

ВВЕДЕНИЕ

Сортировка кучей сама по себе имеет сложность по времени $O(n \log n)$ вне зависимости от данных. Чтобы из себя не представлял массив, сложность heapsort никогда не деградирует до $O(n^2)$, что может приключиться, к примеру, с быстрой сортировкой. Обратной стороной медали является то, что сортировку бинарной кучей нельзя и ускорить, сложности $O(n)$ ожидать также не приходится (а вот та же быстрая сортировка, при определённых условиях может достигать таких показателей).

Встал вопрос: можно ли сделать так, чтобы временная сложность сортировка кучей, с одной стороны, была не ниже чем $O(n \log n)$, но при благоприятном раскладе (в частности, если обрабатывается почти отсортированный массив) повышалась до $O(n)$?

Чтобы решить этот вопрос, Дейкстра предложил использовать специальные двоичные кучи, построенные на числах Леонардо.

Целью работы является изучение такой структуры данных как «куча леонардовых куч» и реализация на её основе плавной сортировки. Также необходимо провести сравнительное исследование плавной сортировки с поразрядной и сопоставить экспериментальные значения с теоретическими.

1. ПОСТАНОВКА ЗАДАЧИ

Реализация и экспериментальное машинное исследование алгоритма плавной сортировки в сравнении с поразрядной сортировкой.

В исследование входят следующие подзадачи:

- Генерацию представительного множества реализаций входных данных.
- Выполнение исследуемых алгоритмов на сгенерированных наборах данных.
- При этом в ходе вычислительного процесса фиксируется как характеристики работы программы, так и количество произведенных базовых операций алгоритма.
- Фиксацию результатов испытаний алгоритма, накопление статистики.
- Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Определение кучи леонардовых куч

Числа Леонардо — последовательность чисел, задаваемая зависимостью:

$$L(n) := \begin{cases} 1, & \text{если } n=0 \\ 1, & \text{если } n=1 \\ L(n-1) + L(n-2) + 1, & \text{если } n>1 \end{cases}$$

Эдсгер Дейкстра использовал их как составную часть своего алгоритма плавной сортировки, и изучил их некоторые особенности.

Первые 20 чисел Леонардо:

1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529

Абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера.

Это в нашем случае очень полезно. Массив из n элементов не всегда можно представить в виде одной кучи Леонардо (если n не является числом Леонардо). Но зато, любой массив всегда можно разделить на несколько подмассивов, которые будут соответствовать разным числам Леонардо, т.е. представлять из себя кучи разного порядка.

Пример массива из 21-го элемента, состоящий из трёх леонардовых куч представлен на рисунке 1. В каждой из куч количество узлов соответствует какому-либо числу Леонардо.

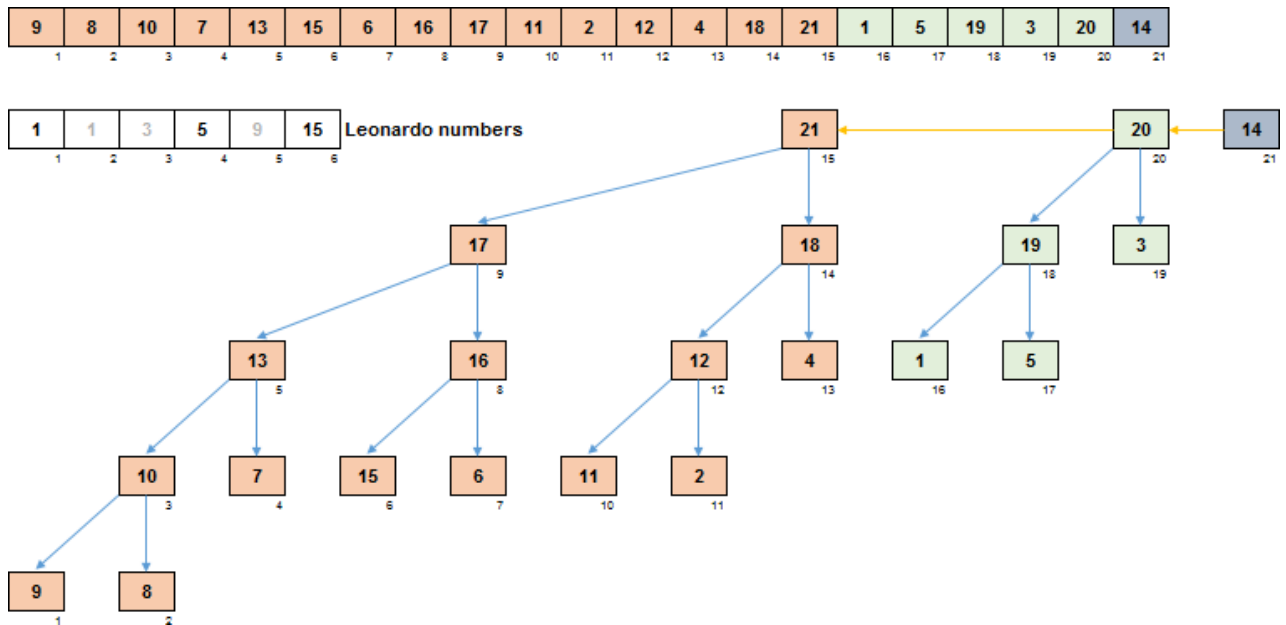


Рисунок 1 — Пример массива леонардовых куч

1. Каждая леонардова куча представляет собой несбалансированное бинарное дерево.
2. Корень каждой кучи — это последний (а не первый, как в обычной бинарной куче) элемент соответствующего подмассива.
3. Любой узел со всеми своими потомками также представляет из себя леонардову кучу меньшего порядка.

2.2. Описание алгоритма плавной сортировки

Пусть у нас в массиве есть два соседних подмассива, которые соответствует кучам, построенных на двух соседних числах Леонардо. С помощью элемента, который находится сразу за этими подмассивами, эти подмассивы можно объединить в общую кучу, которая соответствует следующему леонардовому числу.

Перебирая элементы в массиве мы так и выстраиваем кучу из леонардовых куч. Если с помощью элемента можно объединить две предыдущие кучи (это возможно тогда и только тогда, когда две предыдущие кучи соответствуют двум последовательным числам Леонардо), то и

объединяем. Если объединение не возможно (две предыдущие кучи не соответствуют двум последовательным числам Леонардо), то текущий элемент просто образует новую кучу из одного элемента, соответствующую первому (или второму, если первое использовано перед ним) числу Леонардо.

