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 3822Б1ПМ-1

Ковалев К.И.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Результаты экспериментов 8](#_Toc26962567)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Поставлена задача: реализовать сортировку по возрастанию массива случайных чисел типа **double** (c плавающей точкой, двойной точности: 64 бита) размера, вводимого пользователем, тремя различными алгоритмами: «пузырьком», «расчёской» и поразрядной сортировкой на языке C. Программа должна осуществлять проверку сортировки, иметь понятный пользователю интерфейс, быть простой и удобной в использовании и эффективной по времени и ресурсам компьютера. Необходимо описать функционирование программы, убедиться в соответствии скорости её работы с заявленной (для каждого конкретного алгоритма) экспериментально, описать условия проведения экспериментов и его результаты. Для этого программа должна фиксировать и выводить время работы каждого конкретного алгоритма.

# Метод решения

Данные будем сортировать тремя алгоритмами: «пузырьком», «расчёской» и поразрядно.

Рассмотрим отдельно каждый из методов.

**Сортировка пузырьком**

Принцип работы алгоритма заключается в перемене мест рядом стоящих элементов в случае, если они не отсортированы. Сортировка выполняется в несколько проходов, в каждом из которых не менее одного элемента встают на своё место: в случае сортировки по возрастанию на место (после еще не отсортированных элементов) встаёт наибольший элемент из ещё не отсортированных. Массив, начиная с этого элемента и до конца (в сторону увеличения индексов), отсортирован. Т.е. на каждом следующем проходе по массиву можно рассматривать на 1 элемент меньше, чем на предыдущем.

Таким образом, нетрудно получить оценку вычислительной сложности: лучший случай (массив верно отсортирован, 1 проход, 0 перестановок) – **O(N),** худший случай (массив отсортирован в обратном порядке) – **O(N\*(N+1)/2)** ≈ **O(N2),** здесь и далее **N** – количество элементов массива. В случае, если массив оказался отсортирован на какой-либо итерации, т.е. на ней не будет выполнено ни одной перестановки элементов, программа завершит работу благодаря флагу остановки.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 65 | 23 | 17 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 23 | 65 | 17 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 23 | 17 | 65 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 23 | 17 | 8 | 65 |

*Рис. 1: иллюстрация прохода по массиву при сортировке пузырьком. Зелёным обозначен элемент уже отсортированной части массива*

**Сортировка расчёской**

Сортировка расчёской является оптимизированной вариацией пузырьковой сортировки. Принцип её работы схож с принципом работы пузырьковой сортировки, но переставляются элементы, стоящие не рядом, а удалённые на определённый шаг. Сортировка запускается с шагом, равным длине массива, уменьшенной на 1 (т.е. сравниваются первый и последний элементы), и в начале каждой (включая первую) итерации шаг уменьшается в 1,247 раз (для числа 1,247 доказано, что сортировка работает). Шаг округляется до ближайшего меньшего целого. Сравниваются элементы, разность индексов которых равна шагу. В случае, если они стоят в неправильном порядке, элементы меняются местами. За конечное число итераций шаг становится равным единице, т.е. будут рассматриваться соседние элементы, и эта операция будет производиться до тех пор, пока количество перестановок не станет равно 0, т.е. массив не будет отсортирован. После программа завершается.

В худшем случае сортировка имеет вычислительную сложность **O(N2),** что соответствует вычислительной сложности пузырьковой сортировки, в лучшем и среднем случаях - **O(N\*logN).**

*Завершилась вторая итерация, произведена 1 перестановка эл-тов, идущих с шагом 3.*

*Завершилась первая итерация (сравнение первого и последнего эл-тов), произведена 1 перестановка.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 65 | 23 | 17 | 4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 65 | 23 | 17 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 8 | 23 | 17 | 65 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | 8 | 17 | 23 | 65 |

*После 4-й итерации (шаг равен 1) массив отсортирован.*

*Рис. 2: сортировка расчёской.*

*На третьей итерации (шаг равен двум) не выполнено ни одной перестановки.*

**Поразрядная сортировка**

Принцип работы алгоритма заключается в сравнении разрядов числа и расположении чисел согласно возрастанию разрядов. Существует два алгоритма поразрядной сортировки: сортировка от старшего разряда к младшему (**MSD** – *«most significant digit»*) и от младшего к старшему **(LSD** – *«less significant digit»*. Выбрана сортировка **LSD**. Отметим, что поразрядная сортировка требует дополнительной памяти, равной по размеру объёму входных данных.

