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»**

**Выполнила**:

студентка группы 3821Б1ПМ2

Анисимова Ю. Д.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 7](#_Toc26962564)

[Описание программной реализации 8](#_Toc26962565)

[Подтверждение корректности 10](#_Toc26962566)

[Результаты экспериментов 11](#_Toc26962567)

[Заключение 15](#_Toc26962568)

[Приложение 16](#_Toc26962569)

# Постановка задачи

Задачей моей лабораторной работы реализация четырех сортировок массива типа данных double: сортировку пузырьком, быструю сортировку, сортировку слиянием и поразрядную сортировку. Нужно описать алгоритмы работы данных сортировок, проверить их корректность, посчитать количество сравнений и обменов. Провести эксперименты для подтверждения сложности и описать способ их проведения, в заключение сделать выводы по полученным данным.

# Метод решения

**Сортировка пузырьком**

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. Элементы последовательно сравниваются попарно каждый проход и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются до тех пор, пока на каком-либо из проходов не окажется, что обмены не происходят, это означает, что массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива. Количество сравнений в каждом проходе равно m-i, где i – это номер прохода по массиву (первый, второй, третий и т.д.). Сложность данного алгоритма сортировки равна O(n2).

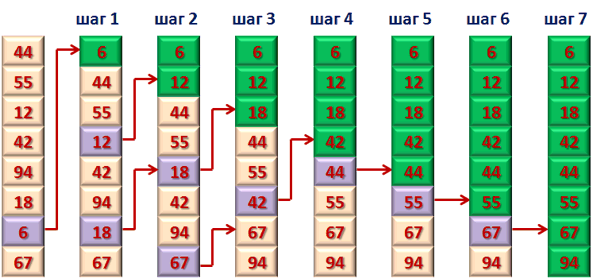


Иллюстрация работы сортировки пузырьком

На рисунке видно, как последовательно “всплывают пузырьки” наверх массива (в данном случае меньшие всплывают наверх).

**Быстрая сортировка**

Из исходного массива выбирается некоторый элемент, который принимается в качестве разделителя или опорного элемента(ключа). Все ключи, меньшие разделителя, располагаются до него, а все большие после. Перестановка элементов выполняется путём обмена местами ключей, которые необходимо переместить в другую часть массива. При этом обмениваются ключи, расположенные на большом расстоянии друг от друга и этим достигается высокий эффект упорядочивания.

Массив a[l…r] типа T(в нашем случае типа double) разбивается на два подмассива a[l…q] и a[q+1…r], таких, что каждый элемент a[l…q] меньше или равен a[q], который в свою очередь, не превышает любой элемент подмассива a[q+1…r]. Индекс вычисляется в ходе процедуры разбиения. Подмассивы a[l…q] и a[q+1…r] сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.

Сложность данного алгоритма сортировки равна O(N\*log(N)).

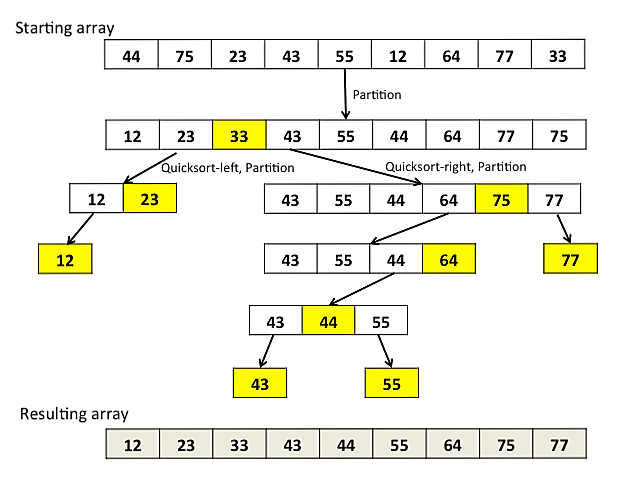


Иллюстрация работы быстрой сортировки

**Сортировка слиянием**

Сортируемый массив разделяется на два подмассива приблизительно одинакового размера. Рекурсивное разбиение продолжается до тех пор, пока размер массива не достигнет единицы. Каждая из частей сортируется отдельно. Два упорядоченных массива объединяются в третий результирующий массив.