Пример построения кучи куч показан на рисунке 2.

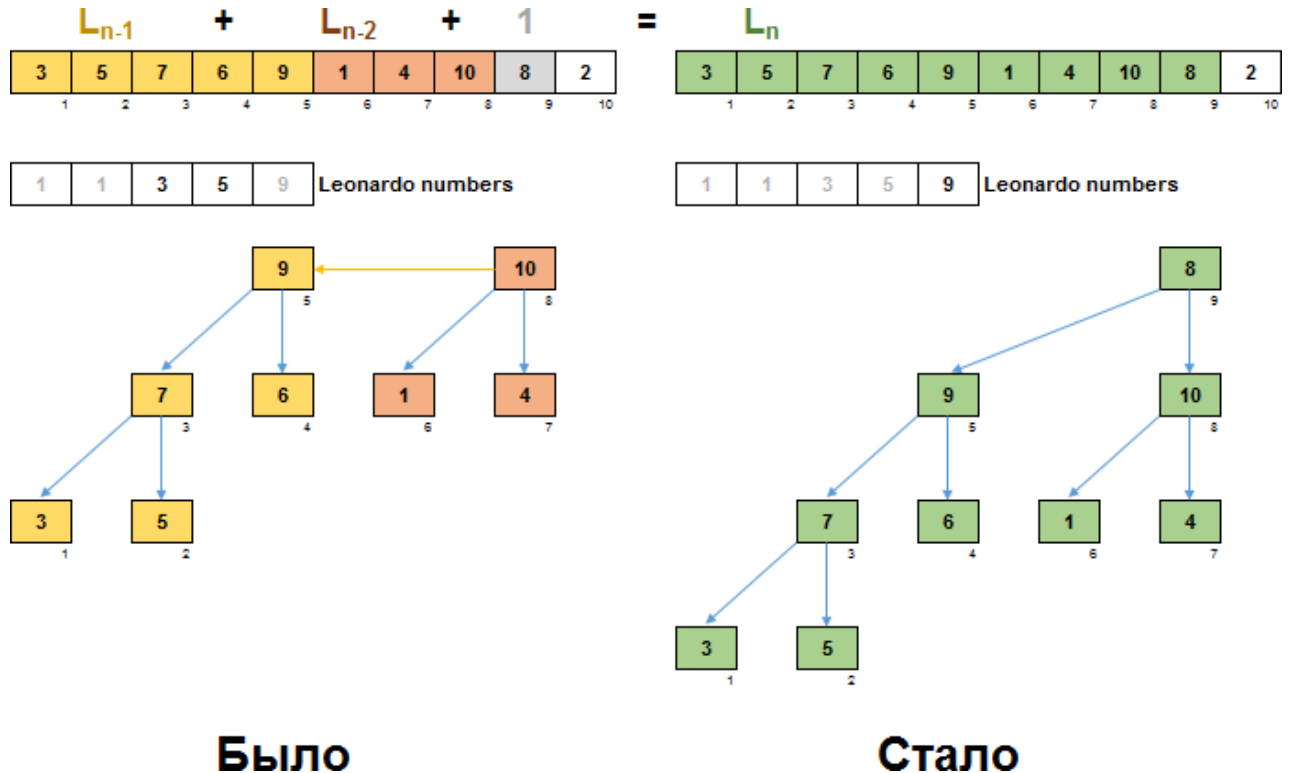


Рисунок 2 — Пример построения кучи леонардовых куч

Итоговый алгоритм плавной сортировки:

1. Создаём из массива кучу леонардовых куч, каждая из которых является сортирующим деревом.

1.1. Перебираем элементы массива слева-направо.

1.2. Проверяем, можно ли с помощью текущего элемента объединить две крайние слева кучи в уже имеющейся куче леонардовых куч:

1.2.a. Если да, то объединяем две крайние слева кучи в одну, текущий элемент становится корнем этой кучи, делаем просейку для объединённой кучи.

1.2.6. Если нет, то добавляем текущий элемент в качестве новой кучи (состоящей пока из одного узла) в имеющуюся кучу леонардовых куч.

2. Извлекаем из куч текущие максимальные элементы, которые перемещаем в конец неотсортированной части массива:

2.1. Ищем максимумы в леонардовых кучах. Так как на предыдущем этапе для куч постоянно делалась просейка, максимумы находятся в корнях этих куч.

2.2. Найденный максимум (который является корнем одной из куч) меняем местами с последним элементом массива (который является корнем самой последней кучи).

2.3. После этого обмена куча, в которой был найден максимум перестала быть сортирующим деревом. Поэтому делаем для неё просейку.

2.4. В последней куче удаляем корень (в которой находится текущий максимум), в результате чего эта куча распадается на две кучи поменьше.

2.5. После перемещения максимального элемента в конец, отсортированная часть массива увеличилась, а неотсортированная часть уменьшилась. Повторяем пункты 2.1-2.4 для оставшейся неотсортированной части массива.

2.3. Описание алгоритма поразрядной сортировки

Исходно предназначен для сортировки целых чисел, записанных цифрами.

Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Так как выравнивать сравниваемые записи относительно друг друга можно в разную сторону, на практике существуют два варианта этой сортировки. Для чисел они называются в терминах значимости разрядов числа, и получается так: можно выровнять записи чисел в сторону менее значащих цифр (по правой стороне, в сторону единиц, least significant digit, LSD) или более значащих цифр (по левой стороне, со стороны более значащих разрядов, most significant digit, MSD).

При LSD сортировке (сортировке с выравниванием по младшему разряду, направо, к единицам) получается порядок, уместный для чисел. Например: 1, 2, 9, 10, 21, 100, 200, 201, 202, 210. То есть, здесь значения сначала сортируются по единицам, затем сортируются по десяткам, сохраняя отсортированность по единицам внутри десятков, затем по сотням, сохраняя отсортированность по десяткам и единицам внутри сотен, и т. п.

2.4. Описание структур данных и функций программы

1. Класс Smootshort – класс представления плавной и поразрядной сортировок.

