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

**Отчёт по лабораторной работе**

**«Сортировки»**

**Выполнил**:

студент группы 3823Б1ПМ1

Болтенков С. С.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**Содержание**

[Постановка задачи 3](#_7zt7dbrq9he3)

[Метод решения 4](#_fib6j129whi9)

[Сортировка пузырьком 4](#_33lnet3tm59k)

[Сортировка Шелла 5](#_e2g0iy160kg5)

[Сортировка Слиянием 6](#_tihygcv8bphc)

[LSD-сортировка 7](#_4dnrvagze7yp)

[Руководство пользователя 8](#_1fob9te)

[Описание программной реализации 9](#_iktba7k35lxj)

[Реализация сортировки пузырьком 9](#_g49bd2t9tzvl)

[Реализация сортировки Шелла 10](#_n2rxfakfotgj)

[Реализация сортировки слиянием 11](#_w3itixyseo0t)

[Реализация LSD-сортировки 13](#_btph0yyxt2h)

[Подтверждение корректности 15](#_8heamawbqxxm)

[Результаты экспериментов 16](#_cfns0eshvdr8)

[Подтверждение сложности сортировки пузырьком 17](#_ah4303tmkz52)

[Подтверждение сложности сортировки Шелла 19](#_qe7vr8fzzr74)

[Подтверждение сложности сортировки слиянием 20](#_3zj0e3pb4uie)

[Подтверждение сложности LSD-сортировки 21](#_fxfl2vm8ckow)

[Заключение 22](#_ppwz2d6weh14)

[Список литературы 23](#_24vyboa6d4g3)

[Приложение 24](#_ns12nycut1nq)

# **Постановка задачи**

В данной лабораторной работе были реализованы следующие алгоритмы сортировок для типа данных float на языке программирования С [5]:

1. пузырьковая сортировка;
2. сортировка Шелла;
3. сортировка слиянием;
4. LSD-сортировка;

Также была выполнена проверка асимптотической корректности алгоритма.

# **Метод решения**

В данной главе будут рассмотрены реализации алгоритмов сортировок со всеми их тонкостями и нюансами [1].

## **Сортировка пузырьком**

Одним из самых простых алгоритмов сортировки - является пузырьковая сортировка. В процессе работы она выполняет проходы по массиву и обменивает местами соседние элементы, нарушающие порядок. Если мысленно прокрутить этот процесс в голове становится очевидно, почему этот алгоритм получил такое название. Пусть мы продвигаемся по массиву справа налево. Когда на первом проходе встречается минимальный элемент, он обменивается местами со всеми и попадает в начало списка. После второго прохода второй минимум оказывается на второй позиции и т. д. Другими словами, маленькие элементы всплывают, а большие тонут при каждом проходе.

Чтобы полностью отсортировать массив потребуется N проходов, где N - количество элементов массива, т. к. за один проход на своё место гарантировано может попасть только один элемент. А каждый такой проход в свою очередь работает за . Следовательно, сложность пузырьковой сортировки составляет .

На практике пузырьковую сортировку можно не асимптотически улучшить:

1. Каждый раз выполнять проходы не по всей длине массива, а только по его неотсортированной части.
2. Считать алгоритм завершившимся, если за проход не было совершено хотя бы одной перестановки элементов.

## 

## **Сортировка Шелла**

Алгоритм сортировки Шелла представляет собой модификацию алгоритма сортировки вставками. Суть алгоритма состоит в том, что он сначала сортирует элементы, находящиеся находящиеся на определенном расстоянии друг от друга, а затем уменьшает это расстояние и повторяет процесс сортировки.

Перед началом работы алгоритма выбирается последовательность промежутков, по которым будет производиться сортировка. После того как последовательность промежутков выбрана, для каждого из них выполняется сортировка вставками. Затем этот процесс повторяется с уменьшением длины промежутка на каждом шаге, и так далее, до того момента, пока его длина не станет равной единице.

В результате остаётся только промежуток длины один, который отсортировывается при помощи обычной сортировки вставками. Таким образом происходит постепенная сортировка всего массива, начиная с больших интервалов и заканчивая малыми.

Сортировка Шелла признана эффективной, поскольку время её работы в среднем и она сочетает в себе преимущества сортировки вставками (устойчивость к предварительной отсортированности) и сортировки пузырьком (эффективное сокращение количества инверсий среди соседних элементов при каждом сравнении).

## 

## **Сортировка Слиянием**

Алгоритм сортировки слиянием (Merge Sort) основан на принципе "разделяй и властвуй". Алгоритм состоит из двух основных этапов: разделения массива на подмассивы и последующего слияния этих подмассивов.

Первый этап включает рекурсивное разбиение исходного массива пополам до тех пор, пока не будет достигнута последовательность из одного элемента.

После этого происходит этап слияния, в ходе которого подмассивы объединяются в результирующий массив. Процесс слияния осуществляется путем сравнения элементов из различных подмассивов и объединения их в упорядоченном порядке.

