Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Сортировки»**

**Выполнил**:

студент группы 3824Б1ПМ1

Бутусов А. П.

**Проверил**:

преподаватель каф. ВВСП

Волокитин В.Д.

Нижний Новгород

2024

**Содержание**

[Постановка задачи 3](#_Toc185422498)

[Метод решения 4](#_Toc185422499)

[Руководство пользователя 6](#_Toc185422500)

[Описание программной реализации 9](#_Toc185422501)

[Подтверждение корректности 17](#_Toc185422502)

[Результаты экспериментов 19](#_Toc185422503)

[Заключение 23](#_Toc185422504)

[Литература 24](#_Toc185422505)

[Приложение 25](#_Toc185422506)

# Постановка задачи

В рамках данной лабораторной работы необходимо:

1. Реализовать на языке программирования C несколько алгоритмов сортировки массивов, таких как:
   * Пузырьковая сортировка (Bubble Sort)
   * Сортировка расческой (Comb Sort)
   * Merge Sort
   * Radix Sort
2. Написать программу, которая:
   * создаёт массивы случайных чисел разного размера
   * сортирует их с использованием реализованных алгоритмов
   * измеряет время выполнения каждого алгоритма для каждого массива
   * подсчитывает количество операций
   * проверяет правильность сортировки (выводит массивы)
3. Провести тестирование алгоритмов:
   * выполнить сортировку для разных массивов
   * вывести полученные результаты времени выполнения
   * считает количество операций
4. На основе полученных данных сравнить время выполнения сортировки для разных размеров массивов, сделать вывод с какой сложностью увеличивается время сортировки и посчитать.

# Метод решения

В данной программе исследуется четыре сортировки (Bubble Sort, Comb Sort, Merge Sort, Radix Sort).

Рассмотрим сначала Bubble Sort. Сортировка пузырьком — это метод сортировки массивов и списков путём последовательного сравнения соседних элементов и их обмена, если предшествующий оказывается больше последующего (при сортировке по возрастанию).

Принцип работы: алгоритм проходит по списку и сравнивает каждую пару соседних элементов. Если элементы находятся в неправильном порядке (первый больше второго для сортировки по возрастанию) — они меняются местами. Этот процесс повторяется для всех пар элементов в списке. После каждого полного прохода самый большой элемент «всплывает» на своё место (как пузырь на поверхности воды), и поэтому он больше не участвует в следующих проходах. Процесс повторяется до тех пор, пока список не будет отсортирован.

Сложность сортировки .

Следующая сортировка Comb Sort. Основная идея «расчёски» в том, чтобы первоначально использовать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива уменьшать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, который может быть, например, 1.25. Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы как и в сортировке пузырьком, но такая итерация одна.

Сложность сортировки расчёской в худшем случае — , в лучшем — .

Далее рассмотрим Merge Sort. Сортировка слиянием (merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

Принцип работы основан на принципе «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи.

Алгоритм выполняется в несколько шагов:

1. Исходный массив разбивается на две части примерно одного и того же размера. 3
2. Каждая часть сортируется отдельно.
3. Два получившихся подмассива половинного размера соединяются в результирующий массив.

Главное преимущество сортировки слиянием — она работает всегда с одной и той же скоростью на любых массивах. Благодаря этому свойству алгоритм используют там, где нужно обработать данные за заранее известное время. 4

Временная сложность алгоритма — .

И наконец рассмотрим четвёртую сортировку “Radix Sort”. Radix Sort (цифровая сортировка) — алгоритм сортировки, использующий внутреннюю структуру сортируемых объектов.

Принцип работы в том, что алгоритм сортирует не числа целиком, а значения разрядов. Сначала элементы массива делят на группы по отдельным разрядам, а затем сортируют группы в порядке от младшего к старшему. Этот процесс повторяют до тех пор, пока не будут отсортированы все элементы.

Примеры объектов, которые удобно разбивать на разряды и сортировать по ним, — числа и строки. Для чисел уже существует понятие разряда, поэтому их представляют как последовательности разрядов. Строки представляют собой последовательности символов, поэтому в качестве разрядов в данном случае выступают отдельные символы.

Radix Sort часто работает быстрее других алгоритмов сортировки для больших наборов данных, особенно когда ключи имеют много разрядов. Однако его вычислительная сложность растёт линейно с количеством разрядов, поэтому для небольших наборов данных алгоритм не так эффективен.

Сложность алгоритма Radix Sort составляет , где n — количество элементов во входном массиве, а d — количество цифр в наибольшем числе.

# Руководство пользователя

**Описание работы программы.**

Программа предназначена для оценки времени выполнения различных алгоритмов сортировки на массивах разного размера, проверки корректности сортировки, подсчёта количества сравнений. Пользователь выбирает один из четырёх алгоритмов сортировки (Bubble sort, Comb sort, Merge sort, Radix sort), после чего программа предоставляет информацию с размерами массивов с соответствующими временами сортировки и количество присвоений (для Radix Sort) и сравнений (для остальных сортировок).