Сложность данного алгоритма сортировки равна O(N\*log(N)).

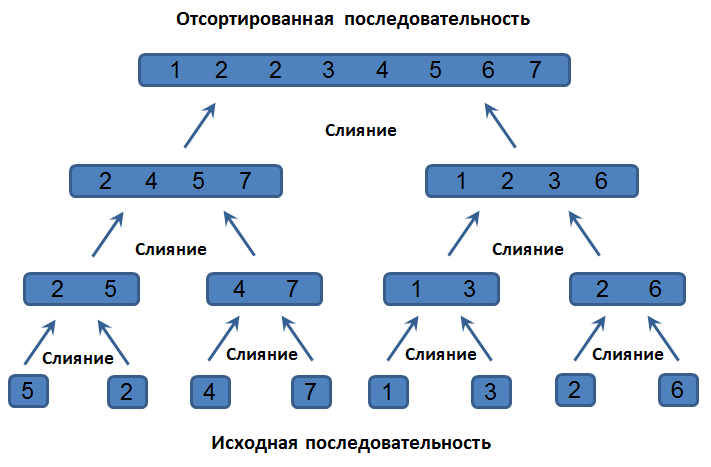


Иллюстрация работы сортировки слиянием

**Поразрядная сортировка**

Сравнение элементов 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.

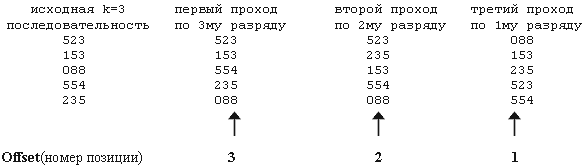
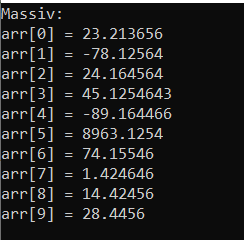


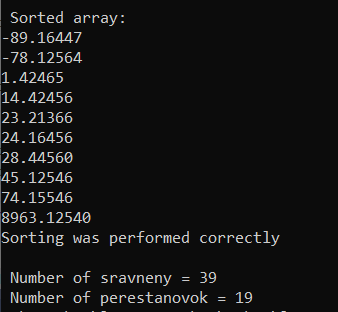
Иллюстрация работы поразрядной сортировки

# Руководство пользователя

При запуске программы появляется окно ввода, где предлагается ввести вручную значения сортируемого массива. Далее выполняется сортировка.



В результате работы программы будет выводится отсортированный массив, результат проверки отсортированного массива на корректность, количество перестановок и сравнений.



# Описание программной реализации

В моем проекте есть 4 файла: Buble sort my, MergeSort, QuickSort, RadixSort. В каждом из них содержится сортировка, соответствующая названию файла.

***Общая функция:***

void checking(double\* arr, double\* check, int n) Функция, которая выполняет проверку массива на корректность. В ней производится вызов библиотечной функции qsort, с помощью неё сортируется копия исходного массива. Затем копия сравнивается с массивом после сортировки.

***Сортировка пузырьком:***

void bubble\_sort(double\* arr, const size\_t size) Функция принимает на вход указатель на массив (double) и его размер. Сортирует массив по возрастанию, сравнивая и переставляя элементы, начиная с конца.

***Быстрая сортировка:***

void quickSort(double\* arr, int first, int last) Функция принимает на вход указатель на массив (double) и номер крайнего правого и крайнего левого элементов. Делит массив на две части, серединный элемент выбирает опорным, относительно него отдельно сортирует части массива.

***Сортировка слиянием:***

void merge(double\* arr, double\* second, int p, int q, int t) Функция принимает на вход изначальный и дополнительный массивы (double), номер крайнего правого и крайнего левого элементов, номер элемента из середины массива. Упорядочивает части массива, после чего собирает их в один.

void mergeSort(double\* arr, double\* second, int p, int t) Функция принимает на вход изначальный и дополнительный массивы (double), номер крайнего правого и крайнего левого элементов. Обеспечивает рекурсивное разбиение задачи, за счет чего части массива сортируются отдельно.