Основное преимущество сортировки слиянием заключается в том, что она гарантированно работает за время в худшем, лучшем и среднем случаях, что делает ее одним из наиболее эффективных алгоритмов сортировки.

Таким образом, сортировка слиянием является надежным и эффективным алгоритмом для упорядочивания больших объемов данных. Она часто используется в различных областях, включая разработку программного обеспечения, обработку данных и теоретическую информатику.

## **LSD-сортировка**

LSD (Least Significant Digit) сортировка [4] — это алгоритм сортировки, который реализует поразрядную сортировку типов данных. Он начинает с младших разрядов (least significant digits) и постепенно сдвигается к старшим разрядам.

Алгоритм LSD сортировки проходит через список чисел (или строк), начиная с самого правого разряда. Он сортирует числа в порядке возрастания или строки в лексикографическом порядке, используя значения их байт.

Процесс LSD сортировки продолжается до тех пор, пока все разряды не будут упорядочены. Алгоритм обычно выполняется с использованием вспомогательных массивов и буферов, чтобы отсортировать элементы на каждом разряде.

LSD сортировка является стабильной сортировкой, что означает, что она сохраняет относительный порядок равных элементов. Время выполнения составляет , где k - количество байт в типе данных.

# **Руководство пользователя**

При запуске программы вам необходимо ввести два значения. Первое - это номер алгоритма сортировки, который вы хотите протестировать. Доступно 4 алгоритма:

1. 0 - пузырьковая сортировка
2. 1 - сортировка Шелла
3. 2 - сортировка слиянием
4. 3 - LSD сортировка

И второе - это значение 0 или 1. 1 в том случае, если вы хотите, чтобы выполнялась проверка корректности работы алгоритма сортировки, после упорядочивания данных, и 0 в противном случае.

После окончания ввода, программа выведет название выбранной сортировки и начнет тестирование алгоритма на различных объемах данных (). После каждого выполнения выводится затраченное время в секундах. В случае, если была подключена проверка корректности, и алгоритм неверно отсортировал данные вы увидите сообщение “sort incorrect”, и программа завершит своё выполнение. Если алгоритм сортировки работает верно в конце вы увидите “sort correct”.

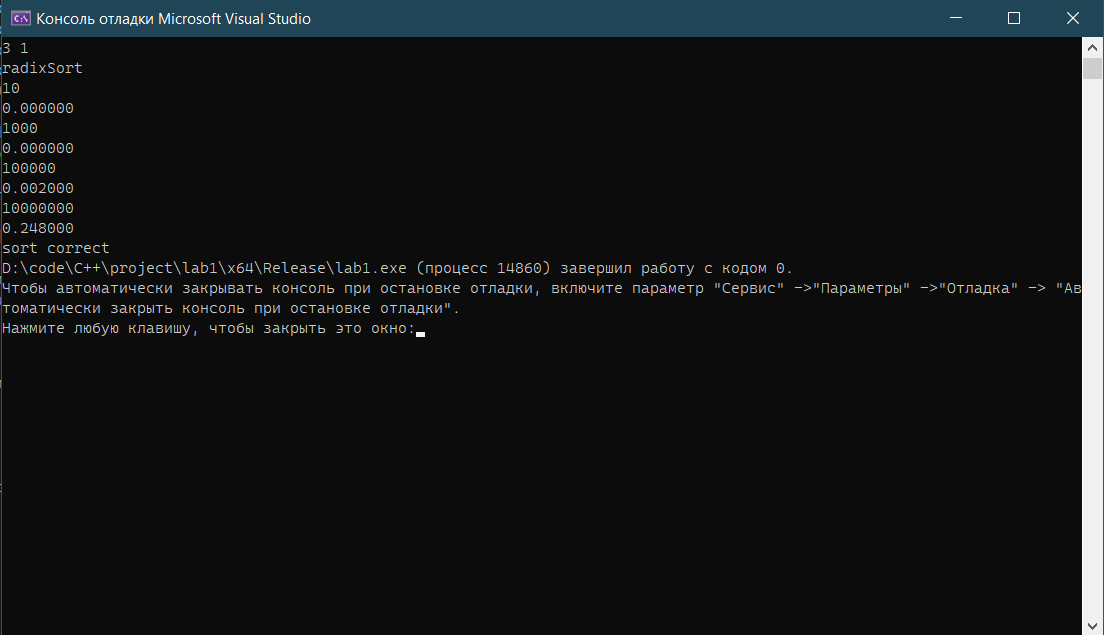


Рисунок 1. Пример работы программы

# **Описание программной реализации**

В данной главе будут рассмотрены программные реализации алгоритмов сортировок .

## **Реализация сортировки пузырьком**

Ниже будет привёдён псевдокод алгоритма пузырьковой сортировки и описание его работы.