**Действия пользователя:**

Шаг 1: Запуск программы

1. Запустите программу. Появится консольное меню с предложением выбрать количество элементов для сортировки. (рис.1)

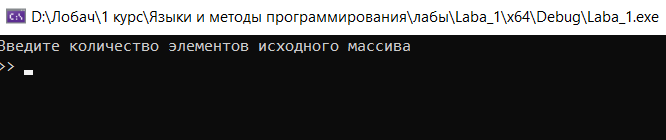


Рис. 1. Ввод количества элементов для массива

Шаг 2: Выбор количества элементов для сортировки

1. Выберите любое число (не более 100 тысяч элементов)

Шаг 3: В консольном меню появится список из четырёх сортировок. Выбор алгоритма сортировки (рис.2)

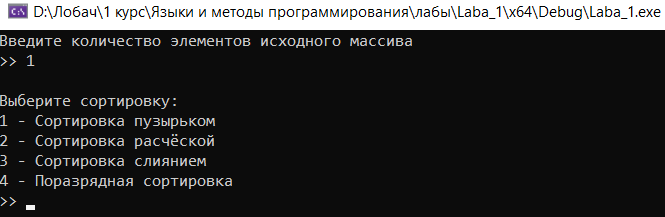


Рис.2. Выбор сортировки

1. Введите номер алгоритма сортировки, который вы хотите протестировать (от 1 до 4).

2. Нажмите клавишу ‘Enter’.

Шаг 3: Ожидание результата

1. Программа начнет вычислять время сортировки, количество операций для первого массива.

2. Затем для массива в 21, 22, 23, 24, 25 раз больше исходного. (рис. 3.)

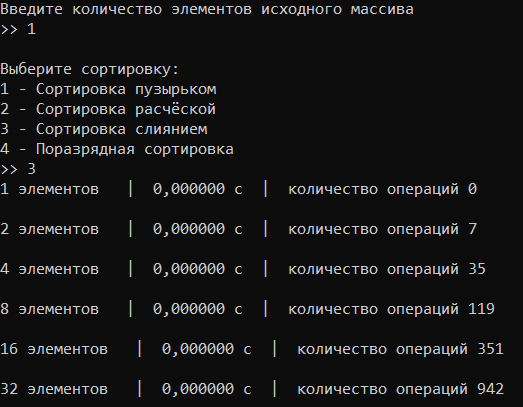


Рис. 3. Вывод результата сортировки Merge Sort

3. Также появляется список из элементов каждой сортировки, где можно убедиться в корректности сортировки, и количество операций у сортировок. (рис. 4)

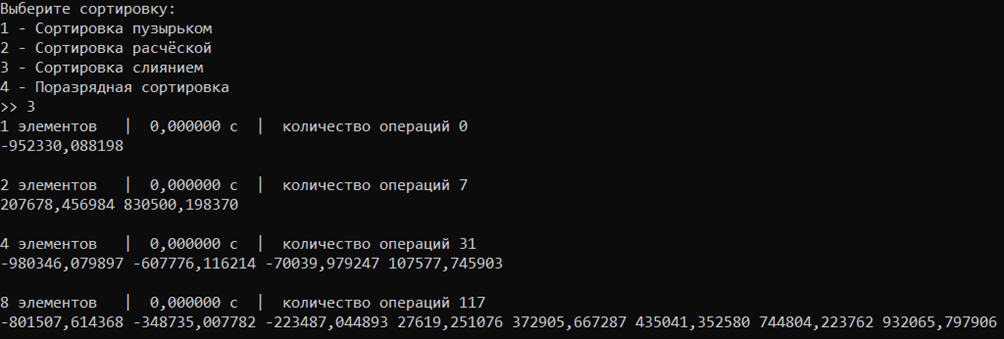


Рис.4. Вывод результатов сортировок

Шаг 4: Дополнительная информация о массиве.

1. Нужно написать: «0», чтобы вывелась дополнительная информация о массиве. (рис.5)

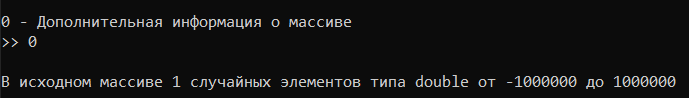


Рис.5. Дополнительная информация

Шаг 5: Просмотр результатов (рис.6)

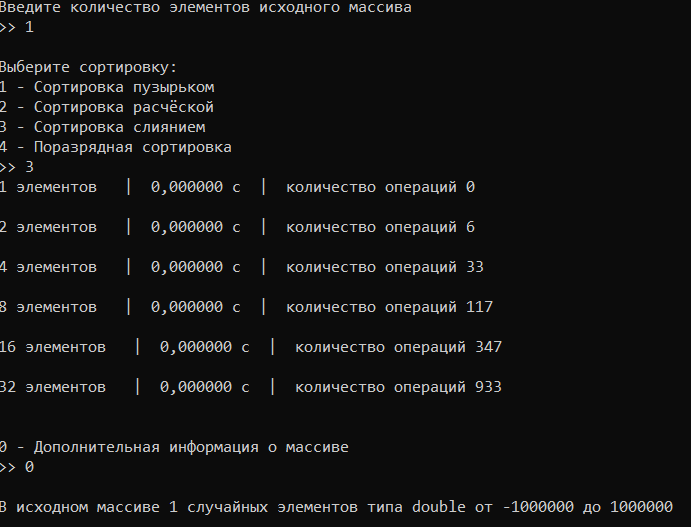


Рис.6. Весь интерфейс.

