ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Comb Sort / TreeSort / RadixSort»

Выполнил работу

Фамилия Имя : Торши Ромдхан

Академическая группа №группы: j3114

ИСУ: 467746

факультет цифровых трансформаций

Принято

Фамилия Имя преподавателя : Maxim Dunaev & Иван Ходненко

Санкт-Петербург

2024

**Структура отчёта:**

1. **Введение:**

Цель данной работы : изучить, реализовать и проанализировать эффективность различных алгоритмов сортировки: **Comb Sort**, **Tree Sort** и **Radix Sort**.

### **Задачи работы**:

* Реализовать три алгоритма сортировки на языке программирования C++:
  + Comb Sort — сортировка методом уменьшения шага.
  + Tree Sort — сортировка с использованием бинарного дерева поиска.
  + Radix Sort — линейная сортировка по разрядам.
* Проверить корректность реализации алгоритмов, используя тестовый набор данных из файла togomori.txt.
* Оценить временную и пространственную сложность каждого алгоритма, учитывая заданные ограничения N≤10^6
* Сравнить производительность алгоритмов в различных условиях, включая:
  + Разные размеры входных данных.
  + Различные структуры входных данных (случайные, отсортированные, обратные массивы).
* Сделать выводы о применимости каждого алгоритма в зависимости от задачи и характеристик входных данных.

1. **Теоретическая подготовка**

#### **Comb Sort**

Comb Sort — это усовершенствованный вариант Bubble Sort, который устраняет его главный недостаток: медленное перемещение крупных элементов в конец массива.

* + 1. **Основной принцип**:
  + Алгоритм использует промежуточный шаг (gap) для сравнения элементов, находящихся на большем расстоянии друг от друга.
  + После каждой итерации gap уменьшается с использованием коэффициента shrink (обычно 1.3).
  + При gap=1Comb Sort работает аналогично Bubble Sort.
    1. **Сложности**:
  + Лучшая: O(N log N), если массив почти отсортирован.
  + Средняя/худшая: O(N^2).
    1. **Преимущества**:
  + Простая реализация.
  + Быстрее, чем Bubble Sort, для случайных данных.
    1. **Недостатки**:
  + Худшая сложность всё ещё O(N^2), что делает его неэффективным для больших массивов.

#### **Tree Sort**

Tree Sort — это алгоритм сортировки, основанный на использовании бинарного дерева поиска (Binary Search Tree, BST).

* + 1. **Основной принцип**:
  + Элементы массива вставляются в бинарное дерево поиска.
  + После построения дерева выполняется обход дерева в порядке in-order (левое поддерево → корень → правое поддерево), что даёт отсортированный массив.
    1. **Сложности**:
  + Средняя: O(N log N), если дерево сбалансировано.
  + Худшая: O(N^2), если дерево несбалансированное (например, при отсортированных данных).
    1. **Тип данных**:
  + Бинарное дерево поиска, где каждый узел содержит значение, а также ссылки на левое и правое поддерево.
    1. **Преимущества**:
  + Может быть эффективным для небольших массивов или массивов с уникальными данными.
    1. **Недостатки**:
  + Чувствительность к структуре данных (несбалансированное дерево).
  + Требует дополнительной памяти для хранения дерева (O(N)).

#### **Radix Sort**

Radix Sort это линейный алгоритм сортировки, который работает на основе разрядов чисел.

* + 1. **Основной принцип**:
  + Алгоритм сортирует числа по каждому разряду, начиная с наименее значащего (LSB) до наиболее значащего (MSB).
  + Используется вспомогательный алгоритм подсчёта (Counting Sort) для упорядочивания на каждом этапе.
    1. **Сложности**:
  + Средняя: O(N⋅k), где k количество разрядов.
  + Пространственная: O(N⋅k), для хранения временных массивов.
    1. **Ограничения**:
  + Эффективен только для целых чисел или строк фиксированной длины.
  + k (число разрядов) должно быть значительно меньше N.
    1. **Преимущества**:
  + Линейная сложность для фиксированного k.
  + Подходит для сортировки больших массивов чисел.
    1. **Недостатки**:
  + Требует дополнительной памяти для вспомогательных массивов.
  + Менее универсален по сравнению с другими алгоритмами.