1 процедура bubbleSort(массив a):

2

3 for(по i от 0 до n - 1):

4

5 for(по j от n - 1 до i):

6 если a[j - 1] > a[j], то меняем местами a[j - 1] 7 и a[j]

8

9 если не было хотя бы одной перестановки, то массив 10 уже отсортирован, завершаем сортировку

Описание работы алгоритма:

1. Процедура bubbleSort принимает на вход массив.
2. n - переменная, хранящая длину массива.
3. Двойной цикл for используется для прохода по массиву и перестановки элементов. Внешний цикл отвечает за количество проходов по массиву, а внутренний - за выполнение сравнений и перестановок.
4. Условие на строке 6 сравнивает текущий элемент с предыдущим и, если они находятся в неправильном порядке, выполняется перестановка.

## **Реализация сортировки Шелла**

Ниже будет привёдён псевдокод сортировки Шелла и описание его работы.

1 процедура shellSort(массив a):

2

3 for(по step от 1 до (n - 1) / 9): //1 4 13 40 121 364 1093

4 step = step + 3 \* step + 1

5

6 while (step > 0), то):

7

8 for(по i от step до n):

9 j = i

10 while(j >= step и a[j] < a[j - step]):

11 меняем местами a[j] и a[j - step]

12

13 step = step / 3

Описание работы алгоритма:

1. Процедура shellSort принимает на вход массив.
2. n - переменная, хранящая длину массива.
3. step - переменная хранящая длину сортируемой подпоследовательности.
4. В цикле for на 3 строке предподсчитываются длины промежутков.
5. Цикл while на 6 строке используется для сортировки промежутков длины step.
6. Внутренний цикл - это сортировка выбором с шагом step (строки 8 - 11).

## **Реализация сортировки слиянием**

Ниже будет привёдён псевдокод сортировки Шелла и описание его работы.

1 процедура meregeSort(массив a):

2

3 копируем все данные из массива a в b

4 for(по step от 1 до n):

5

6 for(по i от 0 до n):

7 merege(подмассив1, подмассив2, b)

8 i = i + 2 \* step

9

10 меняем местами a и б

11 step = step \* 2

12

13 если отсортированный массив лежит в b, то меняем их 14 местами и копируем данные из b в a

15

16

17 процедура merege(массив a, массив b, массив res):

18

19 for(по ir от 0 от an + bn):

20 если ia равно an, то:

21 res[ir] = b[ib]

22 ib = ib + 1

23 continue

24

25 если ib равно bn, то:

26 res[ir] = a[ia]

27 ia = ia + 1

28 continue

29

30 eсли b[ib] > a[ia], то:

31 res[ir] = a[ia]

32 ia = ia + 1

33 иначе:

34 res[ir] = b[ib]

35 ib = ib + 1

Описание работы алгоритма meregeSort:

1. Процедура meregeSor tпринимает на вход массив.
2. n - переменная, хранящая длину массива.
3. step - переменная, хранящая длину подмассива
4. Внешний цикл на строке 4 проходит по длине подмассивов, а внутренний цикл на 6 строке вызывает функцию объединения для всех подмассивов длины step

Описание работы алгоритма merege:

1. Процедура merege принимает два подмассива и результирующий массив, в котором они будут объединены
2. an и bn - переменные, хранящие длины подмассивов
3. ia, ib, ir - переменные, хранящие индексы массивов
4. В цикле for на 19 строке объедняются два подмассива в третий

## **Реализация LSD-сортировки**

Ниже будет привёдён псевдокод сортировки Шелла и описание его работы.

1 процедура radixSort(массив a):

2

3 for(по d от 0 до количества байт типа данных):

4

5 for (по j от 0 до 257):

6 count[j] = 0

7 for (по i от 0 i до n):

8 count[(значение байта d переменной a[i]) + 1] += 1

9 for (по j от 1 до 257):

10 count[j] += count[j - 1]

11 for (по i от 0 до n):

12 b[count[значение байта d переменной a[i]]] = a[i]

13 count[значение байта d переменной a[i]] += 1

14 for (по i от 0 до n):

15 a[i] = b[i];

Описание работы алгоритма radixSort:

1. Процедура shellSort принимает на вход массив.
2. n - переменная, хранящая длину массива.
3. count - массив, который используется для распределяющего подсчёта
4. b - массив, в который отсортировано распределяются данные из a по массиву count
5. в цикле for на 5 строке зануляем массив count
6. в цикле for на 7 строке подсчитываем количество байт определённого значения
7. в цикле for на 9 строке считаем массив префикс-сумм по массиву count
8. в цикле for на 11 строке переносим отсортированные данные из массива a в массив b (с помощью метода распределяющего подсчёта)
9. в цикле for на 14 строке отсортированные данные из b копируются в a

# **Подтверждение корректности**

В данной главе будет рассказано, какие методы использовались для проверки корректности реализованных алгоритмов.

Первый метод тестирования - проверка на случайных данных. Все алгоритмы были проверены на случайно сгенерированных данных различного объёма. Для проверки правильности выполнения алгоритма применялось сравнение с функцией из стандартной библиотеки языка C - qsort(). Результаты выполнения написанной сортировки сравнивались с результатом стандартной функции.