# Описание программной реализации

Данная программа создана с использованием IDE Visual Studio и компилятор MSVC. Она состоит из пяти файлов. Содержит три текстовых файла (“Laba\_1.cpp”, “Sortss.cpp”, “UsefulFunctionss.cpp”) и два созданных заголовочных файла (“Sortss.h”, “UsefulFunctionss.h”).

**1. Текстовые файлы.**

1.1. Текстовый файл “Laba\_1.cpp” - главный. В нём содержится интерфейс программы. В нём находятся заголовочные файлы, показанные на рис. 7

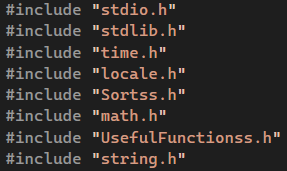


Рис. 7. Заголовочные файлы в “Laba\_1.cpp”

Подключенные библиотеки:

* “stdio.h”, используется для ввода и вывода данных в консольном приложении;
* “stdlib.h”, используется для работы с динамической памятью;
* “string.h”, используется в функции в Radix Sort;
* “time.h”, используется для подсчета времени выполнения сортировки;
* “locale.h”, используется для задач, связанных с локализацией;
* “math.h”, содержит множество математических функций для выполнения сложных расчётов.

1.2. Текстовый файл “Sortss.cpp” - файл с сортировками. В нём содержится код функций для четырёх сортировок: Bubble Sort, Comb Sort, Merge Sort, Radix Sort.

В нём находятся заголовочные файлы, показанные на рис. 8

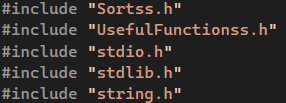


Рис. 8. Заголовочные файлы в “Sortss.cpp”

Прототипы функций:

* void Bubble(double a[], int n) – функция для Bubble Sort, в аргумент принимает массив double, int (размер массива).

Принцип работы сортировки пузырьком (Bubble Sort):

1. Проход по массиву: алгоритм последовательно сравнивает каждый элемент массива с его следующим соседом.
2. Обмен элементов: если текущий элемент больше следующего, они меняются местами. Таким образом, на каждом проходе самый большой элемент “всплывает” к своему правильному месту в конце массива.
3. Повторение процесса: этот процесс повторяется для каждого элемента массива, но с каждым полным проходом количество проверяемых элементов уменьшается на один, так как последний элемент уже отсортирован.

Подсчет операций в Bubble Sort:

#ifdef check\_operations

bubblechecks += 2;

#endif

* void Comb(double a[], int n) – функция для Comb Sort, в аргумент принимает массив double, int (размер массива).

Принцип работы сортировки расческой (Comb Sort):

Функция “Comb” сортирует массив чисел. Она использует улучшенный алгоритм пузырьковой сортировки, называемый сортировкой "расческой". Вот как это работает:

1. Уменьшение шага:

* Сначала шаг (step) устанавливается как половина длины массива.
* Внешний цикл уменьшает шаг в два раза в каждой итерации.
* Внутренний цикл сравнивает и меняет местами элементы, которые находятся на расстоянии step друг от друга, если они стоят в неправильном порядке.

2. Финальная пузырьковая сортировка:

* Когда шаг становится равным 1, выполняется улучшенная пузырьковая сортировка.
* Этот этап окончательно упорядочивает все элементы массива.

Подсчет операций в Comb Sort:

#ifdef check\_operations

combchecks += 2;

#endif

* void Merge(double\* a, int n, double\* b, int m, double\* c) – функция для Merge Sort, в аргумент принимает указатель на первый массив double, int (размер массива), указатель на второй массив double, int (размер массива), указатель на массив double, куда будет записан результат — объединенный и отсортированный массив.

Функция “Merge” реализует алгоритм слияния для объединения двух отсортированных массивов в один отсортированный массив.

1. Цикл for проходит от 0 до “n + m”, что соответствует общему количеству элементов в объединенном массиве.