Поля класса:

- `std::vector<size_t> leonardo` — вектор леонардовых чисел;

Методы класса:

- `void createLeonardoNumbers(const unsigned sizeVec)` — создает вектор чисел леонардо.
- `bool isLeonardoNumbers(const size_t digit)` — проверяет, является ли `digit` леонардовым числом.
- `void restoreHeap(std::shared_ptr<Node> &node)` — просейка кучи.
- `void smootshort(std::vector<int> &sortVec)` — плавная сортировка вектора.
- `void radix(std::vector<int> digitArr)` — поразрядная сортировка вектора.

Исходный код в приложении А.

2. Класс Node – класс представляет узел кучи.

Поля класса:

- `int data` — данные, хранящиеся в данном узле.
- `size_t count` — счетчик, определяющий кол-во узлов в куче.
- `Std::shared_ptr<Node> left` — умный указатель на левого предка.
- `Std::shared_ptr<Node> right` — умный указатель на правого предка.

3. Функции:

- `void generator(const size_t count, std::vector<int>& arr)` — функция для генерации вектора случайных значений.

Исходный код в приложении Б.

4. Функция `main`:

Функция `main` имеет 2 возможных варианта работы:

- Если на вход подаётся один аргумент, то программа генерирует такое количество тестов, которое ввёл пользователь. Сгенерированные данные подаются на вход плавной и поразрядной сортировкам.
- Если в `main` подаётся несколько аргументов, то из этих значений формируется один тест, который проходит сортировку реализованным алгоритмом.

Все тесты также проходят и поразрядную сортировку, это необходимо в дальнейшем для сравнения двух алгоритмов между собой по времени выполнения.

Исходный код в приложении В.

3. ТЕСТИРОВАНИЕ

Входные данные: на вход программе подаётся текстовый файл, котором через пробел записаны численные значения. Либо пользователь вводит значения самостоятельно через консоль.

Выходные данные: программа выводит элементы в отсортированном порядке в консоль.

Тестирование производится при помощи скрипта script, написанного на Bash. Исходный код представлен в приложении Г.

Для сборки программы был также написан Makefile. Исходный код представлен в приложении Д.

Пример запуска тестирующего скрипта представлен на рисунке 2.

Результаты тестирования представлены в таблице 1.

| № | Входные данные | Выходные данные | Комментарии |
|---|-------------------------------------------------------------|-------------------------------------------------------------|------------------------------------|
| 1 | 2 4 -6 8 -10 12 43 65 76 | -10 -6 2 4 8 12 43 65 76 | Набор случайных чисел |
| 2 | 2 1 | 1 2 | Проверка на малом кол-ве элементов |
| 3 | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | Подача уже упорядоченного массива |
| 4 | 93 23 13 14 | 13 14 23 93 | Набор случайных чисел |
| 5 | -3 4 5 6 -4 23 -12 65 -76 -1 23 455 | -76 -12 -4 -3 -1 4 5 6 23 23 65 455 | Набор случайных чисел |
| 6 | -1 32 56 7 34 7 87 23 45 54 | -1 7 7 23 32 34 45 54 56 87 | Набор случайных чисел |
| 7 | 945 34 23 656 834 734 232 | 23 34 232 656 734 834 945 | Набор случайных чисел |

```
Test 1:
Начальная строка = 2 4 -6 8 -10 12 43 65 76
-10 -6 2 4 8 12 43 65 76

Test 2:
Начальная строка = 2 1
1 2

Test 3:
Начальная строка = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Test 4:
Начальная строка = 93 23 13 14
13 14 23 93

Test 5:
Начальная строка = -3 4 5 6 -4 23 -12 65 -76 -1 23 455
-76 -12 -4 -3 -1 4 5 6 23 23 65 455

Test 6:
Начальная строка = -1 32 56 7 34 7 87 23 45 54
-1 7 7 23 32 34 45 54 56 87

Test 7:
Начальная строка = 945 34 23 656 834 734 232
23 34 232 656 734 834 945
```

Рисунок 2 — Пример вывода программы при запуске через script

4. ИССЛЕДОВАНИЕ

4.1. Сложность по времени плавной сортировки и поразрядной

Для поразрядной сортировки сложность по времени зависит от количества бит, требуемых для хранения каждого ключа (w). В худшем случае это приводит к $O(w \cdot n)$, в лучшем алгоритм стремиться к $O(n)$.

Для плавной сортировки.

На первом этапе мы перебираем n элементов, добавляя его в уже имеющиеся слева кучи. Само добавление в кучу обходится в $O(1)$, но затем для кучи нужно сделать просейку. В упорядоченных данных неглубокая просейка часто обходится $O(1)$ для одного добавляемого в кучу элемента. В неупорядоченных данных просейка для каждого добавления обходится в $O(\log n)$, так как из-за рандома просейке приходится проходить уровни дерева часто до самого низа.

Поэтому, на первом этапе наилучшая сложность по времени:

- для почти упорядоченных данных — $O(n)$,
- для случайных данных — $O(n \log n)$.

Для второго этапа ситуация аналогичная. При обмене очередного максимума опять нужно просеять кучу, в корне которой он находился. И показатели просейки для упорядоченных и неупорядоченных данных будут различные.

На втором этапе наилучшая сложность по времени такая же как и на первом:

- для почти упорядоченных данных — $O(n)$,
- для случайных данных — $O(n \log n)$.

Складывая временные сложности для первого и второго этапа:

- для почти упорядоченных данных — $O(2n) = O(n)$,
- для случайных данных — $O(2 n \log n) = O(n \log n)$.

В общем, худшая и средняя временная сложность для плавной сортировки — $O(n \log n)$.

Дейкстра в своих выкладках доказал, что лучшая сложность плавно стремится к $O(n)$ чем более упорядоченными будут входящие данные. Отсюда и название — плавная сортировка.

Сравнение по времени плавной сортировки с алгоритмом поразрядной представлено в таблице 2.

Таблица 2 - сравнение плавной сортировки с поразрядной по времени

| | Сложность по времени | | |
|------------------------|----------------------|----------------------|--------|
| | Худшая | Средняя | Лучшая |
| Плавная сортировка | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ |
| Поразрядная сортировка | $O(w \cdot n)$ | $O(w \cdot (w + n))$ | $O(n)$ |

4.2. Затраты на дополнительную память плавной и поразрядной сортировок

Для поразрядной сортировки.