В итоге проверки, можно сделать вывод, что все алгоритмы корректно сортируют данные.

# **Результаты экспериментов**

В данной главе будет приведен пример работы реализованных сортировок для типа данных float на различных объёмах данных. Также рассмотрим график зависимости времемя выполнения сортировок от объема входных данных и докажем их асимптотическую сложность.

Начнём с алгоритма сортировки пузырьком. Как видим из рисунка ниже алгоритм работает корректно.

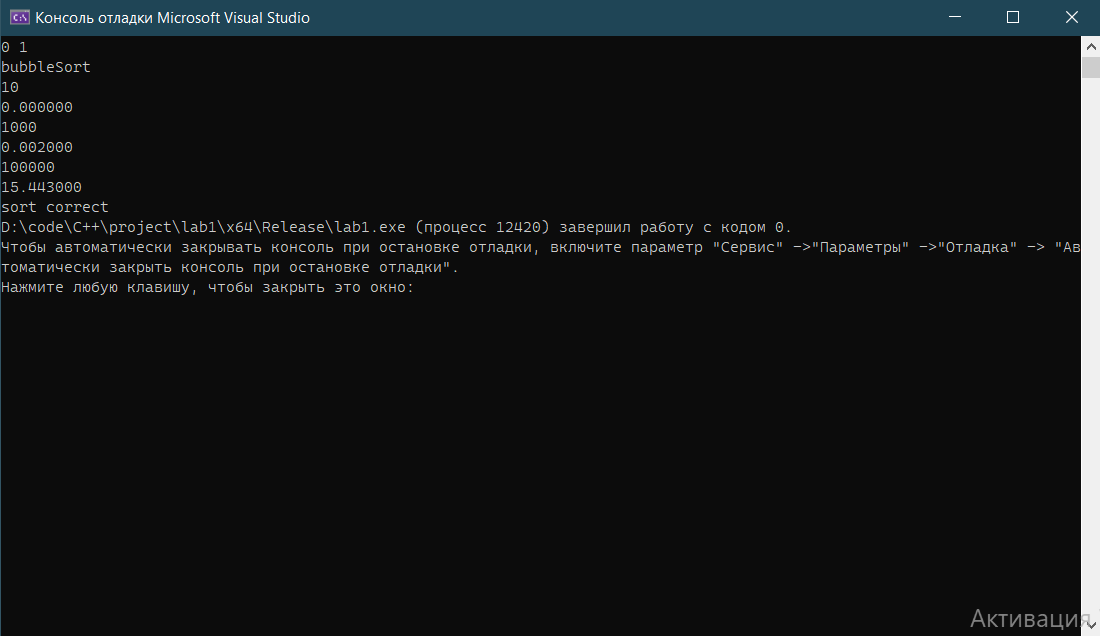


Рисунок 2. Результат работы программы для сортировки пузырьком с проверкой корректности

Аналогичные тесты были проведены и для всех остальных алгоритмов сортировок, все отработали корректно.

Рассмотрим полученные результаты тестирования. На графике ниже приведена зависимость времени выполнения от объёма входных данных реализованных сортировок. Из графика наглядно видно, насколько пузырьковая сортировка работает медленней по сравнению с логарифмическими или линейной. При больших объёмах данных разница существенно увеличивается из-за чего их сравнеие затрудняется. Например, в то время, как пузырёк работает десятки секунд, LSD-сортировка отработает за доли секунд.