2. На каждой итерации:

* Если индекс j достиг конца первого массива a, это означает, что все элементы из первого массива были добавлены в третий массив c. В этом случае оставшиеся элементы из второго массива b добавляются в массив c. То тогда записывает текущий элемент из второго массива b в третий массив c и увеличивает индекс k.
* Если индекс k достиг конца второго массива b, это означает, что все элементы из этого массива были добавлены в третий массив c. В этом случае оставшиеся элементы из первого массива a добавляются в третий массив c. Записывает текущий элемент из массива a в массив c и увеличивает индекс j.
* Если оба массива еще не закончены, сравниваются текущие элементы из обоих массивов.
* Если элемент из первого массива a меньше элемента из второго массива b, то он добавляется в третий массив c.
* В противном случае добавляется элемент из второго массива b.
* void Mergesort(double\* a, int n, double\* b) - функция для Merge Sort, в аргумент принимает указатель на массив double, int (размер массива), указатель на массив double

Принцип работы сортировки слиянием:

1. Базовый случай:

* Если размер массива равен 1, то массив уже отсортирован, и ничего не нужно делать. Это условие завершает рекурсию.

2. Рекурсивное деление:

* Функция делит массив на две части и рекурсивно вызывает “Mergesort” для каждой из них:
* Mergesort(&a[0], n / 2, &b[0]) - сортировка первой половины массива.
* Mergesort(&a[n / 2], n - (n / 2), &b[n / 2]) - сортировка второй половины массива

3. Слияние:

* После сортировки двух частей массива, функция вызывает “Merge” для их объединения:
* Merge(&a[0], n / 2, &a[n / 2], n - (n / 2), &b[0]) - слияние двух отсортированных частей в дополнительный массив “b”.

4. Копирование обратно:

* После слияния, все элементы из временного массива “b” копируются обратно в исходный массив “a”.

Подсчет операций в Merge Sort:

#ifdef check\_operations

mergechecks += 2;

#endif

* void count(double mas[], unsigned int counter[], int n, int offset) - в аргумент принимает массив double, массив без знаковых целых чисел, int (длину массива), int (). Функция подсчитывает, сколько раз встречается каждый байт в массиве чисел типа double, и вычисляет накопленные суммы для каждого байта.

Функция “count”:

* Преобразует массив чисел в массив байтов и сдвигаем на “offset”.
* Обнуляет массив “counter”.
* Подсчитывает количество каждого байта.
* Вычисляет накопленные суммы для каждого байта.
* void Radix(double a[], int n) - функция для Radix Sort, в аргумент принимает массив double, int (размер массива).

Функция “Radix” реализует алгоритм поразрядной сортировки (Radix Sort) для массива чисел типа double.

Принцип работы сортировки:

1. Инициализация:

* Создаем массив “counter” размером 256 для подсчета байтов.
* Выделяем память для временного массива “b” для хранения промежуточных результатов сортировки.
* Преобразуем указатель “a” на массив чисел в указатель “c” на массив байтов.

2. Поразрядная сортировка:

* Проходим по каждому байту числа типа double (от младшего к старшему байту).
* Для каждого байта:
* Используем функцию “count” для подсчета количества каждого байта и вычисления накопленных сумм.
* Используем массив “counter” для размещения элементов в правильные позиции в массиве “b”.
* Копируем отсортированные элементы из массива “b” обратно в массив “a”.

3. Корректировка отрицательных чисел:

* Находим индекс первого отрицательного числа в отсортированном массиве.
* Переносим отрицательные числа в начало массива “a”, инвертируя их порядок.
* Переносим положительные числа в конец массива “a”.

4. Освобождение памяти:

* Освобождаем выделенную память для временного массива “b”.

Подсчет операций в Radix Sort:

#ifdef check\_operations

radixchecks += 1;

#endif

Директива #define check\_operations (рис. 9) необходима для подсчёта операций в сортировках.

#ifdef check\_operations – проверяет был ли ранее определен макрос check\_operations. Если он определен, то компилятор включит код, который следует за этой директивой, до тех пор, пока не встретит #endif.

extern unsigned long long bubblechecks = 0; Эта строка объявляет переменную bubblechecks типа unsigned long long и инициализирует её значением 0. С остальными аналогично.

Extern - указывает, что эта переменная может быть использована в других файлах программы. Это означает, что фактическое определение переменной (возможно, с инициализацией) будет находиться в другом месте кода.

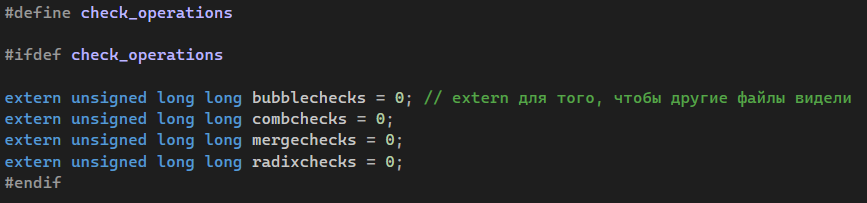


Рис. 9 Директива #define check\_operations

1.3. Текстовый файл “UsefulFunctionss.cpp” - файл со всеми остальными вспомогательными функциями.

В нём находятся заголовочные файлы, показанные на рис. 10.