***Поразрядная сортировка:***

void createCounters(double\* in, long\* count, long N) Функция принимает на вход указатель на исходный массив (double), указатель на дополнительный массив (long), размер исходного массива. Создает ячейки для возможных значений разрядов чисел.

void radixPass(short Offset, long N, double\* in, double\* out, long\* count) Функция принимает на вход номер байта Offset, число элементов, исходный массив, массив, куда будут записываться числа, отсортированные по байту Offset, массив счетчиков count. На первом проходе подсчитывается количество различных значений в этом разряде. Для каждого возможного значения подготавливаются массивы. При втором проходе выписываются сами элементы в эти массивы. Второй и последующие проходы выполняются отдельно над каждой корзиной, полученной на предыдущем проходе, с делением её на «подкорзины» и сравнением соответственно второго и последующих символов строк.

void signedradixLastPass(short Offset, long N, double\* in, double\* out, long\* count) Функция принимает на вход номер байта Offset, число элементов, исходный массив, массив, куда будут записываться числа, отсортированные по байту Offset, массив счетчиков count. Работает со старшими байтами, обеспечивая правильное положение знаковых чисел в отсортированном массиве.

void radixSort(double\* in, double\* out, int N) Функция принимает на вход указатель на исходный массив (double), указатель на дополнительный массив, размер массива. Осуществляет sizeof(double) проходы по направлению от младшего байта к старшему.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция Cheking. В ней производится вызов библиотечной функции qsort, с помощью неё сортируется копия исходного массива. Затем копия сравнивается с массивом после сортировки.

Если сортировка пройдет корректно, будет произведен вывод фразы «Сортировка работает корректно», в худшем случае «Сортировка работает некорректно»

# Результаты экспериментов

**Сортировка пузырьком**

Сортировка пузырьком имеет сложность n2

На Рисунке 1 изображен график изменения количества сравнений (синяя ветка) и количества перестановок (красная ветка) к n (количеству элементов в массиве). Видно, что данные графики имеют асимптотику n2, что является экспериментальным подтверждением указанной сложности.

На Рисунке 2 изображен график изменения отношения количества сравнений к сложности (зеленая ветка) и график изменения отношения количества перестановок к сложности (желтая ветка). Видно, что значения сходятся к константам. Поскольку графики сходится к некоторому значению, это является экспериментальным подтверждением указанной сложности.

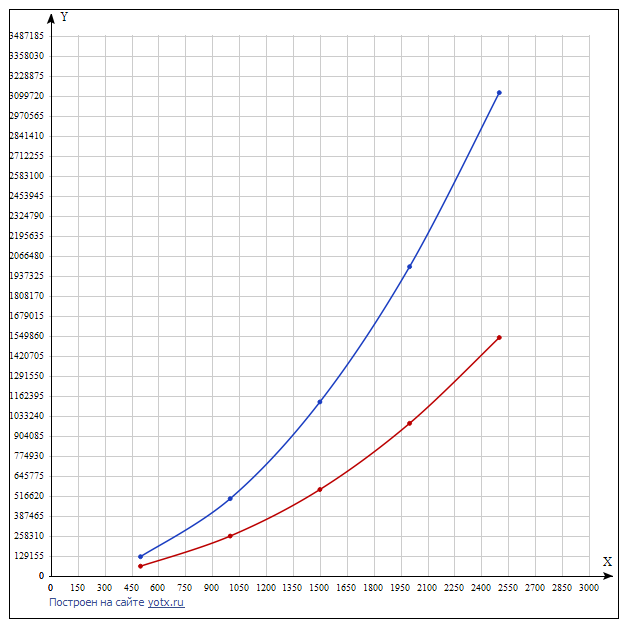
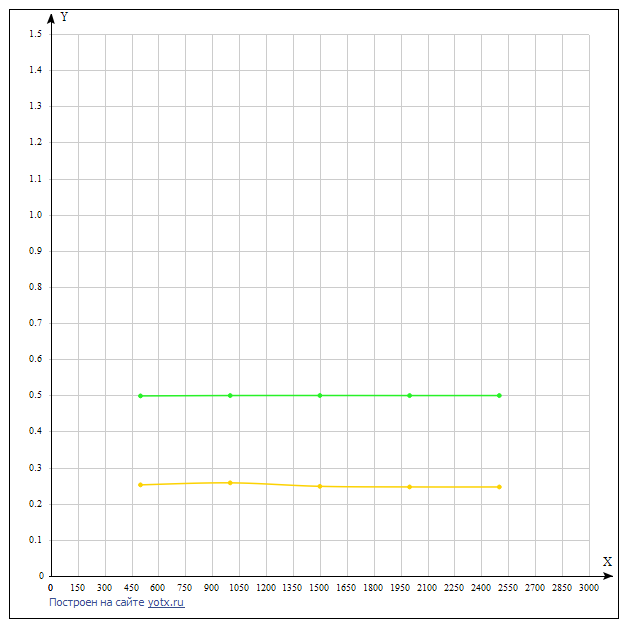