Элементы перегруппировываются по определённому разряду (сначала по самому младшему). Затем разбиваются на подгруппы в зависимости от значения этого разряда: равного 0, равного 1, равного 2, ..., равного 9. После разбиения на блоки, они объединяются в единый массив и сортировка повторяется, но уже для следующего разряда. Засчёт такого разброса, где необходимо ещё хранить значение каждого ключа, сложность по дополнительной памяти следующая: $O(w + n)$.

Для плавной сортировки.

Чтобы разложить данные на кучу леонардовых куч, достаточно только запоминать, какие именно числа Леонардо задействованы на каждом шаге. Зная

эти числа, алгоритмически выстраиваются сами кучи. Этот числовой ряд растёт очень быстро, поэтому даже для крупных массивов понадобится совсем небольшой набор леонардовых чисел. Алгоритм плавной сортировки требует памяти для хранения размеров всех куч в последовательности. Так как все эти значения различны, как правило, для этой цели применяется битовая карта. Кроме того, так как в последовательности не больше $O(\log n)$ чисел, биты могут быть закодированы $O(1)$ машинными словами при условии использования трансдихотомической модели. Для хранения списка длин куч придется выделить $O(\log N)$ дополнительной памяти.

4.3. План исследования

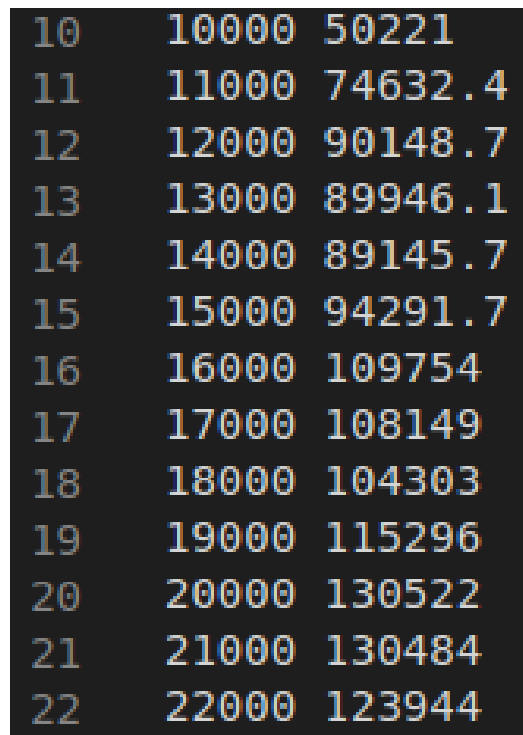
При запуске программы через исполняемый файл пользователю предлагают ввести число, на основе которого будет сгенерированно заданное количество тестов со случайными наборами данных.

Каждая последовательность подаётся алгоритму плавной сортировки и поразрядной соответственно. Во время работы алгоритмов фиксируются количество микросекунд, за которое та или иная сортировка справляется с массива.

Результаты исследования записываются в файлы `timeRadix.txt` и `timeSmootshort.txt` для плавной и поразрядной сортировок соответственно

Вывод записывается в виде пар значений «количество элементов в массиве — время сортировки».

Пример записи исследования в файл для плавной сортировки (в каждом последующем тесте количество элементов по умолчанию увеличивается на 2) представлен на рисунке 3.



| | | |
|----|-------|---------|
| 10 | 10000 | 50221 |
| 11 | 11000 | 74632.4 |
| 12 | 12000 | 90148.7 |
| 13 | 13000 | 89946.1 |
| 14 | 14000 | 89145.7 |
| 15 | 15000 | 94291.7 |
| 16 | 16000 | 109754 |
| 17 | 17000 | 108149 |
| 18 | 18000 | 104303 |
| 19 | 19000 | 115296 |
| 20 | 20000 | 130522 |
| 21 | 21000 | 130484 |
| 22 | 22000 | 123944 |

Рисунок 3 — Пример записи исследования затраченного времени для плавной сортировки

4.4. Результаты исследования

На основе записанных в файлы timeRadix.txt и timeSmootshort.txt данных были построены графики.

Следующие 4 рисунка показывают зависимость времени работы алгоритмов плавной и поразрядной сортировки от количества элементов в массиве. Первые 2 графика для среднего случая с случайными последовательностями чисел, вторые 2 графика для лучшего случая с подаваемыми на вход упорядоченными последовательностями. Последние 3 графика показывают скорость выполнения алгоритмов в зависимости от разрядности чисел хранящихся в сортируемых массивах.

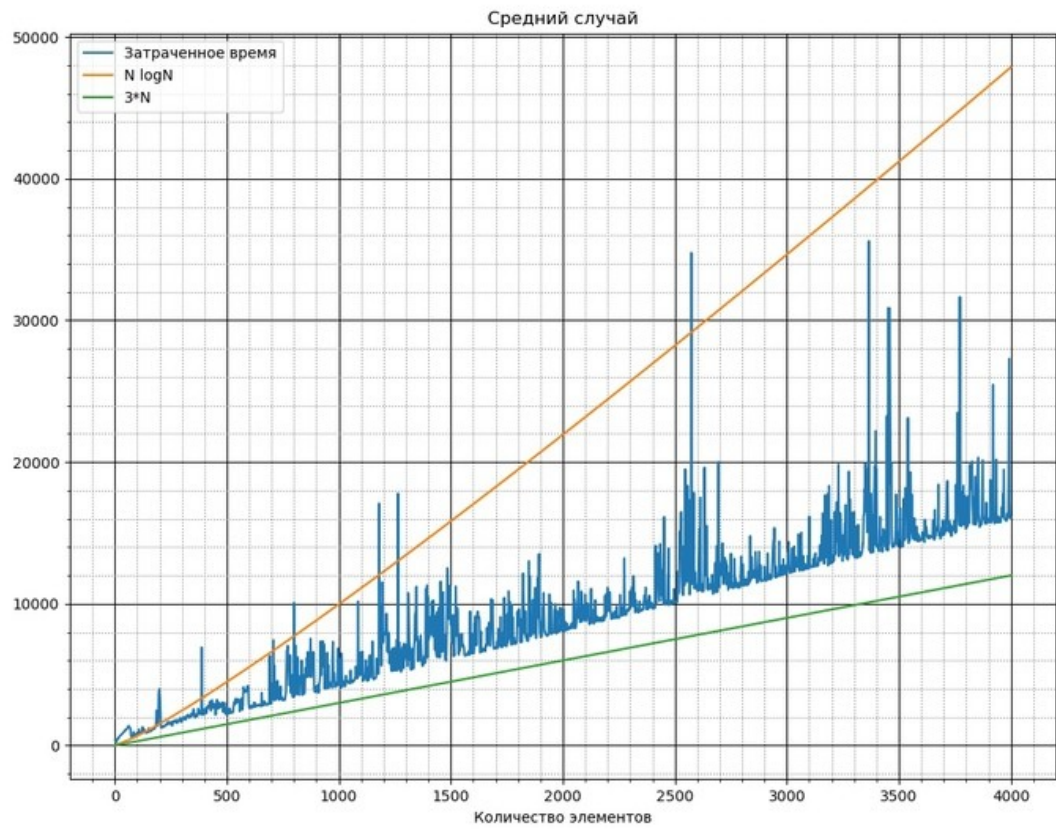


Рисунок4 — График 2000 тестов для поразрядной сортировки случайных последовательностей