Фактически рассматриваются блоки по 1 биту (из восьми двоичных комбинаций), принимающие соответственно 28 = 256 значений. Реализация сортировки по разрядам десятичных чисел представляется более сложной задачей, т.к. формирование конкретного разряда десятичного числа происходит посредством суммирования значений сразу нескольких двоичных разрядов. Поэтому ввиду двоичного представления, которое реализовано в памяти компьютера, проще и эффективнее сортировать числа по блокам из одного или нескольких двоичных разрядов, чем десятичных.

Как уже было сказано, для каждого блока определено 256 значений, поэтому массив входных данных типа **double** интерпретируется программой как массив элементов типа **unsigned char**. Этот тип имеет размер 1 байт, т.е. принимает 256 значений 0…255. Фактически, элементы **double**, имеющие размер 8 байт, рассматриваются как эл-ты **unsigned char** (**double** разбивается побайтово). Каждому эл-ту **double** сопоставляется 8 эл-тов **unsigned char**, записанных в памяти последовательно.

**LSD** сортировка - сортировка от младшего разряда (блока в 1 байт) к старшему. Для **k**-х (где **k = 0…7**) блоков входных данных реализуется сортировка подсчётом (по возрастанию) и формируется массив индексов, по которым должны располагаться элементы, содержащие соответствующие блоки. Затем по этим индексам во вспомогательный массив помещаются эти элементы. Значения исходного массива переписываются значениями вспомогательного и сортируется следующий разряд (блок).

Отметим, что описанный алгоритм сортирует верно беззнаковые целые числа. Обратимся к представлению чисел с плавающей запятой: **double** представляется как **(-1)s\*2^(p-bias)\*(1.m)**, имеет размер 8 байт и в памяти представляется следующим образом (числа сверху номеруют биты):

**0**

**51**

**62**

**63**

**52**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S** | **P** | **P** | **P** | **P** | **P** | **P** | **M** | **M** | **M** | **M** | **M** | **M** | **M** | **M** | **M** |

*Рис. 3: представление значений типа double в памяти.*

где **S** - знак числа ( (-1)**S**);

**P** - порядок + смещение (для **double** смещение **bias** равно 511) (2^(**P-bias**));

**M** – мантисса.

Таким образом, представление отрицательного числа начинается с единицы, а представление положительного - с нуля, поэтому при поразрядной сортировке отрицательные значения окажутся после всех положительных. Однако все числа одного знака между собой будут отсортированы верно. Следовательно, для верной сортировки чисел с плавающей запятой достаточно отсортировать входные данные по уже рассмотренному алгоритму, а затем все отрицательные значения, не нарушая порядка их следования, записать перед положительными.

Сортировка обладает линейной вычислительной сложностью **O(N)**, т.к. сортировка подсчётом, переписывание массива из вспомогательного в исходный и другие операции также обладают сложностью не выше линейной, и выполняются количество раз, равное константе.

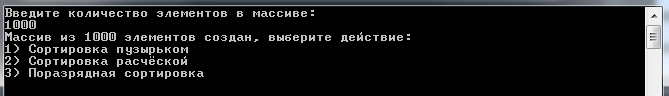
# Руководство пользователя

Интерфейс программы интуитивно понятен пользователю, и всё же следует рассмотреть его отдельно.



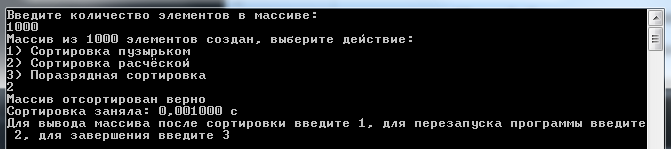
*Рис. 4: ввод количества элементов массива.*

1. При запуске программы пользователь должен ввести с клавиатуры количество элементов в сортируемом массиве.



