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»**

**Выполнила**:

студентка группы 3823Б1ПМ1

Булычева Л. С.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**Содержание**

[Постановка задачи 3](#__RefHeading___1)

[Метод решения 4](#__RefHeading___2)

[Руководство пользователя 5](#__RefHeading___3)

[Описание программной реализации 6](#__RefHeading___4)

[Подтверждение корректности 10](#__RefHeading___5)

[Результаты экспериментов 11](#__RefHeading___6)

[Заключение 14](#__RefHeading___7)

[Список литературы 15](#__RefHeading___8)

[Приложение 16](#__RefHeading___9)

# **Постановка задачи**

В данной работе необходимо реализовать 4 сортировки [1] для типа данных double с плавающей точкой:

1. Сортировка вставками
2. Сортировка расчёской
3. Сортировка слиянием
4. Поразрядная сортировка

Также реализованные алгоритмы необходимо проверить на корректность и доказать, что ожидаемая асимптотическая сложность совпадает с действительной.

# **Метод решения**

Описание реализованных алгоритмов:

1. **Сортировка вставками(Insertion sort).** Алгоритмработает путем пошагового сравнения элементов, переставляя их в нужное место в уже отсортированной части массива.

Первым шагом алгоритм берет первый элемент массива (предполагается, что подмассив из одного элемента уже отсортирован), далее по одному элементу добавляет в отсортированную часть массива на определенное место. При этом новый элемент сравнивается с элементами, уже находящимися слева от него, и его позиция корректируется, пока не будет найдено его правильное место. Асимптотическая сложность данного алгоритма .

1. **Сортировка расчёской(Comb Sort).** Алгоритм является модификацией алгоритма сортировки пузырьком и был разработан для более быстрой сортировки массивов. Он получил свое название из-за механизма "расчесывания" элементов массива.

На первом шаге выбирается начальная длина шага (например, длина массива n). Далее элементы массива сравниваются и обмениваются на этом расстоянии. Затем шаг уменьшается по определенному коэффициенту (~1.3) для продолжения сравнения и обмена. Процесс повторяется до тех пор, пока шаг не станет равным 1, после чего алгоритм завершает работу с окончательной проверкой. Асимптотика сортировки расчёской - .

1. **Сортировка Слиянием(Merge sort).** Алгоритм сначала разбивает массив на n (где n - длина массива) отсортированных массивов длины 1, а затем происходит процесс их слияния в n/2 отсортированных массивов длины 2, затем в n/4 отсортированных массивов длины 4 и т.д. до тех пор, пока не останется один отсортированный массив длины n. Асимптотика сортировки слиянием - .
2. **Поразрядная сортировка (Radix sort).** Данный алгоритм сортировки выполняет побайтовую сортировку данных, начиная с их младших разрядов и переходя к старшим шаг за шагом. Процесс поразрядной 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. Время выполнения составляет , где k - количество байт в типе данных.

# **Руководство пользователя**

Взаимодействие с программой осуществляется через консоль. При запуске пользователю необходимо ввести одно значение - длину массива, для которых он хочет проверить алгоритмы сортировок. Далее программа случайным образом генерирует необходимые данные и выведет их пользователю на экран. Далее поочерёдно будут запускаться алгоритмы сортировок и выводиться данные на консоль о том, какая сортировка отработала и за какое время в секундах. Также после завершения каждого алгоритма на экран будет выводиться отсортированный массив, если алгоритм отработал верно. А в случае ошибки пользователь увидит какая сортировка отработала не корректно.

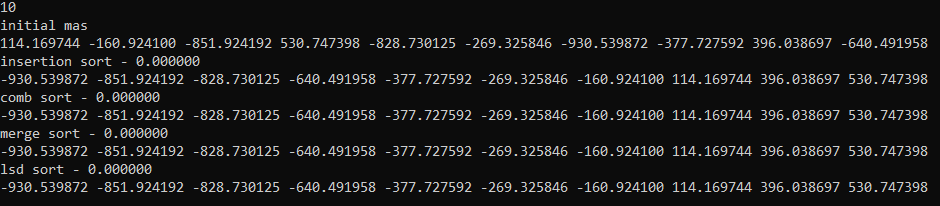


Рис. 1. Пример работы программы для n = 10.