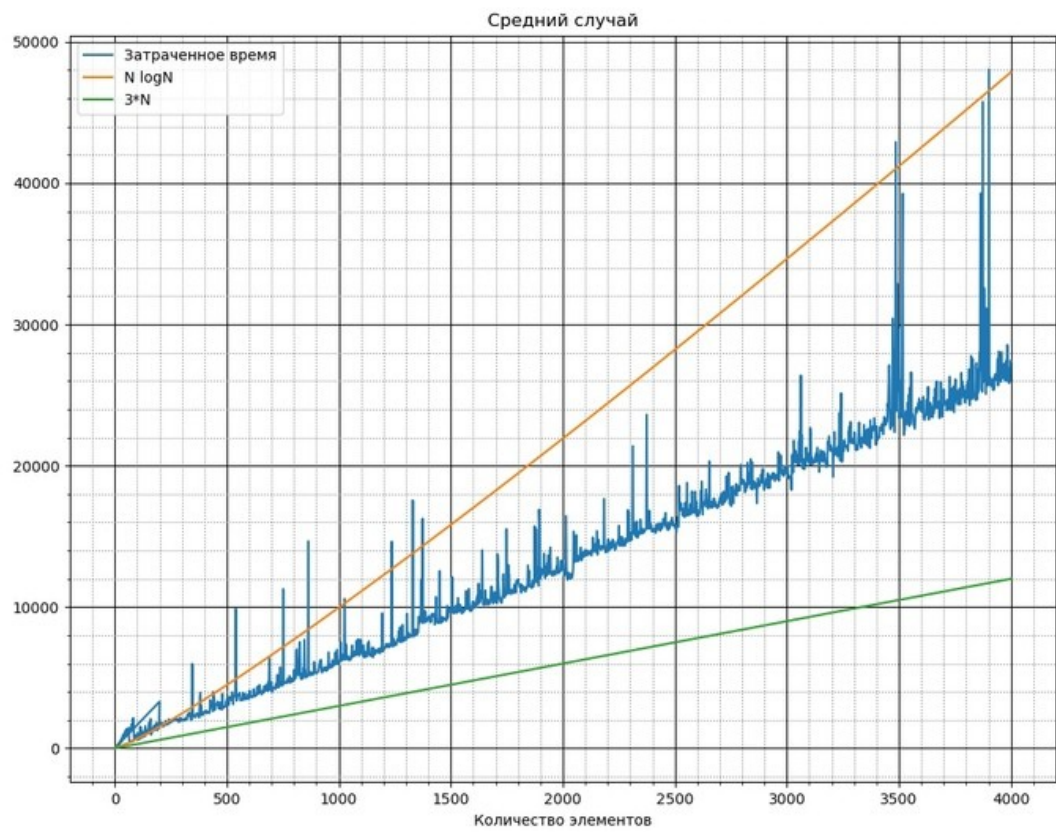


Рисунок 5 — График 2000 тестов для плавной сортировки случайных последовательностей