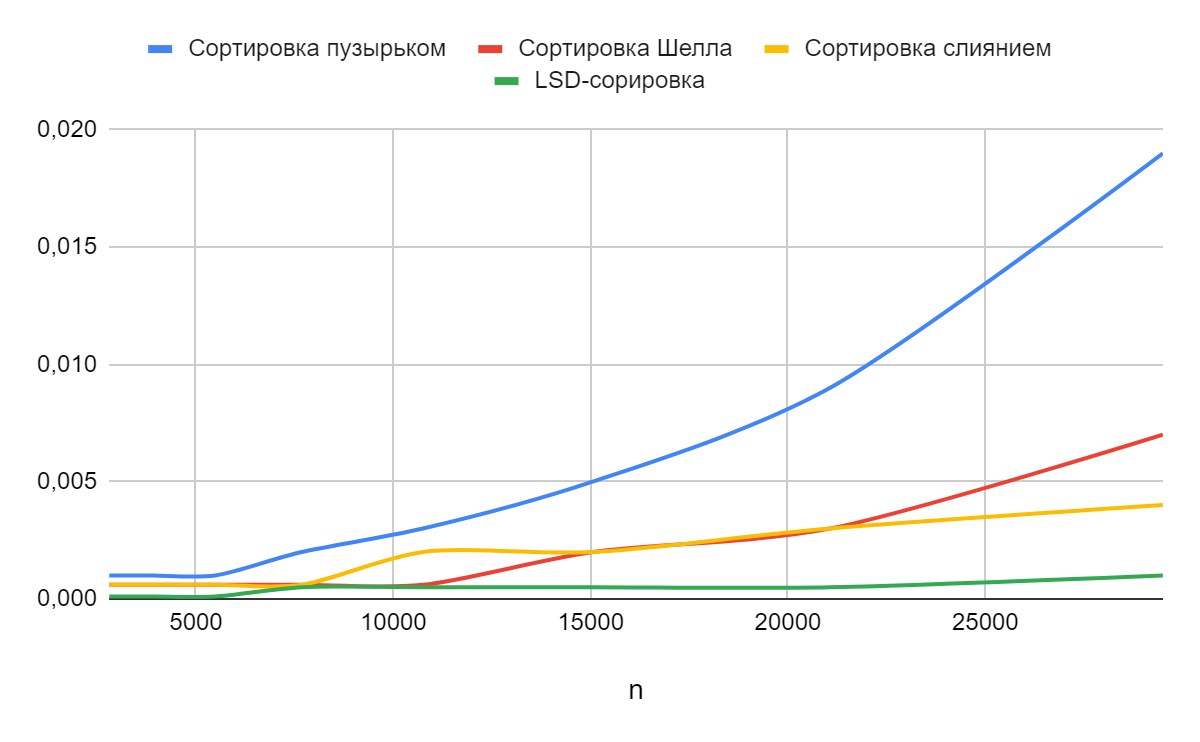


Рисунок 3. График зависимость времени выполнения от объёма входных данных.

На первом месте по скорости выполнения LSD-сортировка, на втором сортировка слиянием, на третьем сортировка Шелла и на последнем месте сортировка пузырьком.

Теперь докажем асимптотическую сложность каждого из алгоритма. В качестве доказательства будет приводиться график отношения времени выполнения (t) программы к предполагаемой сложности. Если предположение будет верно, то график функции должен быть близок к некоторой константе.

## **Подтверждение сложности сортировки пузырьком**

Для пузырьковой сортировки мы предполагали, что сложность алгоритма . Как видно на графике, предположение верно, потому что данная зависимость стремится к некоторой константе.

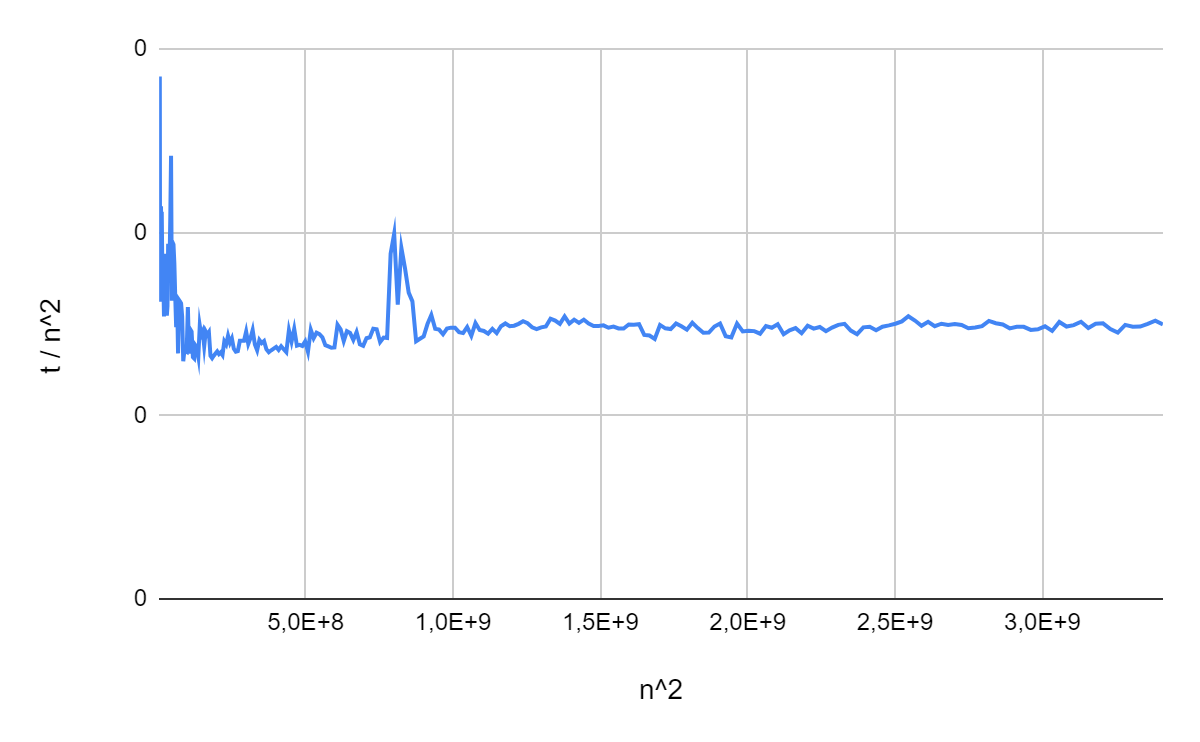


Рисунок 4. График отношения времени выполнения к предполагаемой сложности для сортировки пузырьком

## **Подтверждение сложности сортировки Шелла**

Для сортировки Шелла мы предполагали, что сложность алгоритма . Как видно на графике, предположение верно, потому что данная зависимость стремится к некоторой константе.