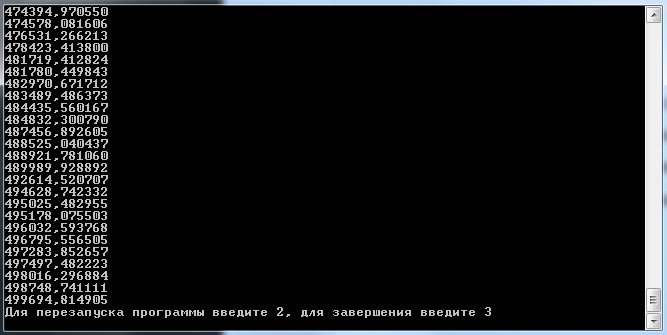
*Рис. 5: выбор алгоритма сортировки.*

1. После этого будет сформирован массив случайных чисел длины, введённой пользователем на шаге ранее. Уведомление об этом будет выведено на экран. Далее пользователю предлагается выбрать алгоритм, по которому будет отсортирован созданный массив. Для этого необходимо ввести номер, под которым в списке располагается выбранный алгоритм.



*Рис. 6: выбор дальнейшего действия.*

1. Массив будет отсортирован, сортировка будет проверена на корректность, уведомления об этом и времени, затраченном на сортировку, будут выведены на экран. Далее отсортированный массив может быть выведен на экран, программа может быть либо перезапущена (начнётся выполнение программы с пункта 1 руководства), либо завершена при вводе соответственно **1**, **2** или **3** пользователем.



*Рис. 7: меню после вывода массива.*

1. После вывода отсортированного массива на экран программа также может быть либо перезапущена (начнётся выполнение программы с пункта **1** руководства), либо завершена при вводе соответственно **2** или **3** пользователем.

# Описание программной реализации

Программа состоит из одного файла – **LR.cpp**, в котором реализованы функции сортировок, смены значений двух переменных, проверки корректности сортировок, вспомогательные и главная – **int main().**

**void swp(double\* a, double\* b) –** функция смены значений двух переменных. Принимает адреса двух переменных типа **double** и меняет местами значения по этим адресам с использованием дополнительной памяти в размере 8 байт (размер данных типа **double**).

**int comp(const void\* a, const void\* b) –** компаратор для функции сортировки **qsort**, используемой в функции проверки корректности сортировки. Принимает два элемента (**a** и **b**) массива, который сортируется **qsort,** сравнивает их и возвращает **-1** в случае, если **a < b**, **1** в случае, если **a > b**, иначе, если **a=b**, – **0**.

**bool correct(double\* in, double\* sort, int n) –** логическая функция, проверяющая корректность сортировки. Принимает на вход исходный массив **in**, отсортированный одним из алгоритмов массив **sort**, размер этих массивов **n**. Запускает для исходного массива сортировку **qsort** из библиотеки **stdlib: qsort (in, n, sizeof(double), comp)**, затем посимвольно сравнивает элементы **in** и **sort**. В случае, если для любого **i** = **0**…**n-1** **in[i]** равен **sort[i]**, возвращает **true**, иначе возвращает **false**.

**void bubble(double\* in, int n)** – функция пузырьковой сортировки. Принимает на вход исходный массив **in** и размер этого массива **n**. Функция состоит из вложенного цикла, в котором сравниваются элементы массива с разностью индексов, равной 1 (соседние элементы) и меняются местами в случае, если они не удовлетворяют условию сортировки (по возрастанию). Внешний цикл имеет флаг остановки **count.**

**void comb(double\* in, int n) –** функция сортировки расчёской. Принимает на вход исходный массив **in** и размер этого массива **n**. Состоит из вложенного цикла, который отвечает за проход по массиву и сравнение элементов массива с разностью индексов, равной шагу, а также смену их значений в случае, если они не удовлетворяют условию сортировки (по возрастанию). Шаг сортировки определяется внешним циклом (уменьшается в **1,247** раза в начале каждой итерации). Внешний цикл имеет условия остановки: он завершается, если на прошлой итерации не было выполнено ни одной перестановки элементов и шаг сортировки уже уменьшился до **1**.

**void countbyte (unsigned long long\* mas, int n, int count[256], int byte) –** функциясортировки подсчётом **k**-х блоков по 1-му байту всех чисел исходного массива. Является частью вспомогательной функцией поразрядной сортировки. Принимает на вход исходный массив **mas**, интерпретированный как массивданных типа **unsigned long long**, размер этого массива **n**, массив для подсчёта элементов **count**, значение **k**, сообщающее, какой байт исходных чисел сортируется (переменная **byte**). Функция интерпретирует исходный массив **mas** как массив **masc** данных типа **unsigned char**, т.е. фактически представляет входные данные побайтово. Функция состоит из трёх циклов, один из которых обнуляет массив **count** перед началом сортировки, второй – собственно заполняет ячейки **count[i]** количествамивхождений байта, равного **i**, в множество всех **k**-х байтов исходных чисел, третий - формирует массив индексов, по которым должны располагаться элементы, содержащие соответствующие блоки.