# **Описание программной реализации**

В папке проекта находится файл main.cpp, в котором реализованы все алгоритмы сортировок и вспомогательные функции. Рассмотрим наиболее важные и интересные из них:

1. void insertionSort(double\* mas, int size); - сортировка вставками
2. void combSort(double\* mas, int size); - сортировка расчёской
3. void mergeSort(double\* mas, int size); - сортировка слиянием
   1. void merge(double\* masa, double\* masb, double\* res, int sizea, int sizeb); - вспомогательный метод для сортировки слиянием, объединяет два отсортированных массива в один, сохраняя отсортированность.
4. void lsdSort(double\* mas, int size); - поразрядная сортировка
   1. void foffset(unsigned char\* mas, int size, int offset, int count[257]); - вспомогательный метод для поразрядной сортировки, считает префикс-сумму по байтам(для метода распределяющего подсчета).

**Сортировка слиянием**

insertionSort(mas, size) {

for (i = 1; i < size; i = i + 1)

for (j = i; j > 0; j--)

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

swap(mas[j], mas[j - 1]);

}

Процедура принимает два аргумента - массив, который необходимо отсортировать и его длину. Первый цикл for по i проходит по неотсортированной части массива. Второй цикл for по j проходит уже по отсортированной части, находя подходящую позицию для нового элемента.

**Сортировка расчёской**

combSort(mas, size) {

step = size - 1;

factor = 1.2473309;

while (step >= 1) {

for (j = 0; j < size - step; j++) {

if (mas[j] > mas[j + step])

std::swap(mas[j], mas[j + step]);

}

step = step / factor;

}

}

Процедура принимает два аргумента - массив, который необходимо отсортировать и его длину. Переменная step хранит длину текущего шага, по которому идёт сортировка. factor - это переменная, которая хранит величину, на которую уменьшается шаг на каждой итерации. Цикл while продолжается до тех пор, пока длина шага более или равна 1. В теле цикла while находится for по j, который обменивает элементы массива на соответствующим растоянии step, если они нарушают порядок. На последней итерации, когда шаг будет равен 1, будет выполнен финальный проход по массиву с обменом соседних элементов.

**Сортировка слиянием**

void mergeSort(mas, size) {

tmpdata;

count = 0;

for (step = 1; step < size; step = step \* 2) {

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

merge(mas + i, mas + i + step, tmpdata + i, sizea, sizeb);

std::swap(mas, tmpdata);

count++;

}

if (count % 2 == 1)

{

std::swap(mas, tmpdata);

for (i = 0; i < size; i++) mas[i] = tmpdata[i];

}

}

Процедура принимает два аргумента - массив, который необходимо отсортировать и его длину. Переменная tmpdata - это вспомогательный массив для сортировки, в который временно будут копироваться данные. Переменная step хранит текущую длину подмассива. И переменная count будет использоваться для подсчета количества итераций, чтобы потом определить в каком из массивов хранятся отсортированные данные. Цикл for по step - это цикл по длине подмассивов, вложенный цикл for по i вызывает процедуру слияния для всех пар подмассивов длины step. После завершения сортировки проверяем count, если это число нечетное, значит, было совершено нечётное число шагов в первом for по step и отсортированные данные лежат в tmpdata и их нужно скопировать в mas.

void merge(masa, masb, res, sizea, sizeb) {

i = j = r = 0;

while (sizea > i and sizeb > j)

{

if (masa[i] < masb[j]) {

res[r] = masa[i];

r = r + 1;

i = i + 1;

}

else {

res[r] = masb[j];

r = r + 1;

j = j + 1;

}

}

for (; i < sizea; i = i + 1) {

res[r] = masa[i];

r = r + 1;

}

for (; j < sizeb; j = j + 1) {

res[r] = masb[j];

r = r + 1;

}

}

Процедура принимает на вход три массива, два массива masa и masb длины sizea и sizeb соответственно - это подмассивы, которые необходимо объединить, массив res - это массив, в который будет записан результат, гарантируется, что его длина не менее sizea + sizeb. Переменные i и j - текущие индексы в подмассивах, r - текущий индекс в результирующем массиве. На первом этапе слияния в цикле while идём, пока не дойдём до конца первого или второго подмассива. В теле этого цикла выполняем сравнение элементов массивов под индексами i и j, и записываем нужный из них в массив res под индексом r. После завершения цикла while, возможно, какой-то из подмассивов не записан до конца в результирующий массив, поэтому в двух последних циклах for переносим оставшиеся элементы из первого или второго массива в res.