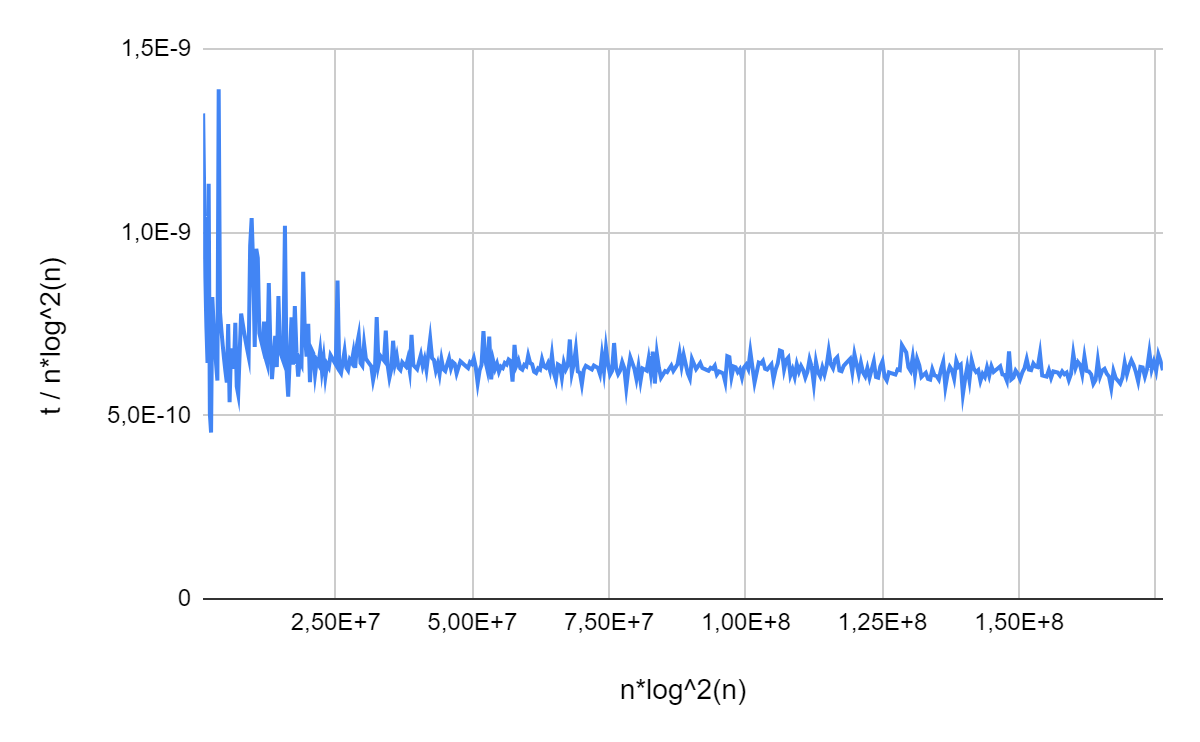
****

Рисунок 5. График отношения времени выполнения к предполагаемой сложности для сортировки Шелла

## **Подтверждение сложности сортировки слиянием**

Для сортировки слиянием мы предполагали, что сложность алгоритма . Как видно на графике, предположение верно, потому что данная зависимость стремится к некоторой константе.