Рисунок 2

Рисунок 1

**Быстрая сортировка**

Быстрая сортировка имеет сложность n\*log(n).

На Рисунке 3 изображен график изменения количества сравнений (синяя ветка) и количества перестановок (красная ветка) к n (количеству элементов в массиве). Видно, что данные графики имеют асимптотику n\*log(n), что является экспериментальным подтверждением указанной сложности.

На Рисунке 4 изображен график изменения отношения количества сравнений к сложности (зеленая ветка) и график изменения отношения количества перестановок к сложности (желтая ветка). Видно, что значения сходятся к константам. Поскольку графики сходится к некоторому значению, это является экспериментальным подтверждением указанной сложности.

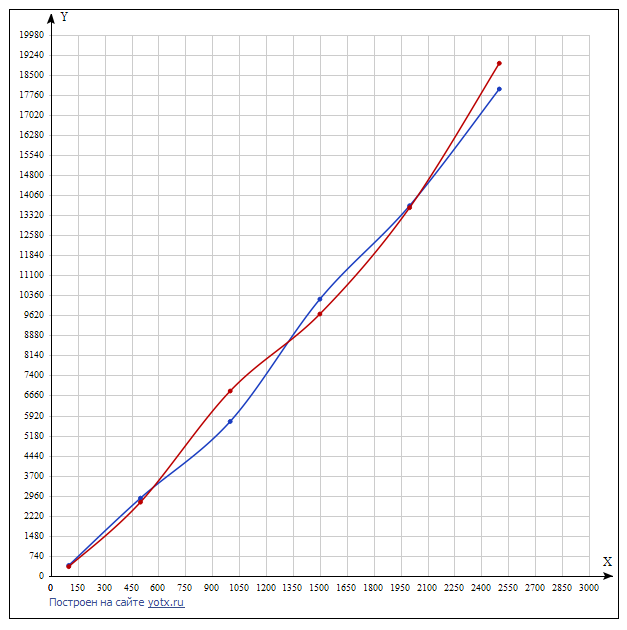
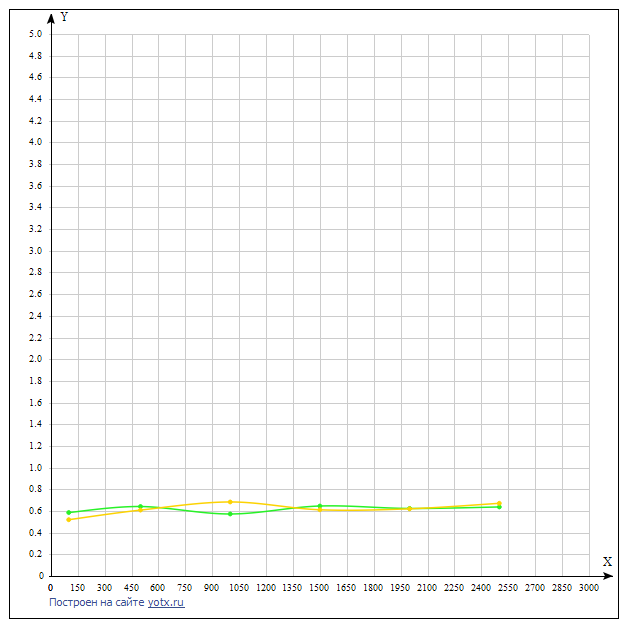


Рисунок 4

Рисунок 3

**Сортировка слиянием**

Сортировка слиянием имеет сложность n\*log(n).

На Рисунке 5 изображен график изменения количества сравнений (синяя ветка) и количества перестановок (красная ветка) к n (количеству элементов в массиве). Видно, что данные графики имеют асимптотику n\*log(n), что является экспериментальным подтверждением указанной сложности.