**Поразрядная сортировка**

void lsdSort(mas, size) {

count[257];

tmpdata;

pm = (char)mas;

for (i = 0; i < sizeof(double); i = i + 1) {

foffset(pm, size, i, count);

for (j = 0; j < size; j = j + 1)

tmpdata[count[pm[j \* sizeof(double) + i]]++] = mas[j];

for (j = 0; j < size; j = j + 1)

mas[j] = tmpdata[j];

}

j = size - 1;

i = 0;

while (tmpdata[j] < 0) mas[i++] = tmpdata[j--];

j = 0;

while (tmpdata[j] > 0) mas[i++] = tmpdata[j++];

}

Процедура принимает два аргумента - массив, который необходимо отсортировать и его длину. Статический массив count будет хранить префикс-сумму по значениям байт. Переменная tmpdata - это вспомогательный массив для сортировки, в который временно будут копироваться данные. Массив pm - это ссылка на исходный массив, которая воспринимает переменные double, как char, чтобы к байтам double можно было удобно обращаться по индексу. Первый цикл for по i идёт по количеству байт в типе данных, в данном случае по количеству байт типа данных double. Метод foffset заполняет массив count. Далее во вложенном цикле for по j применяется метод распределяющего подсчета, чтобы данные из исходного массива сразу расположить на необходимое место в tmpdata. В следующем цикле for по j данные из временного массива копируются в основной. После завершения основного цикла for, данные отсортированы по байтам, но не отсортированы, как double. В последних двух циклах while переносим все элементы в нужные позиции.

void foffset(mas, size, offset, count[257]) {

for (i = 0; i < 257; i = i + 1) {

count[i] = 0;

}

for (i = 0; i < size; i = i + 1) {

count[1 + mas[offset + i \* sizeof(double)]]++;

}

for (i = 0; i < 256; i = i + 1) {

count[i + 1] += count[i];

}

return;

}

Процедура принимает на вход четыре аргумента. mas - это массив char’ов(можно воспринимать, как массив байт исходного массива), size - его длина, offset - номер байтов, которые необходимо посчитать и статический массив count, в него будет записан результат выполнения процедуры. В первом цикле for по i зануляем массив count. Во втором цикле for по i подсчитываем количество байт с определенным значением и записываем всё в массив count . И в последнем цикле for по i вычисляем префикс-сумму по массиву count.

# **Подтверждение корректности**

Первый метод тестирования, который использовался для подтверждения корректности - это проверка сортировок на случайных данных различного объёма. Все алгоритмы были проверены этим методом и все показали правильный результат. Также для проверки правильности выполнения алгоритма применялось сравнение с функцией из стандартной библиотеки языка C - qsort() [3]. Результаты выполнения написанной сортировки сравнивались с результатом - стандартной функции.

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

# **Результаты экспериментов**

Для исследования алгоритмов были проведены тесты на различных объемах данных начиная от и до . Первый вывод, который можно сделать после тестирования, все алгоритмы реализованы верно. Теперь имеет смысл исследовать их асимптотическую сложность.

Для подтверждения предполагаемой сложности построим график зависимости отношения времени к предполагаемой сложности. Если предположение окажется верно, то график будет похож на константу (с небольшими отклонения вверх и вниз).