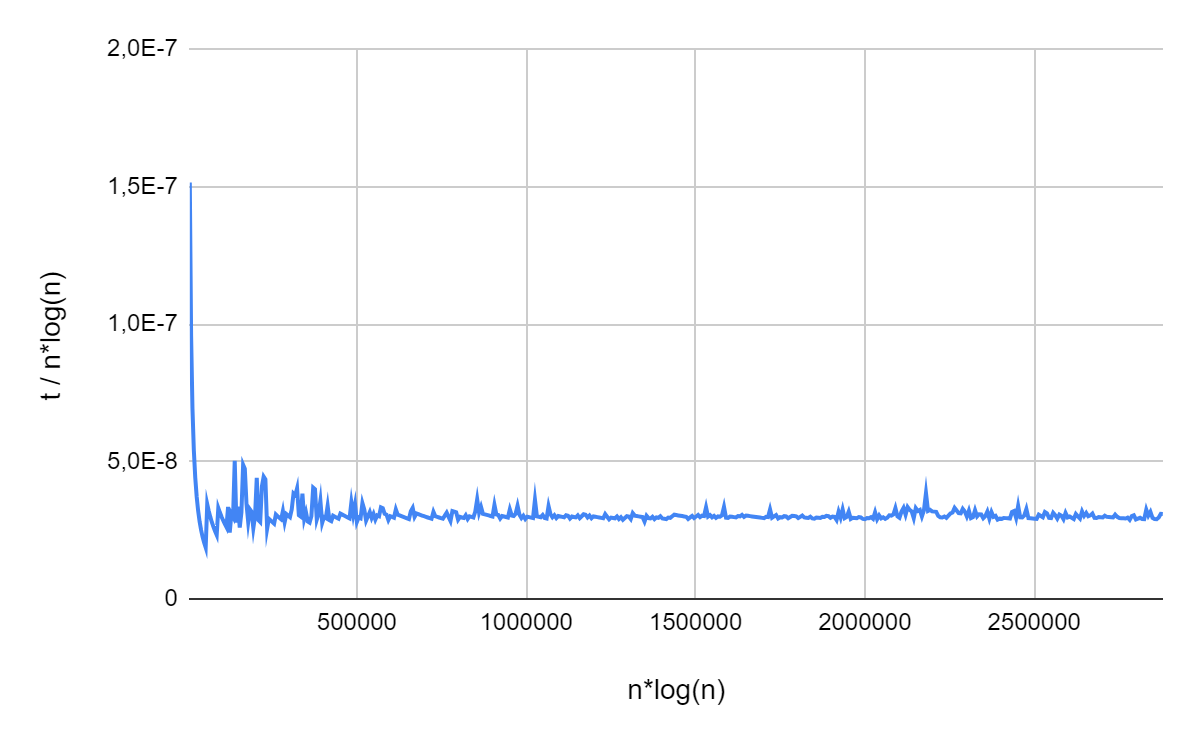
****

Рисунок 6. График отношения времени выполнения к предполагаемой сложности для сортировки Слиянием

## **Подтверждение сложности LSD-сортировки**

Для LSD-сортировки мы предполагали, что сложность алгоритма , где k - количество байт в типе данных. Как видно на графике, предположение верно, потому что данная зависимость стремится к некоторой константе.

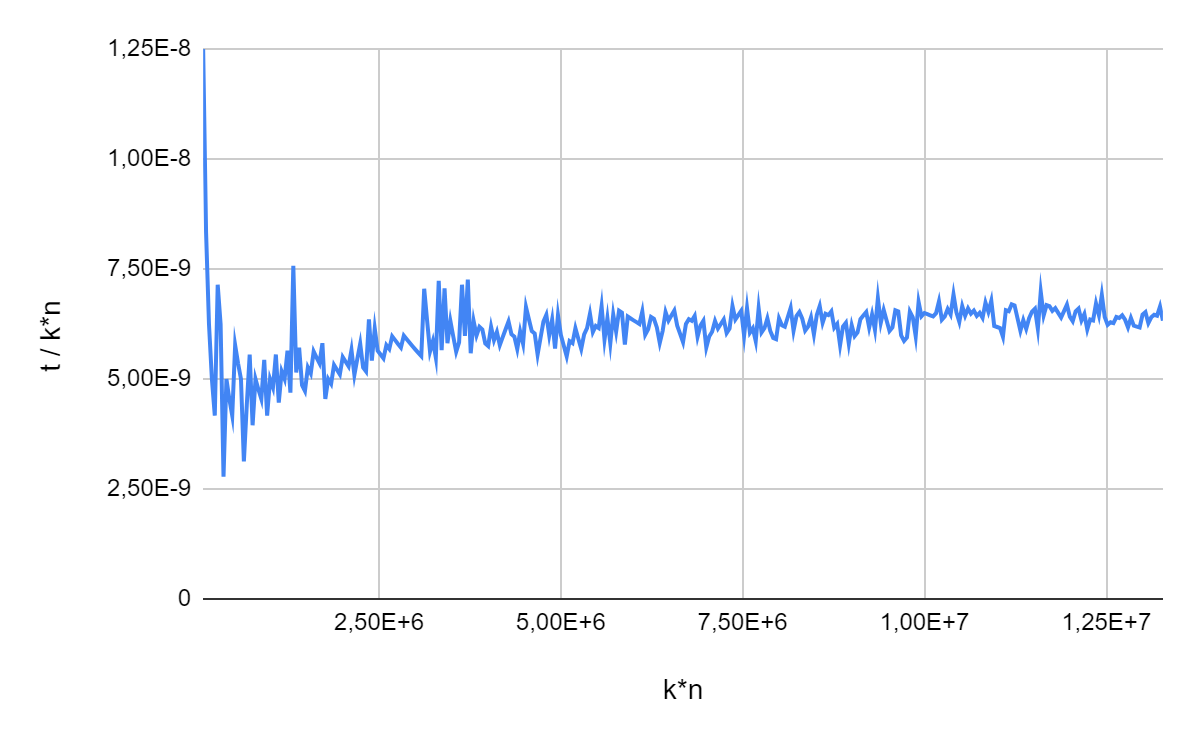


Рисунок 7. График отношения времени выполнения к предполагаемой сложности для LSD-сортировки

Подводя итог экспериментов, можно сделать вывод, что на практике алгоритмы отработали так, как и ожидалось в теории. Асимптотическая сложность также совпала с предполагаемой.