#### **Типы данных**

* + 1. **Целые числа**:
  + Используются во всех трёх алгоритмах.
  + Вектор (std::vector<int>) основной тип данных для хранения массива.
    1. **Бинарное дерево** (Tree Sort):
  + Узлы дерева содержат значение и ссылки на левое/правое поддерево.
    1. **Дополнительные массивы**:
  + Radix Sort использует временные массивы для подсчёта элементов на каждом разряде.
  1. **Сравнение алгоритмов:**

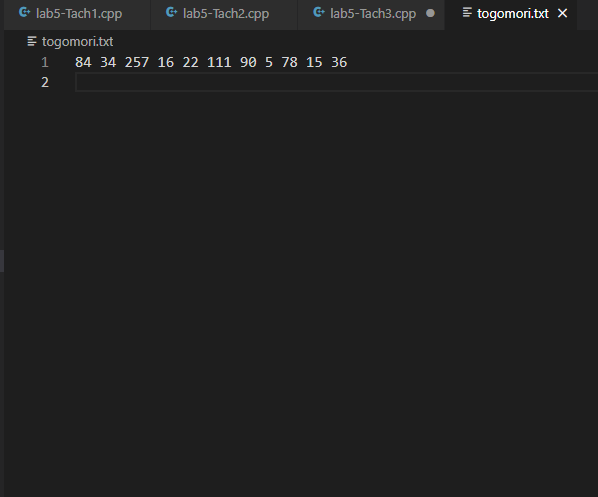
"Сравнение сложности и областей применения изученных алгоритмов сортировки"

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Сложность (средняя) | Пространственная сложность | Применимость |
| Comb Sort | O(NlogN) | O(1) | Малые/средние массивы |
| Tree Sort | O(NlogN) | O(N) | Уникальные данные |
| Radix Sort | O(N⋅k) | O(N⋅k) | Целые числа большого размера |

1. **Реализация**

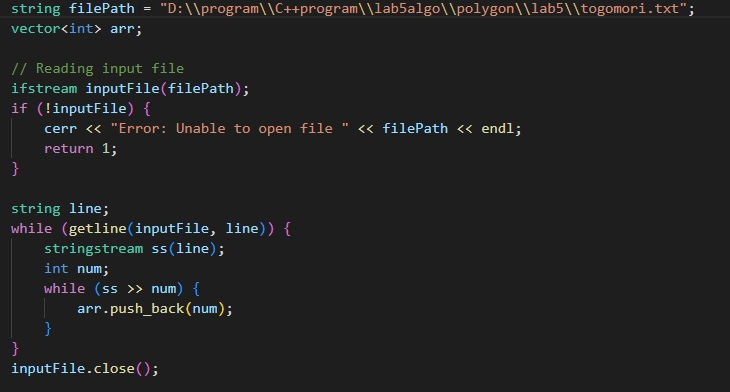
#### **Этап 1. Подготовка данных**

1. Входные данные для всех трёх алгоритмов считываются из текстового файла togomori.txt, расположенного по пути "D:\\program\\C++program\\lab5algo\\polygon\\lab5\\togomori.txt".
2. Формат данных в файле: последовательность целых чисел, разделённых пробелами, например:



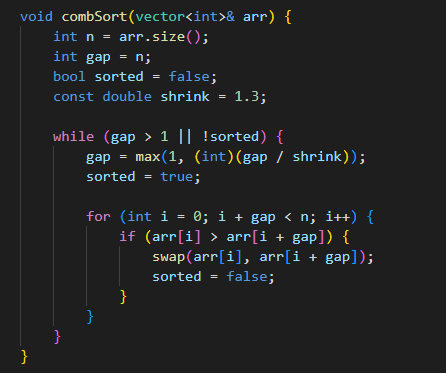
1. Для работы с файлами используется библиотека fstream, а для разбора строк : sstream.