**Сортировка вставками**

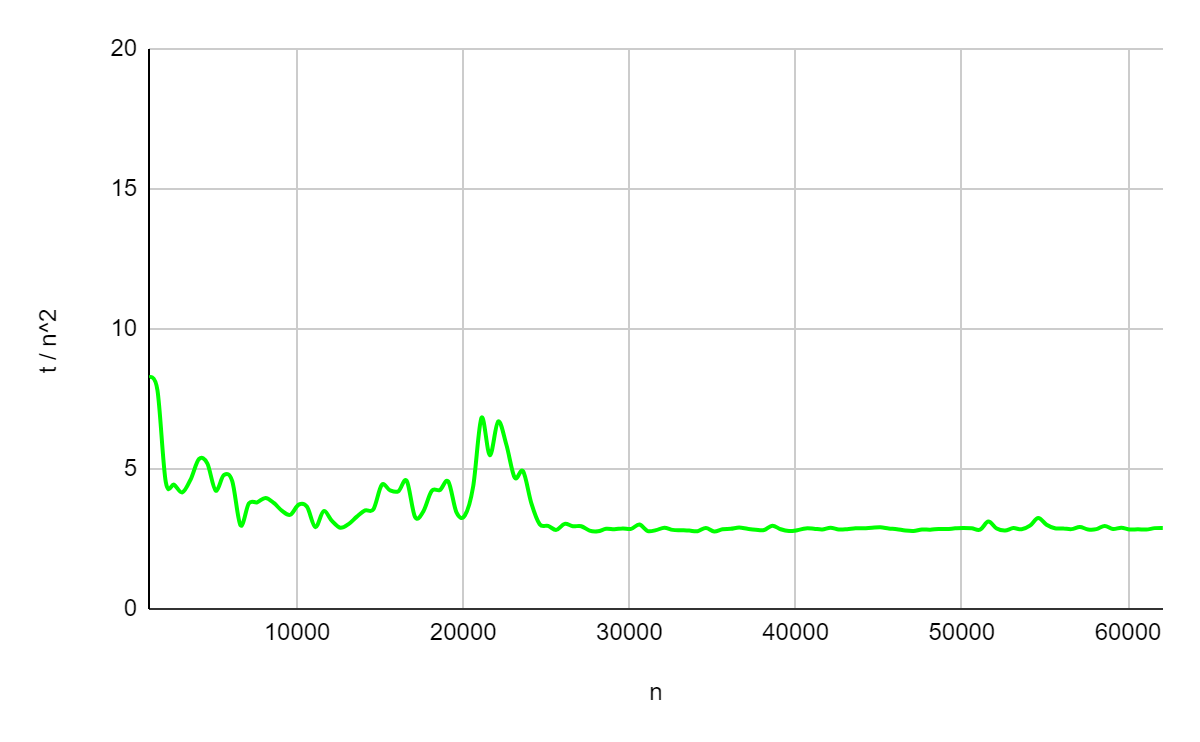


Рис. 2. График отношения времени выполнения к предполагаемой сложности для сортировки пузырьком.

Из графика видно, что функция, начиная с некоторого n стала принимать константное, значит, предположение оказалось верно и сложность алгоритма .