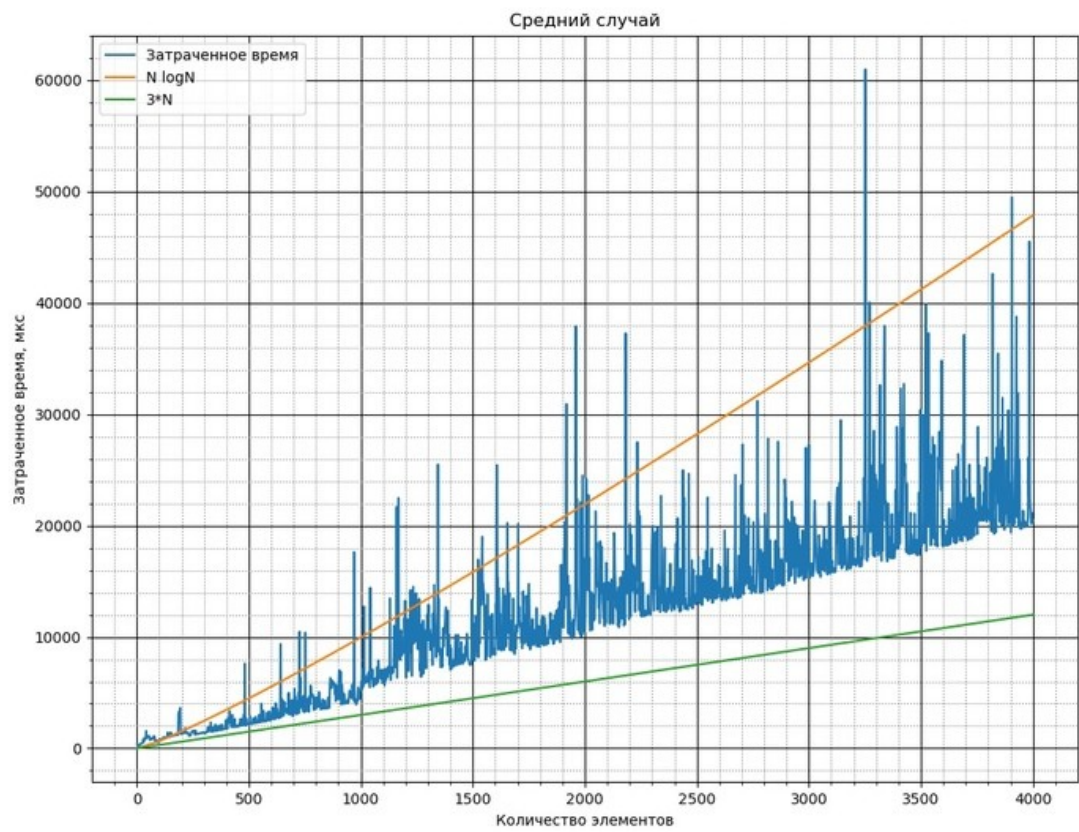


Рисунок 6 — График 2000 тестов для поразрядной сортировки упорядоченных значений

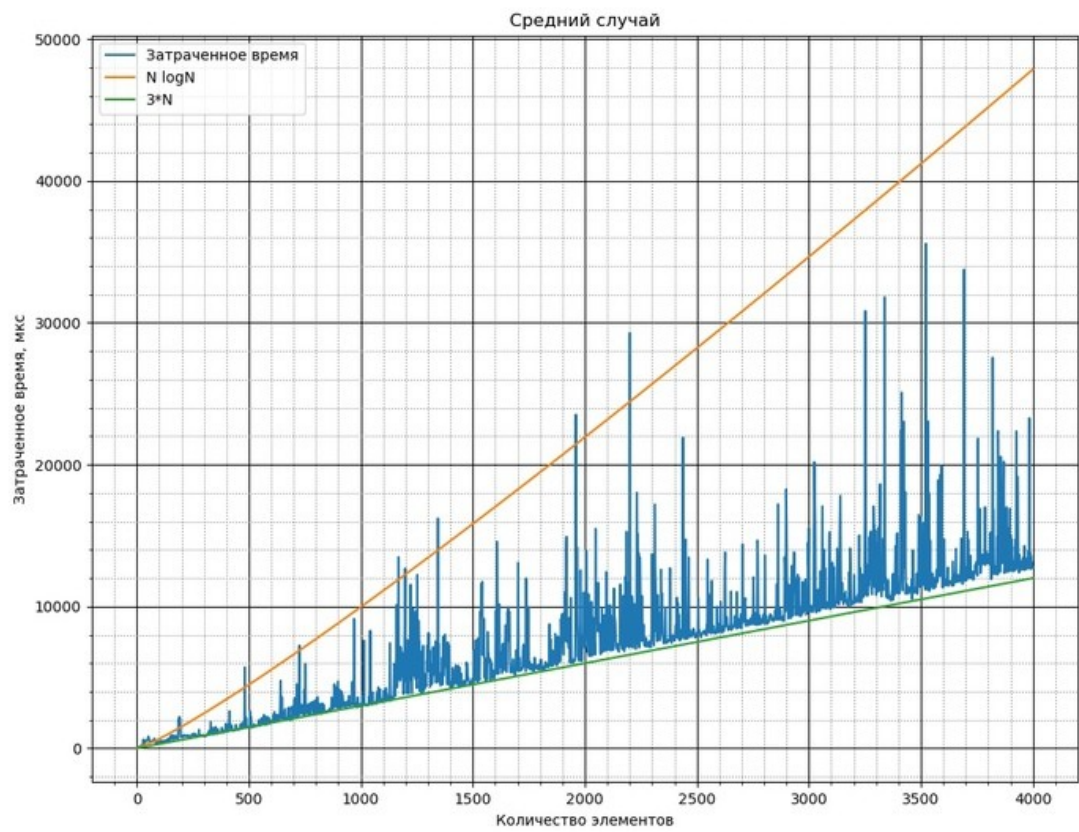
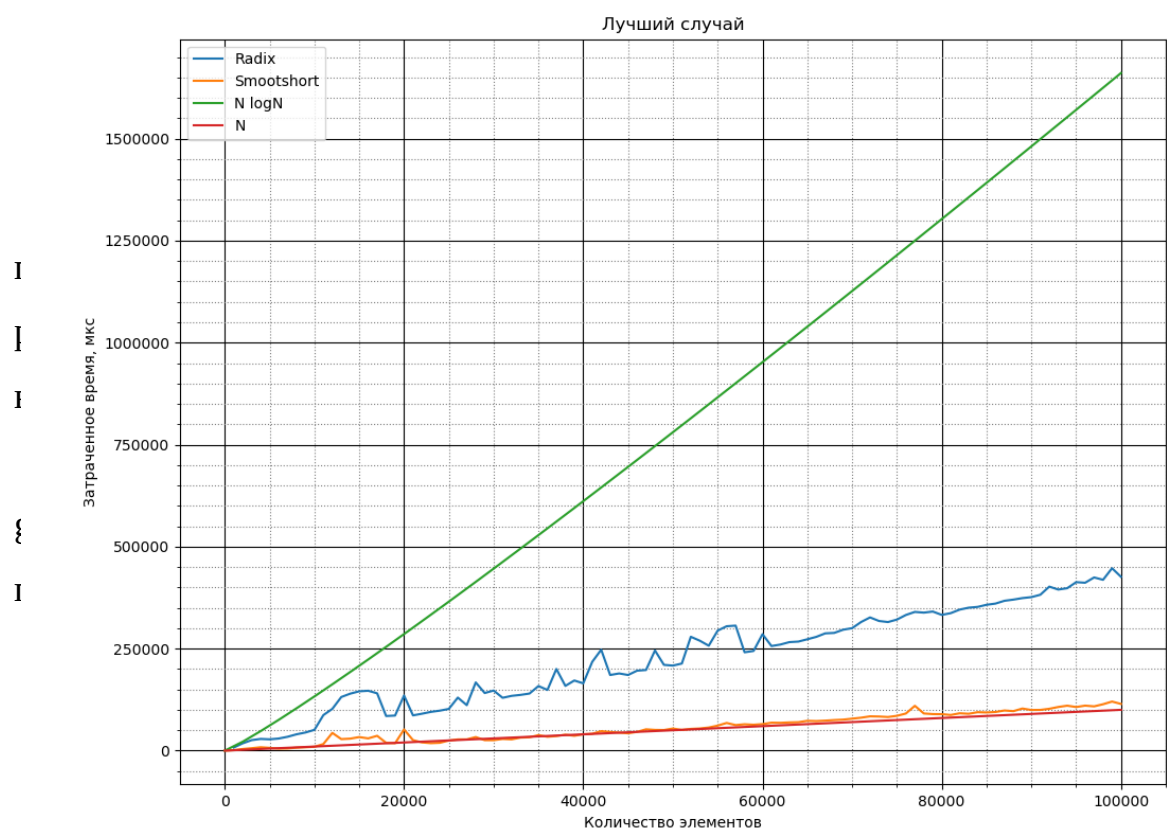


Рисунок 7 — График 2000 тестов для плавной сортировки упорядоченных последовательностей



Опираясь на графики можно сделать следующий вывод: скорость работы поразрядной сортировки напрямую зависит от количества разрядов сортируемых чисел и не зависит от их расположения в массиве. В то время как плавная сортировка показывает лучший результат в случае, если все элементы упорядочены.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была реализована такая структура данных как куча леонардовых куч. На базе такой структуры также был изучен и написан алгоритм плавной сортировки.