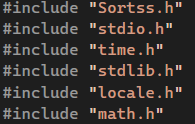


Рис.10 Заголовочные файлы в “UsefulFunctionss.cpp”

Прототипы функций:

* void swap(double& a, double& b) – функция принимает в аргумент ссылки на два элемента массива. Функция предназначена для обмена значениями двух переменных типа double. Она принимает две ссылки на переменные, меняет их значения местами.
* void CreateRandDouble(double arr[], int n, int a, int b) – функция создания случайных чисел, в аргумент принимает массив double, int (размер массива), int (минимальный элемент массива), int (максимальный элемент массива), возвращает double. Функция создает массив вещественных чисел (double) заданного размера, заполняет его случайными значениями в указанном диапазоне и возвращает указатель на этот массив.
* void OutputArr(double a[], int n) – функция вывода, в аргумент принимает массив double, int (размер массива).
* void infoBubble(int n, double arr[]) – функция с информацией для Bubble Sort, в аргумент принимает массив double, int (размер массива).
* void infoComb(int n, double arr[]) – функция с информацией для Comb Sort, в аргумент принимает массив double, int (размер массива).
* void infoMerge(int n, double arr[], double brr[]) – функция с информацией для Merge Sort, в аргумент принимает массивы double, int (размер массива). Для сортировки слиянием (Merge Sort) нужна дополнительная память - массив с таким же количеством элементов, что и изначальный (“brr” - дополнительный массив), а основной “arr”.
* void infoRadix(int n, double arr[]) – функция с информацией для Radix Sort, в аргумент принимает массив double, int (размер массива).
* void decision(int decision1, int n, double arr1[], double arr2[], double arr3[], double arr4[], double arr5[], double arr6[], double brr1[], double brr2[], double brr3[], double brr4[], double brr5[], double brr6[]) – функция для выбора и действий с конкретной сортировкой, в аргумент принимает массивы double, int (размер массива).

**2. Заголовочные файлы (повторяют функции основных файлов)**

2.1. Заголовочный файл “Sortss.h” текстового файла “Sortss.cpp”. В нём содержатся следующие функции, которые показаны на рис. 11

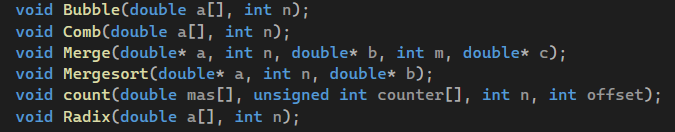


Рис. 11 Функции в заголовочном файле “Sortss.h”

2.2. Заголовочный файл “UsefulFunctionss.h” текстового файла “UsefulFunctionss.cpp”. В нём содержатся следующие функции, которые показаны на рис. 12

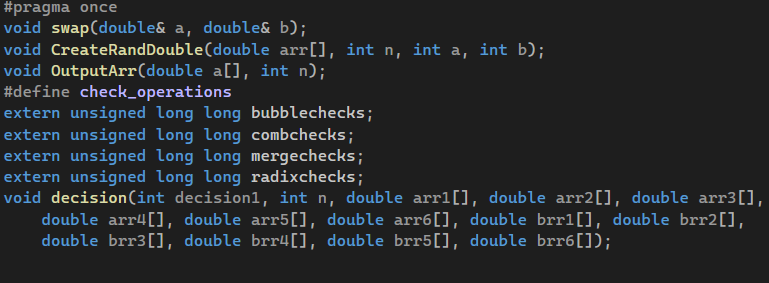


Рис.12 Функции в заголовочном файле “UsefulFunctionss.h”

# Подтверждение корректности

Для подтверждения корректности в программе после каждого запуска выводятся массивы, благодаря которым можно, действительно понять, что массив отсортирован правильно.

Для примера, числом для ввода будет “1”, чтобы всё было читабельно. Затем, соответственно, оно будет увеличенно в 21, 22, 23, 24, 25 раз больше исходного массива.

Начнём с Bubble Sort (рис. 13) (показано только для массивов, которые 21, 22, 23 больше исходного)

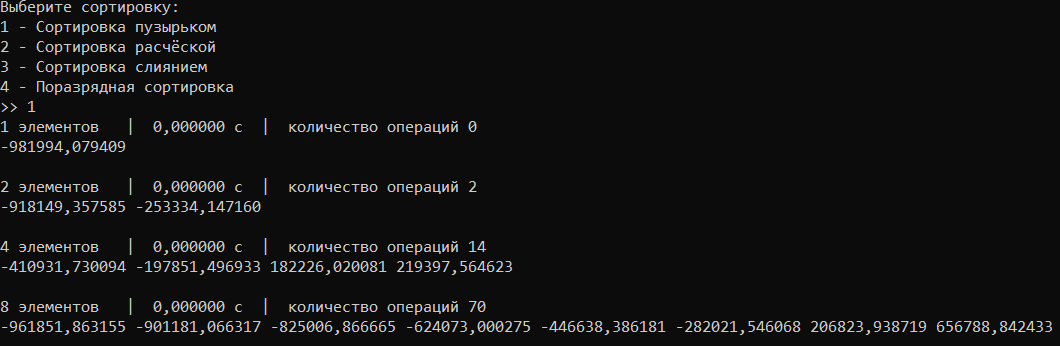


Рис. 13. Подтверждение корректности Bubble Sort

Дальше рассмотрим Comb Sort (рис. 14) (показано только для массивов, которые 21, 22, 23 больше исходного)