**Сортировка расчёской**

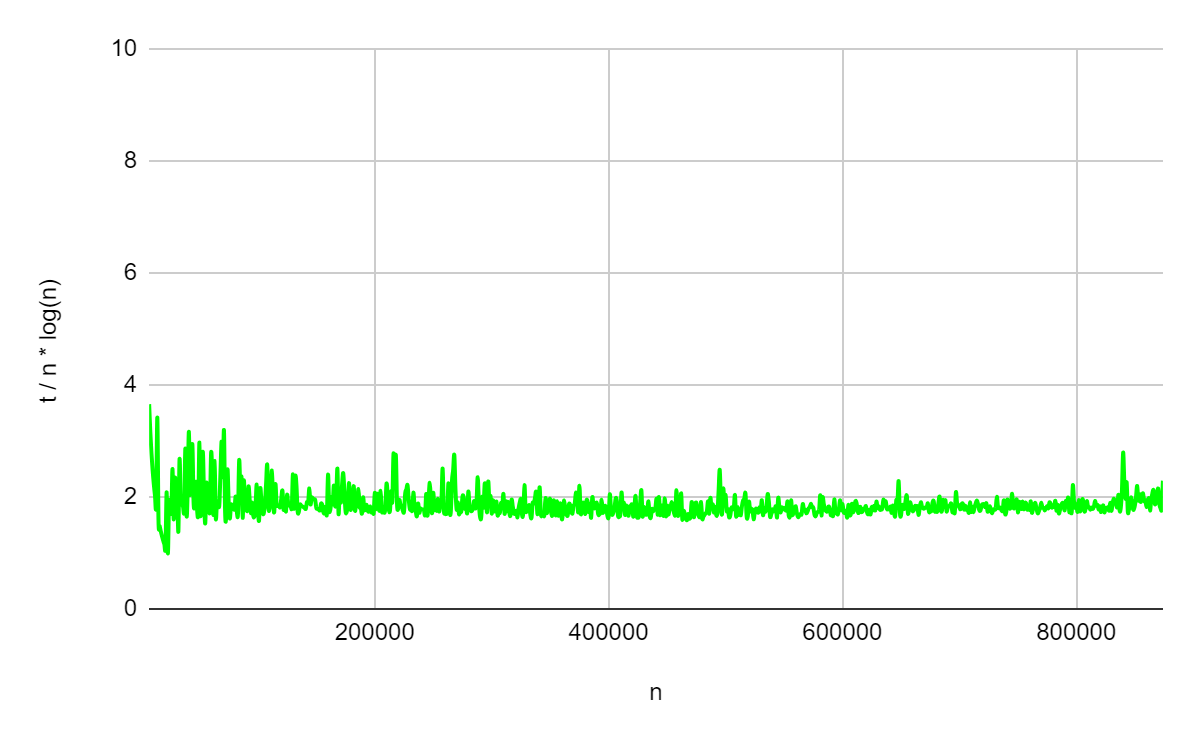
****

Рис. 3. График отношения времени выполнения к предполагаемой сложности для сортировки расчёской.

Из графика видно, что функция, начиная с некоторого n стала принимать константное, значит, предположение оказалось верно и сложность алгоритма .

