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 3823Б1ПМ1-1

Золкин И.А.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2023

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 9](#_Toc26962566)

[Результаты экспериментов 1](#_Toc26962567)0

[Заключение 1](#_Toc26962568)3

[Приложение 14](#_Toc26962569)

# Постановка задачи

На вход программы дается последовательность чисел с плавающей точкой (double), программа должна корректно отсортировать заданную последовательность. Сортировка должна быть реализована четырьмя алгоритмами:

* Сортировкой выбором,
* Сортировкой расчёской,
* Сортировкой слиянием,
* Поразрядной сортировкой.

Реализованные алгоритмы необходимо проверить на корректность, а также провести тестирование алгоритма на временную сложность для проведения сравнения по данному критерию на основе полученных в результате тестирования данных.

Сравнение необходимо для выявления наиболее эффективного алгоритма сортировки.

# Метод решения

Сортировка выбором (selection\_sort):

Алгоритм формирует в массиве элементов две части - отсортированную и неотсортированную. Алгоритм идет по массиву пока в неотсортированной части не остается один элемент (т.к он единственный и наибольший, соответственно на этом моменте массив отсортирован). В каждой итерации алгоритм перемещает в отсортированную часть один следующий элемент из неотсортированной части и пробегает по неотсортированной части, сравнивая добавленный элемент с элементами из неотсортированной части, если находится элемент меньше последнего в отсортированной части, то они меняются местами. Т.е. Алгоритм сортирует многократным выбором наименьшего элемента в неотсортированной части массива и перемещением его в отсортированную часть массива.

Сортировка расчёской (comb\_sort):

Алгоритм является модификацией алгоритма “Пузырёк”. Алгоритм проходится по элементам, сравнивая элементы на некотором расстоянии, постепенно уменьшая расстояние между сравниваемыми элементами. В итоге на последней итерации алгоритм проходит по массиву полностью, но такая итерация в отличии от пузырька совершается единожды. Для уменьшения расстояния между сравниваемыми элементами используется константа, зависящая от золотого числа, которая уменьшает расстояние не пропуская элементы. Поэтому размер корректно сортируемого массива зависит от точности вычисленной константы. При отклонении от данной константы алгоритм может пропустить несколько чисел, не отсортировав их.

Сортировка слиянием (merge\_sort):

Изначально алгоритм рассматривает элементы массива отдельно друг от друга, берет каждые два соседние элемента и соединяет их в один подмассив в отсортированном порядке. Такое соединение выполняется двумя указателями. Первый указатель указывает на первый элемент (в первой итерации)/подмассив (в последующих итерациях). Запускается цикл, в котором элементы на указателях сравниваются между собой и добавляются в массив результата. Т.е если элемент i-й в подмассиве A < элемента j-го в подмасссиве B, то в массив результата записывается элемент i-й и указатель i увеличивается на 1. В противном случаем выполняется всё тоже самое только для подмассива B. Когда в одном из подмассивов заканчиваются элементы, мы перекладываем оставшиеся элементы из второго подмассива в результат. Последующие итерации производятся по такому же принципу, только для большей размерности подмассивов, пока они не сольются в массив изначального размера. Если исходный массив имеет размер 1, то сортировка не выполняется.

Поразрядная сортировка (radix\_sort\_double):

Алгоритм представляет числа в байтовом представлении, при проходе по каждому байту вызывается функция, которая подсчитывает количество значений байтов данного разряда и формирует из данных значений префиксную сумму. Дальше алгоритм по подсчитанным значениям восстанавливает элементы во вспомогательный массив. Т.к проход идет по определенному байту, то на каждой итерации внешнего цикла реализуется “сортировка подсчетом” по данному разряду. Дальше алгоритм меняет адреса начального массива и вспомогательного массива местами, чтобы работать со свежими элементами, так же заново вычисляется байтовое представление. После данных действий массив будет содержать сначала отсортированные положительные элементы (если были), а за ними отсортированные отрицательные элементы (если были), поэтому дальше идет правильное расположение элементов в вспомогательный массив, который в итоге копируется в изначальный.