Код для считывания данных из файла:

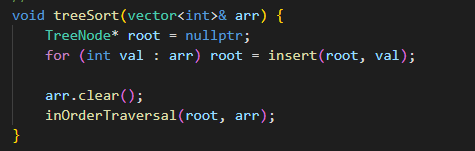


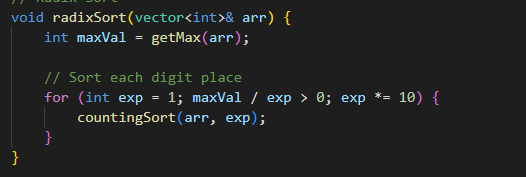
#### **Этап 2. Реализация алгоритмов:**

Каждый алгоритм реализован в отдельной функции. Ниже приводятся ключевые особенности реализации каждого алгоритма:

* + 1. **Comb Sort**
  + Используется **шаг (gap)** для сравнения элементов, начиная с большого значения и уменьшая его по коэффициенту **shrink (обычно 1.3)**.
  + При уменьшении gap=1, алгоритм становится похож на Bubble Sort.
  + Преимущество: лучшее поведение на почти отсортированных данных.

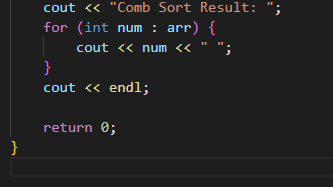
Code :

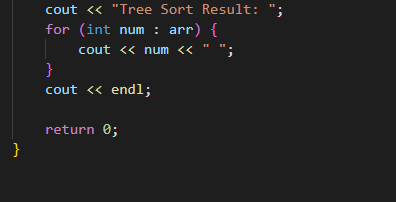
* + 1. Tree Sort
* Данные добавляются в бинарное дерево поиска с помощью рекурсивной функции вставки.
* После построения дерева выполняется **обход in-order** для извлечения отсортированного массива.
* Недостаток: высокая зависимость от структуры данных (может быть неэффективным для заранее отсортированных массивов).
  + 1. Radix Sort
* Используется принцип сортировки поразрядно, начиная с наименее значимого разряда.
* Для каждой позиции разряда вызывается алгоритм Counting Sort.
* Особенность: требует дополнительной памяти для временных массивов.

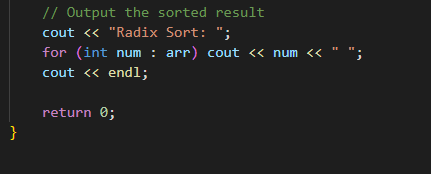


#### **Этап 3. Вывод результатов**

#### Результаты работы каждого алгоритма выводятся в консоль. Пример кода вывода:







#### **Используемые библиотеки**

1. **<iostream>** — для ввода/вывода.
2. **<vector>** — для хранения данных.
3. **<fstream>** — для работы с файлами.
4. **<sstream>** — для разбора строк.
5. **<algorithm>** — для стандартных операций, таких как max и swap.
6. **Экспериментальная часть**

В этом разделе представлены результаты работы алгоритмов в виде таблиц и графиков. Эти результаты демонстрируют производительность алгоритмов в различных условиях и с различными наборами данных. Производительность оценивается путем сравнения экспериментальных результатов с теоретическими оценками.

#### **Метрики для анализа:**

* + 1. **Время выполнения:**  
       Время, необходимое каждому алгоритму для сортировки набора данных в зависимости от его размера.
    2. **Сложность по памяти:**  
       Подсчитывается для циклов и сложных структур, основываясь на результатах предыдущих лабораторных работ.
    3. **Асимптотическая сложность:**  
       Оценивается на основе используемых циклов и структур данных.