**void radix\_unsigned (unsigned long long\* mas, int n, unsigned long long\* mas\_tmp) –** функция поразрядной сортировки беззнаковых целых чисел. Принимает на вход исходный массив **mas**, интерпретированный как массивданных типа **unsigned long long**, размер этого массива **n** и вспомогательный массив **mas\_tmp** данных типа **unsigned long long** также размера **n**. Функция состоит из внешнего цикла на 8 итераций (сортировка по блокам, **k**=0…7), на каждой из которых запускается функция **countbyte** для **k**-го блока, по сформированному массиву индексов во вспомогательный массив помещаются элементы, которым принадлежит байт. Значения исходного массива переписываются значениями вспомогательного.

**void radix\_double (double\* mas, int n, double\* tmp) –** функция поразрядной сортировки знаковых чисел с плавающей запятой. Принимает на вход исходный массив **mas**, размер этого массива **n** и вспомогательный массив **mas\_tmp** также размера **n**. Функция интерпретирует исходный массив **mas** как массив данных типа **unsigned long long** и передаёт его в функцию **radix\_unsigned**. После сортировки входных данных как беззнаковых целых, функция **radix\_double** записываетвсе отрицательные числа, не нарушая порядка их следования, перед положительными во вспомогательный массив. После этого вспомогательный массив хранит отсортированную копию исходного.

В основной функции **int main()** реализован вызов функций сортировок и написано меню. Функция состоит из бесконечного цикла **while (true),** условием остановки которого является ввод пользователем определённого значения. Внутри цикла реализовано меню – вывод запросов ввода на экран, ввод значений с клавиатуры; выделение памяти под вспомогательные и основные массивы; генерация случайных элементов массива **double(rand())/RAND\_MAX\*1000000-500000**, которые одновременно заносятся в основной массив **a**, который после сортируется, и вспомогательный **etalon**, по которому осуществляется проверка сортировки. Меню реализовано с помощью множественного условного оператора **switch-case**. Пользователь вводит номер алгоритма сортировки из списка на экране, соответствующая этому номеру ветка условия запускает функцию сортировки. Время сортировки находится как **(fin-start)/double(CLOCKS\_PER\_SEC))**, где **start** **–** переменная, содержащая значение счётчика тактов процессора до начала сортировки, а **fin** – после её завершения. Константа **CLOCKS\_PER\_SEC** содержит усреднённое значение количества тактов процессора в секунду. В конце каждой итерации выводится значение времени, которое заняла сортировка, отчёт функции проверки сортировки о её корректности, очищается память, занятая под массивы, в зависимости от введённого пользователем значения программа продолжается или завершается.

# Подтверждение корректности

Для подтверждения корректности в программе используется функция **bool correct (double\* in, double\* sort, int n)**, прототип и реализация которой описаны выше. В качестве эталонной используется функция **qsort** стандартной библиотеки **stdlib**. Посимвольное сравнение исходного массива, отсортированного **qsort** и одной из функций сортировок, позволяет сделать вывод о корректности сортировки. Так, если в массивах присутствуют различные элементы с одинаковыми индексами, функция при нахождении первого из них возвращает **false**, что свидетельствует о некорректности сортировки. В противном случае (элементы обоих массивов, имеющие одинаковые индексы, равны) функция возвращает **true**. Сортировка работает за время **O(N\*logN)**, как и сортировка **qsort.**

# Результаты экспериментов

Был проведён ряд экспериментальных запусков программы, в ходе которых одинаковый объем данных сортировался разными алгоритмами, зафиксировано среднее время работы программы:

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов в сортируемом массиве | Среднее время T, с работы программы при: | | |
| сортировке пузырьком | сортировке расчёской | поразрядной сортировке |
| 100 | 0,001 | 0,001 | 0,000000 |
| 1 000 | 0,018 | 0,001 | 0,001 |
| 5 000 | 0,26 | 0,004 | 0,002 |
| 10 000 | 1,022 | 0,009 | 0,004 |
| 50 000 | 26,675 | 0,035 | 0,011 |
| 100 000 | 101,619 | 0,069 | 0,034 |
| 500 000 | - | 0,29 | 0,142 |
| 1 000 000 | - | 0,641 | 0,248 |
| 5 000 000 | - | 3,081 | 1,131 |
| 10 000 000 | - | 6,853 | 2,365 |
| 50 000 000 | - | 34,222 | 11,99 |

*Табл. 1: результаты экспериментов*

*Примечание:*

1. *в таблице «*-*» означает, что время сортировки этого объёма данных велико, а ранее собранных данных достаточно для того, чтобы сделать вывод о вычислительной сложности сортировки;*
2. *значение «0,000000» означает, что время сортировки этого объёма данных мало для того, чтобы быть измеренным использованным методом;*

Вычислим фактическую вычислительную сложность алгоритмов сортировки. Для этого пошагово будем делить время работы конкретной сортировки на **N**, **N\*log2N**, **N2** и строить графики зависимости полученной величины от размера сортируемого массива:

Отметим, что для поразрядной сортировки график зависимости **T/N** от **N** стремится к константе, что свидетельствует о том, что вычислительная сложность поразрядной сортировки в среднем – **O(N)**. Это подтверждает указанную в главе «Метод решения» вычислительную сложность.

Зависимость **T/N\*log2N** от **N** для сортировки расчёской стремится к константе. Это подтверждает среднюю вычислительную сложность **O(N\*logN)**.

Зависимость **T/N2** от **N** для сортировки пузырьком стремится к константе. Это подтверждает среднюю вычислительную сложность **O(N2)**.

# Заключение

Итак, написана программа, реализующая сортировку массива случайных чисел типа **double** по возрастанию тремя различными алгоритмами. Интерфейс, эффективность и скорость работы программы удовлетворяют обозначенным условиям. Описаны принципы функционирования программы, проведена серия экспериментов, подтверждающих соответствие программы теоретическим данным.

# Приложение

void swp(double\* a, double\* b) //смена значений двух переменных

{

double \* tmp = a;

a = b;

b = tmp;

}

bool correct(double\* in, double\* sort, int n) //Функция проверки сортировки

{

int i;

qsort(in, n, sizeof(double), comp);

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

{

if (in[i] != sort[i]) return false;

}

return true;

}

void bubble(double\* in, int n) //Сортировка пузырьком

{

int i, j, count = 1;

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

{

count = 0;

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

{

if (in[j] > in[j + 1])

{

swp(&in[j], &in[j + 1]);

count++;

}

}

}

}

void comb(double\* in, int n) //Модернизированная расчёска

{

double factor = 1.246;

int step = n - 1, i, f = 1;

while (step != 1 || f == 1)

{

step /= factor;

if (step < 1)

step = 1;

f = 0;

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

if (in[i] > in[i + step])

{

swp(&in[i], &in[i + step]);

f = 1;

}

}

}

void countbyte(unsigned long long\* mas, int n, int count[256], int byte) //Подсчёт байтов сортируемого разряда

{

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

int i, tmp1, tmp2, bias = sizeof(unsigned long long);

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

count[i] = 0;

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

count[masc[i \* bias + byte]]++;

tmp1 = count[0];

count[0] = 0;

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

{

tmp2 = count[i];

count[i] = count[i - 1] + tmp1;

tmp1 = tmp2;

}

}

void radix\_unsigned(unsigned long long\* mas, int n, unsigned long long\* mas\_tmp) //Поразрядная сортировка беззнаковых чисел

{

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

int count[256], sizetype = sizeof(unsigned long long), j, i;

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

{

countbyte(mas, n, count, i);

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

mas\_tmp[count[masc[j \* sizetype + i]]++] = mas[j];

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

{

mas[j] = mas\_tmp[j];

}

}

}

void radix\_double(double\* mas, int n, double\* tmp) //Поразрядная сортировка чисел типа double

{

int count = 0, i = n - 1;

radix\_unsigned((unsigned long long\*)mas, n, (unsigned long long\*)tmp);

while (i >= 0 && mas[i] < 0)

{

tmp[count++] = mas[i];

i--;

}

i = 0;

for (; count < n; count++)

tmp[count] = mas[i++];

}