На Рисунке 6 изображен график изменения отношения количества сравнений к сложности (зеленая ветка) и график изменения отношения количества перестановок к сложности (желтая ветка). Видно, что значения сходятся к константам. Поскольку графики сходится к некоторому значению, это является экспериментальным подтверждением указанной сложности.

# 

Рисунок 6

Рисунок 5

**Поразрядная сортировка**

Сложность поразрядной сортировки O(K\*(N+M)), где K – количество цифр в числе, или число разрядов, M – количество цифр в разряде.

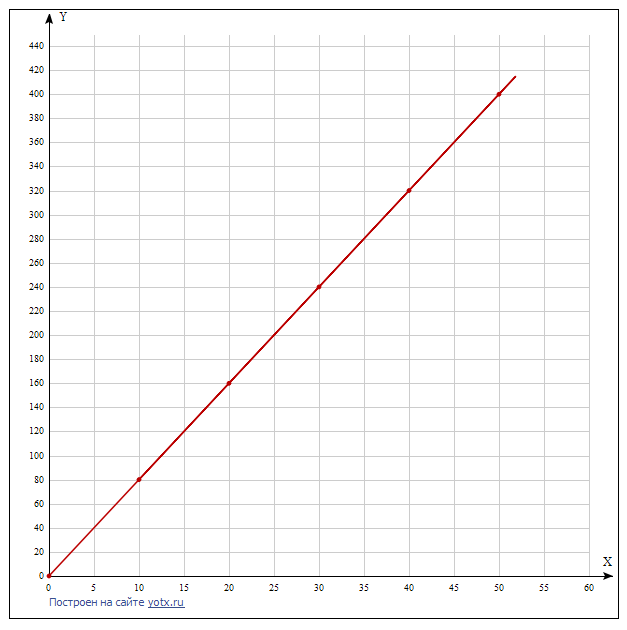


Рисунок 7

На Рисунке 7 изображен график изменения количества перестановок к n (количеству элементов в массиве).

Глядя на график, можно заметить, что количество операций пропорционально числу элементов в массиве, при n = 10 количество операций = 80 и так далее, т. е. сложность может быть получена как O(N). Это является экспериментальным подтверждением указанной сложности.

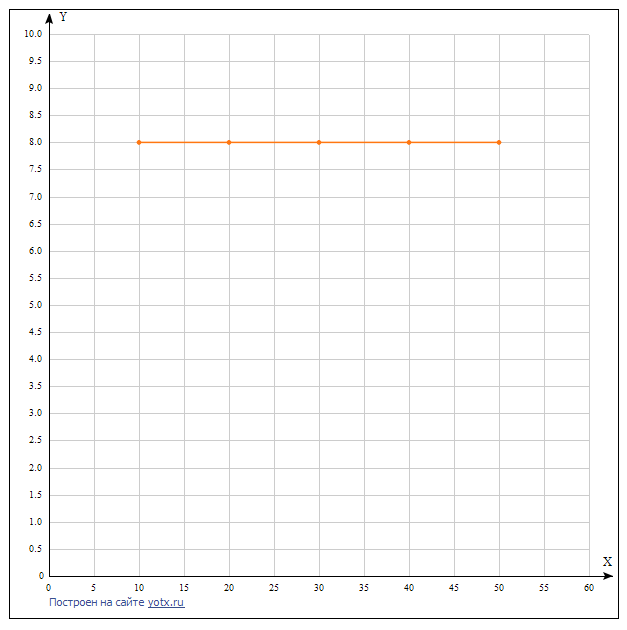


Рисунок 8

На Рисунке 8 изображен график изменения отношения количества перестановок к сложности алгоритма.

Получилось так, что всем значениям количества элементов в массиве сопоставлено число 8 (т.е так же все сходится к константе), что является экспериментальным подтверждением указанной сложности.

# Заключение

В ходе лабораторной работы я реализовала четыре сортировки массива типа данных double: сортировку пузырьком, быструю сортировку, сортировку слиянием и поразрядную сортировку. Описала алгоритмы работы данных сортировок, проверила их корректность, посчитала количество сравнений и обменов. Провела эксперименты для подтверждения сложности и описала способ их проведения, сделала выводы по полученным данным.