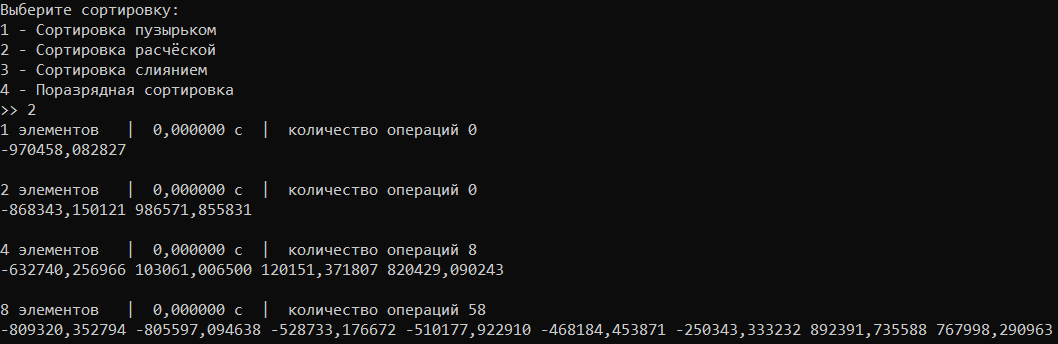


Рис 14. Подтверждение корректности Comb Sort

Рассмотрим Merge Sort (рис. 15) (показано только для массивов, которые 21, 22, 23 больше исходного)

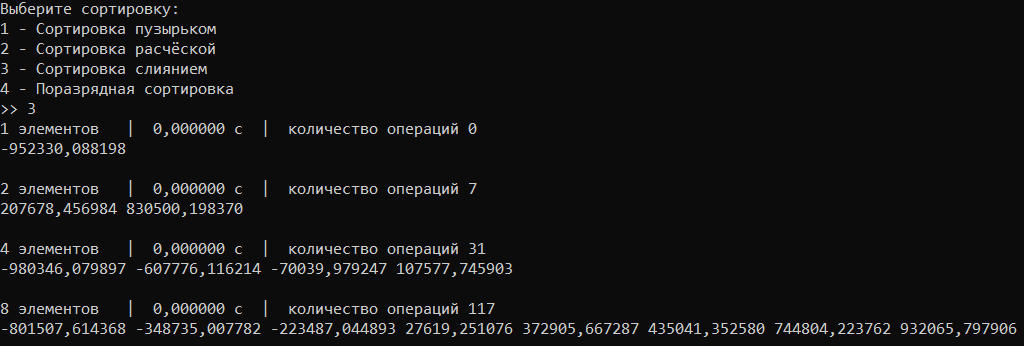


Рис. 15. Подтверждение корректности Merge Sort

Наконец, рассмотрим последнюю сортировку Radix Sort (рис. 16) (показано только для массивов, которые 21, 22, 23 больше исходного)

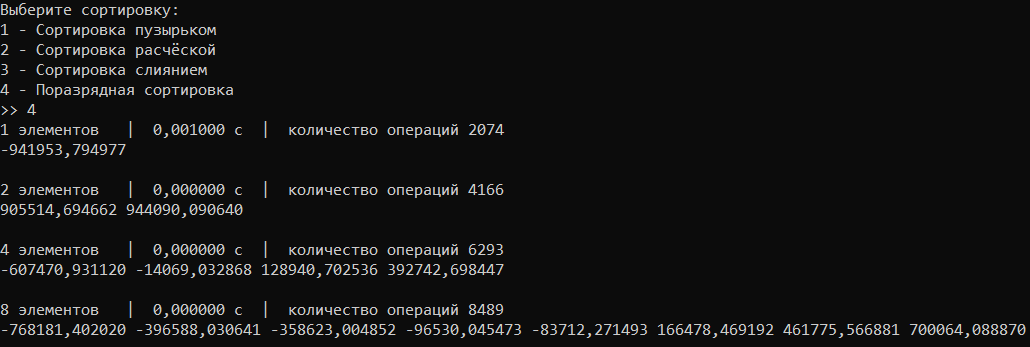


Рис. 16. Подтверждение корректности Radix Sort

# Результаты экспериментов

С целью продемонстрировать эффективность различных алгоритмов сортировки и их сложность, я решил провести испытания на шести массивах, первый из которых равен 5000, а остальные в 21, 22, 23, 24, 25 раз больше исходного. Для обеспечения объективности результатов было принято решение выполнить по три запуска каждой сортировки. После чего посчитать среднее время выполнения сортировки на каждом из шести массивов.