Были проведены исследования времени, которые алгоритмы затрачивают на сортировку массива значений. На основе полученных результатов, были построены графики зависимости времени выполнения алгоритма от количества элементов в массиве.

Также был написан Makefile и скрипт для тестирования. Была реализована программа на языке Python для построения графиков по измеренным в ходе работы программы значениям.

Полученные практические результаты сравнили с теоретическими оценками, в результате чего выяснили, что скорость выполнения поразрядной сортировки зависит от количества разрядов сортируемых чисел. В то время как плавная сортировка показывает лучший результат в случае, если все элементы упорядочены. Худшее же время достигается если данные не упорядочены, и для каждой леонардовой кучи необходима просейка на максимальную глубину.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [https://ru.wikipedia.org/wiki/%D0%9A%D1%83%D1%87%D0%B0_\(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85\)](https://ru.wikipedia.org/wiki/%D0%9A%D1%83%D1%87%D0%B0_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85))
2. <https://habr.com/ru/company/edison/blog/495420/>
3. <https://habr.com/ru/company/edison/blog/496852/>
4. <https://neerc.ifmo.ru/wiki/index.php?title=Smoothsort>
5. <https://temofeev.ru/info/articles/plavnaya-sortirovka/>
6. https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%9B%D0%B5%D0%BE%D0%BD%D0%B0%D1%80%D0%B4%D0%BE
7. https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%B0%D0%B7%D1%80%D1%8F%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0
8. https://pro-prof.com/forums/topic/radix_sort
9. <http://algotlab.valemak.com/radix>
10. <http://algotlab.valemak.com/lsd>

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД SMOOTSHORT

Файл Smootshort.h

```
#pragma once

#include <vector>

class Node;

class Smootshort{
    std::vector<size_t> leonardo {};
    void createLeonardoNumbers(const unsigned sizeVec);
    bool isLeonardoNumbers(const size_t digit);
    void restoreHeap(std::shared_ptr<Node> &node);
public:
    void smootshort(std::vector<int> &sortVec);
    void radix(std::vector<int> digitArr);
};
```

Файл Smootshort.cpp

```
#include <iostream>
#include <memory>
#include <chrono>
#include <fstream>
#include <cmath>
#include <algorithm>

#include "Smootshort.h"
#include "Node.h"

void Smootshort::createLeonardoNumbers(const unsigned sizeVec){
    if(sizeVec <= 1){
        throw "Number of values <= 1";
    }
    unsigned a = 1;
    unsigned b = 1;
    unsigned c = 1;
```

```

while(b <= sizeVec){
    leonardo.emplace_back(b);
    a = b;
    b = c;
    c = a + b + 1;
}
}

```

```

bool Smootshort::isLeonardoNumbers(const size_t digit){
    for(size_t i = 1; i < leonardo.size(); i++){
        if((digit + 1) == leonardo[i]){
            return true;
        }
    }
    return false;
}

```

```

void Smootshort::restoreHeap(std::shared_ptr<Node> &node){

    auto PR = [](std::shared_ptr<Node> &node, auto &&PR){

        if(!node->left && !node->right){
            return;
        }
        if(node->left->data < node->right->data){
            if(node->data < node->right->data){
                std::swap(node->data, node->right->data);
                PR(node->right, PR);
            }
        }else if(node->left->data > node->right->data){
            if(node->data < node->left->data){
                std::swap(node->data, node->left->data);
                PR(node->left, PR);
            }
        }
    };

    PR(node, PR);
}

```

```

void Smootshort::smootshort(std::vector<int> &sortVec){

    try{
        createLeonardoNumbers(sortVec.size());

        std::vector<std::shared_ptr<Node>> heap{};

        for(size_t i = 0; i < sortVec.size(); i++){    // Создание леонардовых куч

            if(heap.size() >= 2){

                if(isLeonardoNumbers(heap[heap.size() - 2]->count +
heap[heap.size() - 1]->count)){
                    auto q = std::make_shared<Node>(sortVec[i]);
                    q->left = heap[heap.size() - 2];
                    q->right = heap[heap.size() - 1];
                    q->count = q->left->count + q->right->count + 1;
                    restoreHeap(q);
                    heap.pop_back();
                    heap.pop_back();
                    heap.emplace_back(q);
                    continue;
                }
            }

            auto q = std::make_shared<Node>(sortVec[i]);
            heap.emplace_back(q);
        }

        std::chrono::high_resolution_clock::time_point begin =
std::chrono::high_resolution_clock::now();

        long position = sortVec.size();
        size_t count;

        while(position){    // Разбираем леонардовы кучи

            count = heap.size() - 1;
            position--;

            if(heap.size() == 1){    // Если куча одна - извлекаем
корень и разбиваем на подкучи

```



```

        std::shared_ptr<Node> tmp = heap[0];
        sortVec[position] = tmp->data;
        heap.pop_back();
        heap.emplace_back(tmp->left);
        heap.emplace_back(tmp->right);
    }else{
        // Если же несколько, то находим
        наибольший корень и меняем его со значением последнего элемента
        for(int i = heap.size() - 2; i >= 0; i--){
            if(heap[heap.size() - 1]->data < heap[i]->data){
                count = i;
            }
        }
        if(count != heap.size() - 1){
            std::swap(heap[heap.size() - 1]->data, heap[count]->data);
            restoreHeap(heap[count]);
        }
        sortVec[position] = heap[heap.size() - 1]->data;

        if(heap[heap.size() - 1]->count > 1){
            std::shared_ptr<Node> tmp = heap[heap.size() - 1];
            heap.pop_back();
            heap.emplace_back(tmp->left);
            heap.emplace_back(tmp->right);
        }else{
            heap.pop_back();
        }
    }
}

std::chrono::high_resolution_clock::time_point end =
std::chrono::high_resolution_clock::now();
std::chrono::duration<double> result =
std::chrono::duration_cast<std::chrono::duration<double>>(end - begin);

std::ofstream timeFile("timeSmootshort.txt", std::ios_base::app);

timeFile << sortVec.size() << ' ';
timeFile << result.count() * 10'000'000 << '\n';

}catch(const char* err){
    std::cerr << err << std::endl;
}
}