# Приложение

**Сортировка пузырьком**

void bubble\_sort(double\* arr, const size\_t size)

{

for (int i = 0; i < SIZE; i++)

{

flag = 0;

for (int p = 0; p < SIZE - i - 1; p++)

{

sravn++;

if (arr[p] > arr[p + 1])

{

perest++;

double num = arr[p];

arr[p] = arr[p + 1];

arr[p + 1] = num;

flag = 1;

}

}

if (flag == 0)

break;

}

}

**Быстрая сортировка**

void quicksort(double\* a, int L, int R)

{

int i = L, j = R;

double q = a[(L + R) / 2];

do {

while (a[i] < q) { i++; sravn++; }

while (a[j] > q) { j--; perest++; }

if (i <= j)

{

double w = a[i];

a[i] = a[j];

a[j] = w;

i++; j--; sravn++; perest++;

}

} while (i <= j);

if (L < j)

quicksort(a, L, j);

if (i < R)

quicksort(a, i, R);

}

**Сортировка слиянием**

void merge(double\* arr, double\* second, int p, int q, int t)

{

int f, i, s, k;

q = (p + t) / 2; f = p; i = p; s = q + 1;

while ((f <= q) && (s <= t))

{

sravn++;

if (arr[f] <= arr[s])

{

perest++;

second[i] = arr[f];

f++;

}

else

{

perest++;

second[i] = arr[s];

s++;

}

i++;

} if (f > q)

{

for (k = s; k <= t; k++)

{

perest++;

second[i] = arr[k];

i++;

}

}

else

{

for (k = f; k <= q; k++)

{

perest++;

second[i] = arr[k];

i++;

}

} for (k = p; k <= t; k++)

{

arr[k] = second[k];

}

}

void mergeSort(double\* arr, double\* second, int p, int t) {

if (p < t) {

int q = (p + t) / 2;

mergeSort(arr, second, p, q);

mergeSort(arr, second, q + 1, t);

merge(arr, second, p, q, t);

}

}

**Поразрядная сортировка**

void createCounters(double\* in, long\* count, long N) {

unsigned char\* bp = (unsigned char\*)in;

unsigned char\* end = (unsigned char\*)(in + N);

memset(count, 0, 256 \* sizeof(double) \* sizeof(long));

while (bp != end) {

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

count[256 \* i + \*(bp++)]++;

}

}

}

void radixPass(short Offset, long N, double\* in, double\* out, long\* count) {

double\* sp;

long s, c, \* cp;

unsigned char\* bp;

s = 0;

cp = count;

for (int i = 256; i > 0; --i, ++cp) {

c = \*cp;

\*cp = s;

s += c;

}

bp = (unsigned char\*)in + Offset;

sp = in;

for (int i = N; i > 0; --i, bp += sizeof(double), ++sp) {

cp = count + \*bp;

out[\*cp] = \*sp;

(\*cp)++;

perest++;

}

}

void signedradixLastPass(short Offset, long N, double\* in, double\* out, long\* count) {

double\* sp;

long s, c, \* cp, numNeg = 0;

unsigned char\* bp;

for (int i = 128; i < 256; i++) numNeg += count[i];

s = numNeg;

cp = count;

for (int i = 0; i < 128; ++i, ++cp) {

c = \*cp;

\*cp = s;

s += c;

}

cp = count + 255;

s = 0; \*cp = 0;

for (int i = 255; i >= 128; i--, cp--) {

s += \*cp;

\*cp = s;

}

bp = (unsigned char\*)in + Offset;

sp = in;

for (int i = N; i > 0; i--, bp += sizeof(double), sp++) {

cp = count + \*bp;

if (\*bp < 128) {

out[\*cp] = \*sp;

(\*cp)++;

}

else {

(\*cp)--;

out[\*cp] = \*sp;

}

perest++;

}

}

void radixSort(double\* in, double\* out, int N) {

int i;

long\* counters = (long\*)malloc(sizeof(double) \* 256 \* sizeof(long));

long\* count;

createCounters((double\*)in, counters, N);

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

count = counters + 256 \* i;

radixPass(i, N, in, out, count);

double\* tmp = in;

in = out;

out = tmp;

}

count = counters + 256 \* (i);

signedradixLastPass(i, N, in, out, count);

}