**Сортировка слиянием**

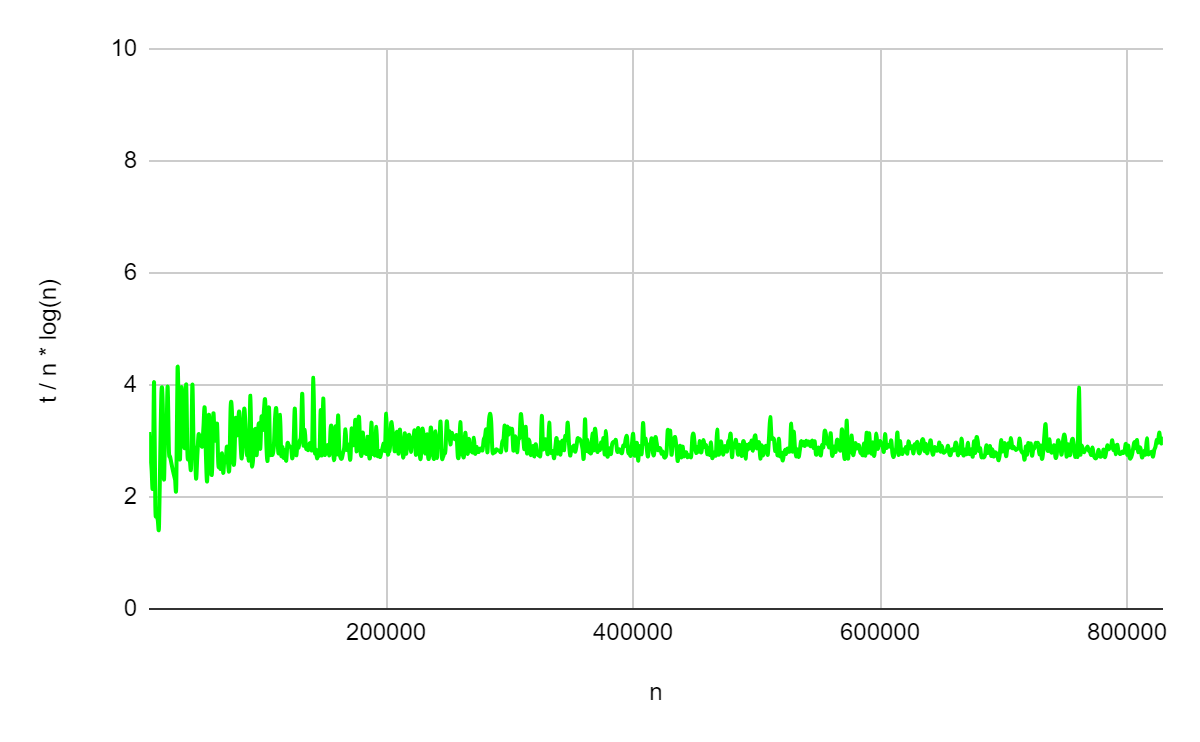
****

Рис. 4. График отношения времени выполнения к предполагаемой сложности для сортировки слиянием.

Из графика видно, что функция, начиная с некоторого n стала принимать константное, значит, предположение оказалось верно и сложность алгоритма .

**Поразрядная сортировка**

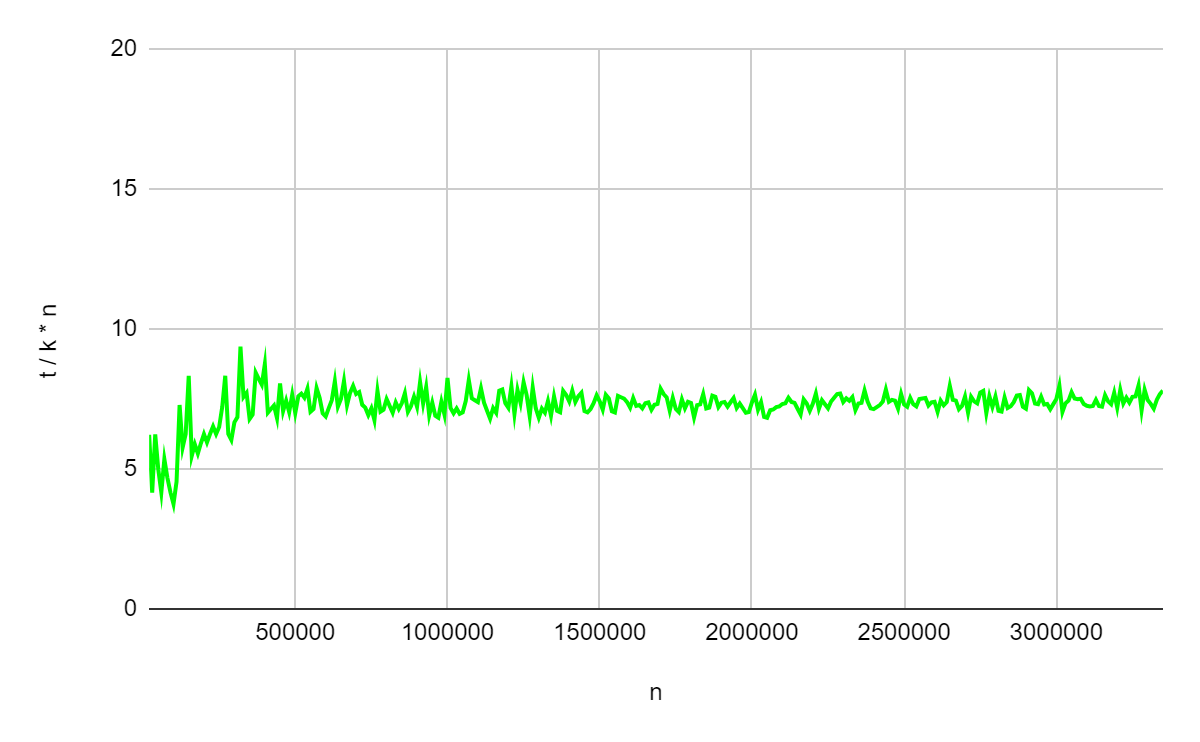
****

Рис. 5. График отношения времени выполнения к предполагаемой сложности для поразрядной сортировки.