Для подтверждения сложности алгоритмов необходимо сравнить среднее время выполнения для массивов с различным количеством элементов, в частности, сопоставить два соседних результата.

Рассмотрим каждую.

1. Bubble Sort (Таблица 1.)

Таблица 1. Проверка эффективности Bubble Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер** | **Время**  **(1 запуск)** | **Время**  **(2 запуск)** | **Время**  **(3 запуск)** | **Среднее время** |
| 5000 | 0,11 | 0,11 | 0,08 | 0,1 |
| 10000 | 0,43 | 0,43 | 0,34 | 0,4 |
| 20000 | 1,87 | 1,95 | 1,37 | 1,73 |
| 40000 | 8,59 | 7,03 | 7,46 | 7,69 |
| 80000 | 27,37 | 26,99 | 25,76 | 26,71 |
| 160000 | 99,9 | 112,41 | 107,64 | 106,65 |

По данным таблицы видно, что размер массива увеличивается вдвое, значит время должно изменяться в 22, так как сложность сортировки .

Делаем выводы из представленных результатов средних значений, находя соотношения соседних результатов:

0,4 / 0,1 ≈ 4

1,73 / 0,4 ≈ 4,33

7,69 / 1,73 ≈ 4,45

26,71 / 7,69 ≈ 3, 47

106,65 / 26,71 ≈ 3,99

По результатам видно, что действительно сложность сортировки

2. Comb Sort (Таблица 2.)

Таблица 2. Проверка эффективности Comb Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер** | **Время**  **(1 запуск)** | **Время**  **(2 запуск)** | **Время**  **(3 запуск)** | **Среднее время** |
| 5000 | 0,001 | 0,001 | 0,001 | 0,001 |
| 10000 | 0,002 | 0,002 | 0,003 | 0,002 |
| 20000 | 0,005 | 0,004 | 0,004 | 0,004 |
| 40000 | 0,008 | 0,009 | 0,009 | 0,009 |
| 80000 | 0,017 | 0,018 | 0,019 | 0,018 |
| 160000 | 0,028 | 0,029 | 0,026 | 0,028 |

Сложность сортировки равна в лучшем случае , а в худшем N2, это значит, что увеличение времени выполнения массивов, где один больше другого колеблется от 2-х до 4-х.

Результаты:

0,002 / 0,001 ≈ 2

0,004 / 0,002 ≈ 2

0,009 / 0,004 ≈ 2,25

0,018 / 0,009 ≈ 2

0,028 / 0,018 ≈ 1,6

По результатам видно, что отношение двух соседних средних значений колеблется от 2-х до 4-х, значит сложность сортировки действительно .

3. Merge Sort (Таблица 3.)

Таблица 3. Проверка эффективности Merge Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер** | **Время**  **(1 запуск)** | **Время**  **(2 запуск)** | **Время**  **(3 запуск)** | **Среднее время** |
| 5000 | 0,001 | 0,001 | 0,001 | 0,001 |
| 10000 | 0,002 | 0,002 | 0,002 | 0,002 |
| 20000 | 0,003 | 0,004 | 0,004 | 0,004 |
| 40000 | 0,008 | 0,008 | 0,008 | 0,008 |
| 80000 | 0,016 | 0,014 | 0,015 | 0,015 |
| 160000 | 0,032 | 0,03 | 0,034 | 0,032 |

Сложность данной сортировки - . Соответственно если мы будем проверять соседние значения, то их соотношение должно быть меньше 4-х и больше или равно чем 2. Результаты:

0,002 / 0,001 ≈ 2

0,004 / 0,002 ≈ 2

0,008 / 0,004 ≈ 2

0,015 / 0,008 ≈ 1,9

0,032 / 0,015 ≈ 2,1

По результатам видно, что сложность сортировки действительно .

4. Radix Sort (Таблица 4.)

Таблица 4. Проверка эффективности Radix Sort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер** | **Время**  **(1 запуск)** | **Время**  **(2 запуск)** | **Время**  **(3 запуск)** | **Среднее время** |
| 5000 | 0,001 | 0,001 | 0,001 | 0,001 |
| 10000 | 0,001 | 0,001 | 0,001 | 0,001 |
| 20000 | 0,001 | 0,002 | 0,002 | 0,002 |
| 40000 | 0,004 | 0,004 | 0,003 | 0,004 |
| 80000 | 0,006 | 0,008 | 0,007 | 0,007 |
| 160000 | 0,013 | 0,014 | 0,014 | 0,014 |

Сложность такой сортировки - . То есть увеличение времени выполнения сортировки равно увеличению количества элементов в массиве

0,001 / 0,001 ≈1

0,002 / 0,001 ≈ 2

0,004 / 0002 ≈ 2

0,007 / 0,004 ≈ 1,75

0,014 / 0,007 ≈ 2

Из результатов видно, что сложность сортировки действительно равна .

# Заключение

В ходе лабораторной работы были проведены тестирования различных алгоритмов сортировки на массивах с разным количеством элементов. Каждая сортировка выполнялась по 3 раза на каждом массиве, и затем было вычислено среднее время выполнения для каждого случая.