```

```

void Smootshort::radix(std::vector<int> digitArr){

    std::chrono::high_resolution_clock::time_point begin =
std::chrono::high_resolution_clock::now();

    int nim = *std::min_element(digitArr.begin(), digitArr.end());

    for(size_t i = 0; i < digitArr.size(); i++){
        digitArr[i] -= nim;
    }

    std::vector<std::vector<int>> vec {}; // Для каждого разряда свой блок
    unsigned deegree = 0;                // Разряд числа
    unsigned level;                       // Позиция блока в массиве
    bool checkRank = 0;                  // Проверка на наличие отличных от 0
рангов
    vec.resize(10);

    while(1){

        // Разбрасываем по соответствующим блокам
        deegree++;
        for(size_t i = 0; i < digitArr.size(); i++){
            level = digitArr[i] / ((int)pow(10, deegree - 1)) % 10;
            if(level){
                checkRank = 1;
            }
            vec[level].emplace_back(digitArr[i]);
        }

        // Перезапись базового вектора
        digitArr.clear();
        for(size_t i = 0; i < vec.size(); i++){
            for(size_t j = 0; j < vec[i].size(); j++){
                digitArr.emplace_back(vec[i][j]);
            }
        }

        // Если все элементы в 0 блоке - ливаем
        if(!checkRank){
            break;
        }
    }
}

```

```

    }else{
        checkRank = 0;
    }

    // Чистим для следующей итерации
    for(size_t i = 0; i < 10; i++){
        vec[i].clear();
    }

}

for(size_t i = 0; i < digitArr.size(); i++){
    digitArr[i] += nim;
}

std::chrono::high_resolution_clock::time_point end =
std::chrono::high_resolution_clock::now();
std::chrono::duration<double> result =
std::chrono::duration_cast<std::chrono::duration<double>>(end - begin);

std::ofstream timeFile("timeRadix.txt", std::ios_base::app);

timeFile << digitArr.size() << ' ';
timeFile << result.count() * 10'000'000 << "\n";

}

```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД NODE

Файл Node.h

```
#pragma once

#include <memory>

class Node{
public:
    Node(int newData);
    int data;
    size_t count;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
};
```

Файл Node.cpp

```
#include "Node.h"

Node::Node(int newData){
    data = newData;
    count = 1;
}
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД MAIN

Файл main.cpp

```
#include <iostream>
#include <sstream>
#include <iterator>
#include <vector>
#include <memory>
#include <ctime>

#include "Smootshort.h"
#include "Node.h"

void generator(const size_t count, std::vector<int>& arr){
    arr.clear();
    srand(time(0));
    for (size_t i = 0; i < count; i++){
        //arr.push_back(rand() % 1000);
        arr.push_back(i);
    }
}

int main(){

    std::string inputString {};
    getline(std::cin, inputString);
    std::stringstream strStream(inputString);
    std::vector<int> vec {};
    std::copy(std::istream_iterator<int>(strStream), {}, back_inserter(vec));

    std::shared_ptr<Smootshort> SortClass(new Smootshort);

    if(vec.size() == 1){
        std::vector<int> testVec {};
        for(int i = 1; i <= vec[0]; i++){
            generator(2 * i, testVec);
            SortClass->smootshort(testVec);
            SortClass->radix(testVec);
        }
    }
}
```

```
    }  
    }else{  
        SortClass->smootshort(vec);  
        for(size_t i = 0; i < vec.size(); i++){  
            std::cout << vec[i] << ' ';  
        }  
        std::cout << std::endl;  
    }  
  
    return 0;  
}
```

ПРИЛОЖЕНИЕ Г

ИСХОДНЫЙ КОД SCRIPT1

Файл script1

```
#!/bin/bash

for n in {1..7}
do
    arg=$(cat Tests/test$n.txt)
    echo -e "\nTest $n:"
    echo "Начальная строка = $arg"
    ./cw < Tests/test$n.txt
done
```

ПРИЛОЖЕНИЕ Д

ИСХОДНЫЙ КОД MAKEFILE

Файл Makefile

```
CC = g++
CFLAG = -std=c++17 -Wall
DIR = ./Source/

cw: $(DIR)main.cpp $(DIR)Smootshort.cpp $(DIR)Node.cpp
    $(CC) $(CFLAG) $^ -o $@

.PHONY: clean

clean:
    rm cw timeSmootshort.txt timeRadix.txt
```


ПРИЛОЖЕНИЕ Е

ИСХОДНЫЙ КОД GRAPHS

Файл `graphs.py`

```
import sys
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import math
import numpy as np
```

```
file = sys.argv[1]
```

```
fs = file
```

```
f = open(fs, 'r')
```

```
x, y = [0], [0]
l = 0
```

```
m,n = 0,0
```

```
p = 0
```

```
num = 10
```

```
for l in f:
    row = l.split()
    m += float(row[0])
    n += float(row[1])
    x.append(float(row[0]))
    y.append(float(row[1]))
```

```
f.close()
```

```
fig, ax1 = plt.subplots(
    nrows = 1, ncols = 1,
    figsize = (12, 12)
)
```

```
xmax = math.sqrt(y[len(y) - 1])
```

```
t = np.linspace(0.1, max(x))
```

```

j = np.linspace(0.1, max(x))

a = np.log2(t) * t

v = t ** 2

o = j*3

x.pop(0)
y.pop(0)

ax1.plot(x, y, label = 'Затраченное время')
ax1.plot(t, a, label = 'N logN')
ax1.plot(j, o, label = '3*N')
ax1.plot()

ax1.grid(which = 'major',
        color = 'k')

ax1.minorticks_on()

ax1.grid(which = 'minor',
        color = 'gray',
        linestyle = ':')

ax1.legend()

ax1.set_xlabel('Количество элементов')
ax1.set_ylabel('Затраченное время, мкс')

plt.title("Лучший случай")
plt.show()

```