Из графика видно, что функция, начиная с некоторого n стала принимать константное, значит, предположение оказалось верно и сложность алгоритма .

# **Заключение**

Целью данной лабораторной работы было реализовать четыре алгоритма сортировки проверить их корректность и исследовать их асимптотическую сложность. Считаю все задачи выполнены и цель достигнута.

Также в ходе выполнения работы были сделаны следующие выводы:

1. Асимптотическая сложность реализованных сортировок совпадает с предполагаемой.
2. Сортировка вставками хорошо подходит для сортировки небольших объёмов данных, в этом случае время её выполнения практически не отличается от времени выполнения более сложных сортировок.
3. Самой быстрой сортировкой оказалась поразрядная. Она хорошо подходит для сортировки больших объёмов данных, где отдельные элементы занимают относительно небольшое число байт.
4. Также отмечен тот факт, что сортировка слиянием и поразрядная требуют использование дополнительной памяти. И в тех случаях, когда нет возможности выделить дополнительную память стоит использовать сортировку расчёской. Работает она достаточно быстро и в дополнительной памяти не нуждается.

# **Список литературы**

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. Д. Э. Кнут, Искусство программирования, 2-е издание, ред. Ю. В. Козаченко,  [ISBN 5-8459-0082-1](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5845900821)
3. Cppstudio [Электронный ресурс] / Функция qsort. - Режим доступа: [Функция qsort (cppstudio.com)](http://cppstudio.com/post/891/?ysclid=lptonqzd6n380313933).

# **Приложение**

void insertionSort(double\* mas, int size) {

int i, j;

for (i = 1; i < size; i++) {

for (j = i; j > 0 && mas[j] < mas[j - 1]; j--) {

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

}

}

return;

}

void combSort(double\* mas, int size) {

int step, j;

double factor;

step = size - 1;

factor = 1.2473309;

while (step >= 1) {

for (j = 0; j < size - step; j++) {

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

std::swap(mas[j], mas[j + step]);

}

}

step = step / factor;

}

return;

}

void merge(double\* masa, double\* masb, double\* res, int sizea, int sizeb) {

int i, j, r;

i = 0;

j = 0;

r = 0;

while (sizea > i && sizeb > j)

{

if (masa[i] < masb[j]) {

res[r++] = masa[i++];

}

else {

res[r++] = masb[j++];

}

}

for (; i < sizea; i++) {

res[r++] = masa[i];

}

for (; j < sizeb; j++) {

res[r++] = masb[j];

}

return;

}

void mergeSort(double\* mas, int size) {

double\* tmpdata;

int step, i, count;

tmpdata = (double\*)malloc(size \* sizeof(double));

step = 1;

count = 0;

for (; step < size; step \*= 2) {

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

merge(mas + i, mas + i + step, tmpdata + i, std::max(0, std::min(step, size - i)), std::max(0, std::min(step, size - step - i)));

}

std::swap(mas, tmpdata);

count++;

}

if (count % 2 == 1)

{

std::swap(mas, tmpdata);

for (i = 0; i < size; i++) mas[i] = tmpdata[i];

}

free(tmpdata);

return;

}

void foffset(unsigned char\* mas, int size, int offset, int count[257]) {

int i;

for (i = 0; i < 257; i++) {

count[i] = 0;

}

for (i = 0; i < size; i++) {

count[1 + mas[offset + i \* sizeof(double)]]++;

}

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

count[i + 1] += count[i];

}

return;

}

void lsdSort(double\* mas, int size) {

int count[257];

double\* tmpdata;

unsigned char\* pm;

int i, j;

pm = (unsigned char\*)mas;

tmpdata = (double\*)malloc(size \* sizeof(double));

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

foffset(pm, size, i, count);

for (j = 0; j < size; j++) {

tmpdata[count[pm[j \* sizeof(double) + i]]++] = mas[j];

}

for (j = 0; j < size; j++) {

mas[j] = tmpdata[j];

}

}

j = size - 1;

i = 0;

while (tmpdata[j] < 0) mas[i++] = tmpdata[j--];

j = 0;

while (tmpdata[j] > 0) mas[i++] = tmpdata[j++];

free(tmpdata);

return;

}