, BM\_InsertionSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_BubbleSort/" + intUniformName, BM\_BubbleSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_TimSort/" + intUniformName, BM\_TimSort, intUniformFileName)->Unit(benchmark::kMillisecond);

}

for (int i = 50; i <= 500; i += 50) {

auto intUniformFileName = std::format("arrays\\int\_uniform\_{}.txt", i);

auto intUniformName = std::format("int\_uniform\_{}.txt", i);

benchmark::RegisterBenchmark("BM\_LibrarySort/" + intUniformName, BM\_LibrarySort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_MergeSort/" + intUniformName, BM\_MergeSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_CountingSort/" + intUniformName, BM\_CountingSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_InsertionSort/" + intUniformName, BM\_InsertionSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_BubbleSort/" + intUniformName, BM\_BubbleSort, intUniformFileName)->Unit(benchmark::kMillisecond);

benchmark::RegisterBenchmark("BM\_TimSort/" + intUniformName, BM\_TimSort, intUniformFileName)->Unit(benchmark::kMillisecond);

}

benchmark::Initialize(&argc, argv);

benchmark::RunSpecifiedBenchmarks();

return 0;

}