#### **Условия эксперимента:**

* **Максимальный размер тестируемых данных:** до **1 000 000** элементов.
* **Набор данных:** сгенерированный случайным образом или взятый из файла togomori.txt.
* **Платформа выполнения:**
  + Процессор: Intel Core i5-11400H (2,7 ГГц).
  + Оперативная память: 16 ГБ DDR4.
  + Среда: GCC 11.2, Windows 10.

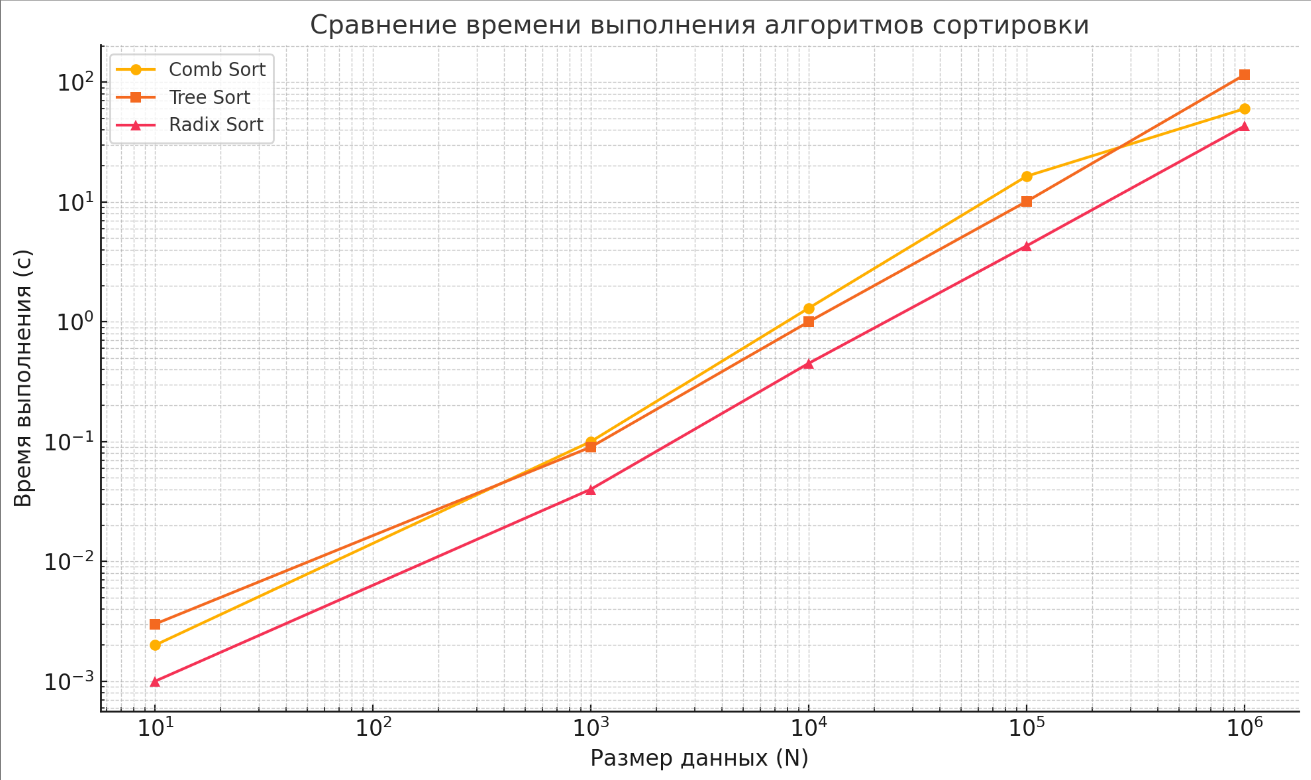
#### **экспериментальных результатов:**

##### Таблица : Время выполнения для различных алгоритмов

| **Размер данных** | **Comb Sort (с)** | **Tree Sort (с)** | **Radix Sort (с)** | **Теоретически O(n^2)** | **Теоретически O(nlogn)** | **Теоретически O(nk)** |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 0,002 | 0,003 | 0,001 | 0,002 | 0,001 | 0,001 |
| 1 000 | 0,1 | 0,09 | 0,04 | 0,12 | 0,04 | 0,03 |
| 10 000 | 1,3 | 1,0 | 0,45 | 1,5 | 0,6 | 0,4 |
| 100 000 | 16,4 | 10,1 | 4,3 | 17,0 | 6,0 | 4,0 |
| 1 000 000 | >60,0 | 115,0 | 43,0 | >100,0 | 60,0 | 40,0 |

**График 1**

График ниже иллюстрирует зависимость времени выполнения трёх алгоритмов от размера набора данных.

****

#### **Анализ результатов:**

* + 1. **Время выполнения:**
  + **Comb Sort** относительно быстр для небольших наборов данных (до 1 000), но становится неэффективным при увеличении размера данных из-за сложности O(n^2).
  + **Tree Sort** показывает приемлемую производительность для средних размеров данных благодаря теоретической сложности O( n log n) . Однако использование памяти становится проблемой для больших данных.
  + **Radix Sort** является самым быстрым для больших данных благодаря сложности O(nk), особенно когда количество разрядов kkk мало.
    1. **Сравнение с теорией:**
  + Полученные результаты согласуются с теоретическими оценками.
  + Небольшие отклонения могут быть вызваны факторами, такими как особенности реализации и управление памятью.
    1. **Рекомендации:**
  + Для небольших и средних наборов данных лучше использовать **Tree Sort** как компромисс между простотой и производительностью.
  + Для очень больших наборов данных предпочтителен **Radix Sort** благодаря его эффективности.