На основе полученных данных можно сделать следующие выводы:

1. **Эффективность сортировок:** Различные алгоритмы сортировки показали разные времена выполнения, что позволило оценить их эффективность в зависимости от размера массива. Быстродействие некоторых сортировок явно лучше на меньших массивах, в то время как другие алгоритмы показали более стабильное поведение при увеличении размера массива.
2. **Практическая применимость:** Полученные результаты позволяют сделать рекомендации по выбору алгоритмов сортировки для различных практических задач. Например, для небольших массивов могут быть более эффективны одни алгоритмы, тогда как для больших объемов данных стоит использовать более сложные, но лучше масштабируемые сортировки.

Таким образом, проведенное тестирование и анализ подтвердили теоретические предположения о сложности и эффективности различных алгоритмов сортировки, а также предоставили практические данные для их применения в различных сценариях.

# Литература

1. Керниган Б., Ритчи Д., Фьюэр А. Язык программирования СИ //М.: Финансы и статистика. – 1992.
2. Кнут Д. Э. Искусство программирования: Сортировка и поиск. – Издательский дом Вильямс, 2000. – Т. 3.
3. Генри С. Сортировка: Введение в алгоритмы сортировки.

# Приложение

Код взят из файла “Sortss.cpp”:

#include "Sortss.h"

#include "UsefulFunctionss.h"

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

void Bubble(double a[], int n) {

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

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

#ifdef check\_operations

bubblechecks += 2; // в 13 и 14 проверка

#endif

if (a[j] > a[j + 1]) {

swap(a[j], a[j + 1]);

}

}

}

}

void Comb(double a[], int n) {

int step = n / 2;

for (; step > 1; step /= 2) {

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

for (int j = i; j + step < n; j += step) {

#ifdef check\_operations

combchecks += 2; // в 28 и 29 проверка

#endif

if (a[j] > a[j + step]) {

swap(a[j], a[j + step]);

}

}

}

}

int i = 0; bool t = true;

while (t) {

t = false;

for (int j = 0; j < n - i - 2; j++) {

#ifdef check\_operations

combchecks += 2; // в 41 и 42 проверка

#endif

if (a[j] > a[j + 1]) {

swap(a[j], a[j + 1]);

t = true;

}

}

i = i + 1;

}

}

void Merge(double\* a, int n, double\* b, int m, double\* c) {

int j = 0, k = 0;

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

if (j == n) {

#ifdef check\_operations

mergechecks += 1;

#endif

c[i] = b[k++];

}

else {

#ifdef check\_operations

mergechecks += 1;

#endif

if (k == m) {

#ifdef check\_operations

mergechecks += 1;

#endif

c[i] = a[j++];

}

else {

#ifdef check\_operations

mergechecks += 1;

#endif

if (a[j] < b[k]) {

#ifdef check\_operations

mergechecks += 1;

#endif

c[i] = a[j++];

}

else {

#ifdef check\_operations

mergechecks += 1;

#endif

c[i] = b[k++];

}

}

}

}

}

void Mergesort(double\* a, int n, double\* b) {

if (n == 1) { return; }

Mergesort(&a[0], n / 2, &b[0]);

Mergesort(&a[n / 2], n - (n / 2), &b[n / 2]);

Merge(&a[0], n / 2, &a[n / 2], n - (n / 2), &b[0]);

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

#ifdef check\_operations

mergechecks += 1;

#endif

a[i] = b[i];

}

}

void count(double mas[], unsigned int counter[], int n, int offset) {

unsigned char\* b = (unsigned char\*)mas;

b = b + offset;

memset(counter, 0, 256 \* sizeof(unsigned int));

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

counter[\*b]++;

b += sizeof(double);

}

int s = 0, tmp;

for (int i = 0; i < 256; i++) {

#ifdef check\_operations

radixchecks += 1;

#endif

tmp = counter[i];

counter[i] = s;

s += tmp;

}

}

void Radix(double a[], int n) {

unsigned int counter[256];

double\* b = (double\*)malloc(n \* sizeof(double));

unsigned char\* c = (unsigned char\*)a;

for (int i = 0; i < sizeof(double); i++) {

#ifdef check\_operations

radixchecks += 1;

#endif

count(a, counter, n, i);

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

#ifdef check\_operations

radixchecks += 1;

#endif

b[counter[\*(c + i + j \* sizeof(double))]++] = a[j];

}

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

#ifdef check\_operations

radixchecks += 1;

#endif

a[j] = b[j];

}

}

int index;

for (index = 0; index < n; index++) {

#ifdef check\_operations

radixchecks += 1;

#endif

if (b[index] < 0.0) {

break;

}

}

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

#ifdef check\_operations

radixchecks += 1;

#endif

a[i] = b[n - 1 - i];

}

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

#ifdef check\_operations

radixchecks += 1;

#endif

a[i] = b[i - n + index];

}

free(b);

}