# **Заключение**

Целью лобараторной работы было написание четырёх алгоритмов сортировок: пузырьком, Шелла, слиянием и LSD и дальнейшее их исследование с оценкой асимптотики.

В итоге проделанной работы получилось реализовать все алгоритмы сортировок. Также удалось строго доказать асимптотическую сложность каждого из них. В ходе тестирования получилось наглядно увидеть отличие времени работы алгоритмов сортировок.

В итоге проделанной работы можно сделать следующие выводы:

1. Пузырьковая сортировка подходит только для сортировки сравнительно небольших объёмов данных. В таком случае время её выполнения практически не отличается от логарифмических, но реализация гораздо проще.
2. Значительное преимущество сортировки Шелла по сравнению с сортировкой слиянием и LSD является то, что она не требует дополнительной памяти.
3. Если требуется сортировать много данных, которые имеют небольшое количество байт, лучше всего использовать LSD-сортировку.

В ходе выполнения данного проекта мною было приобретено множество новых практических навыков, реализация каждой сортировки дала глубокое понимание того, как она устроена. Также был получен опыт сравнения алгоритмов сортировок и практический опыт оценки и доказательства их асимптотики. Полученные опыт и знания будут крайне важны в будущем, они станут фундаментом для исследования уже более сложных вещей.

# **Список литературы**

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., & Штайн, К. (2009). "Introduction to Algorithms." MIT Press. ISBN 978-0-262-03384-8
2. Седжвик, Р., & Уэйн, К. (2011). "Algorithms." Addison-Wesley Professional. ISBN 032157351X
3. Knuth, D. (1998). "The Art of Computer Programming, Volume 3: Sorting and Searching." Addison-Wesley Professional. ISBN 978-0201896855
4. McIlroy, M. (1993). "Engineering Radix Sort." Computing Systems, Vol. 6, No. 1, pp. 5-27.
5. Sedgewick, R. (2014). "Algorithms, 4th Edition." Addison-Wesley Professional. ISBN 032157351X

# **Приложение**

void bubbleSort(float\* a, int n)

{

int i, j, flag;

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

{

flag = 1;

for (j = n - 1; j > i; j--)

{

if (a[j] < a[j - 1])

{

std::swap(a[j], a[j - 1]);

flag = 0;

}

}

if (flag) break;

}

return;

}

void shellSort(float\* a, int n)

{

int step, i, j;

for (step = 1; step < (n - 1) / 9; step = 3 \* step + 1); //1 4 13 40 121 364 1093

for (; step > 0; step /= 3)

{

for (i = step; i < n; i++)

{

for (j = i; j >= step && a[j] < a[j - step]; j -= step)

std::swap(a[j], a[j - step]);

}

}

return;

}

uchar getByte(float\* item, int numberByte)

{

return ((uchar\*)item)[numberByte];

}

void radixSort(float\* a, int n)

{

float\* b;

int\* count, d, i, j, mid;

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

count = (int\*)malloc(257 \* sizeof(int));

for (d = 0; d < sizeof(float); d++)

{

for (j = 0; j < 257; j++) count[j] = 0;

for (i = 0; i < n; i++) count[getByte(&a[i], d) + 1]++;

for (j = 1; j < 257; j++) count[j] += count[j - 1];

for (i = 0; i < n; i++) b[count[getByte(&a[i], d)]++] = a[i];

for (i = 0; i < n; i++) a[i] = b[i];

}

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

{

if (a[i] < 0)

{

mid = i;

break;

}

}

j = 0;

for (i = n - 1; i >= mid; i--) a[j++] = b[i];

for (i = 0; i < mid; i++) a[j++] = b[i];

free(count);

free(b);

return;

}

void merege(float\* a, float\* b, float\* res, int an, int bn)

{

int ia = 0, ib = 0, ir;

for (ir = 0; ir < an + bn; ir++)

{

if (ia == an) { res[ir] = b[ib++]; continue; }

if (ib == bn) { res[ir] = a[ia++]; continue; }

res[ir] = (b[ib] > a[ia]) ? a[ia++] : b[ib++];

}

return;

}

void meregeSort(float\* a, int n)

{

float\* b;

int step, i, k;

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

step = 1;

k = 0;

for (i = 0; i < n; i++) b[i] = a[i];

for (; step < n; step \*= 2)

{

for (i = 0; i < n; i += 2 \* step)

{

merege(a + i, a + i + step, b + i, std::max(0, std::min(step, n - i)), std::max(0, std::min(step, n - step - i)));

}

std::swap(a, b);

k++;

}

if (k % 2)

{

std::swap(a, b);

for (i = 0; i < n; i++) a[i] = b[i];

}

free(b);

return;

}