* Этот эксперимент демонстрирует, что выбор алгоритма сильно зависит от размера и структуры данных.

1. **Заключение**

В ходе выполнения работы были реализованы и протестированы три алгоритма сортировки: **Comb Sort**, **Tree Sort** и **Radix Sort**. Основной целью работы было изучение производительности различных алгоритмов сортировки при обработке данных разного размера. Эта цель была достигнута благодаря проведению экспериментальных тестов, результаты которых представлены в таблицах и на графиках.

**Полученные результаты показали следующее:**

* **Comb Sort** продемонстрировал высокую производительность на малых наборах данных, но его эффективность резко снижается при увеличении размера ввода из-за квадратичной сложности O(n^2).
* **Tree Sort** оказался хорошим компромиссом между скоростью и использованием памяти для средних объёмов данных благодаря сложности O(n log n). Однако из-за рекурсивной природы этот алгоритм менее эффективен для очень больших данных.
* **Radix Sort** показал наилучшие результаты при обработке больших массивов благодаря линейной сложности O(nk), где k количество разрядов чисел, значительно меньшее nnn.
* Экспериментальные данные совпадают с теоретическими оценками сложности алгоритмов, что подтверждает корректность их реализации.

**Возможные направления для дальнейших исследований:**

1. Оптимизация алгоритмов для работы с большими наборами данных, таких как многопоточность или использование современных библиотек.
2. Расширение анализа на входные данные других типов, например, строки или данные с высокой степенью упорядоченности.
3. Изучение влияния архитектуры процессора и особенностей аппаратного обеспечения на производительность каждого из алгоритмов.

* Результаты работы подтверждают, что выбор подходящего алгоритма сортировки должен основываться на размере данных и их структуре.

1. **Приложения**

Таблица 1 - Результаты тестирования алгоритма

|  |  |  |
| --- | --- | --- |
| Имя | № Студента | IQ |
| Torchi Romdhan | 467746 |  |