# Руководство пользователя

Каждая сортировка представлена отдельным файлом, каждая работает по следующему принципу:

* В первой строке входных данных пользователь должен ввести целое положительное число современной арабской записью слева направо, отражающее количество чисел, находящихся в наборе, предназначенном для сортировки.
* Дальше пользователь должен ввести набор чисел, допустимы отрицательные, одинаковые, целые и не целые значения записанные в десятичной записи через точку. Числа, поступающие на вход программы должны быть разделены либо пробелом, либо нажатием “Enter”. Допустим ввод до 2 147 483 647 чисел, но фактическое количество зависит от характеристик вычислительной системы. Модуль введенных чисел должен находится в диапазоне от 2.2250738585072014 E - 308 до 1.7976931348623158 E + 308.
* Оригинальные программы выведут в консоль введенные числа в виде вещественного числа (например 1 выведется в виде 1.00000) в порядке неубывания.

# Описание программной реализации

Каталог содержит четыре подкаталога с разными сортировками:

* comb\_sort - содержит сортировку расческой.
* merge\_sort - содержит сортировку слиянием.
* selection\_sort - содержит сортировку выбором.
* Radix\_sort\_double - содержит поразрядную (LSD) сортировку.

Каждый подкаталог содержит несколько файлов, среди которых файл проекта \*.sln, файл с кодом \*.cpp.

Каждая программа начинает работу с функции main(), внутри реализовано считывание пользовательских данных: в переменную int n - количество чисел, считывание и хранение чисел в массиве double \*mas, вызов функции сортировки с аргументами (mas,n), вывод массива в консоль.

Функция void swap() принимает два указателя тип\* и меняет значения, на которые указывают указатели местами.

* **comb\_sort** содержит 3 функции: int main(), void swap(), void comb\_s().

**Функция void comb\_s(double\* mas,int n)** (запускается из main(), принимает два аргумента: указатель на сортируемый массив double\* mas и длинна массива int n) содержит переменные int step - длинна шага, double basis, factor - хранение коэффициента уменьшения шага, int j - счётчик для реализации цикла for(..;..;..){...} и цикл while(..){...}, который работает пока шаг не становится меньше одного, что значит, что за последнюю итерацию мы целиком прошлись по массиву. Внутри цикла while выполняется цикл for, который проходит по массиву до позиции, когда позиция + шаг будет вне массива, внутри for сравниваются и меняются в порядке неубывания значение на позиции и значение на (позиция + шаг). После цикла while выполняется уменьшение шага step в factor раз.

* **merge\_sort** содержит 5 функций: int main(), void merge\_s(), void merge(), int max(), int min().

**Функции int min(int a, int b)/int max(int a, int b)** принимают два числа int, сравнивают и возвращают минимальное/максимальное из них число int.

**Функция void merge(double\* a, double\* b, double\* res, int an, int bn)** принимает 5 аргументов: указатель на начало первого подмассива double \*a, указатель на начало второго подмассива double \*b, указатель на массив результата double \*res, длина первого подмассива int an, длина второго подмассива int bn. Внутри содержаться переменные int ia, ib, ir - позиции в первом и втором подмасссиве и массиве результата. Дальше идет цикл while(..){..}, работающий пока ia или ib не выйдут за диапазон. Внутри происходит сравнение элементов и тот который меньше попадает в массив результата. После цикла идут еще два цикла for, которые докладывают в res оставшиеся элементы.

**Функция void merge\_s(double \*mas, int n)** - функция, запускаемая из main(), принимает два аргумента: указатель на сортируемый массив double\* mas и длинна массива int n. Внутри выделяется память для хранения результатов функции merge() - double \*tmpdata, переменные для хранения шага (размера подмассивов) - int step, позиции начала подмассивов - int i, указатель для смены местами указателей на массивы mas и tmpdata - double \*temp. Дальше идет цик л for(..;..;..){...}, увеличивающий размер шага (длинну подмассивов), внутри которого работает еще один цикл for(..;..;..){...} бегущий по позициям начала каждого подмассива на данной итерации внешнего for и вызывающий функцию merge(). После внутреннего for mas и tmpdata меняются указателями, для того чтобы сортировка продолжалась для актуальных данных.

* **selection\_sort содержит 3 функции: int main(), void swap(), void selection\_s.**

**Функция void selection\_s(double\* ptr, int n) запускается из main(), принимает два аргумента: указатель на начало сортируемого массива - double\* ptr, длинна массива - int n. Внутри 3 переменные int i, j, minI - позиции элементов и позиция минимального элемента в несортированной части. Массив for(..;..;..){...} бежит по позициям, являющимся границей неотсортированной и отсортированной частей. Внутренний for предназначен для поиска позиции минимального элемента в неотсортированной части. После внутреннего for идет swap() для элементов на позиции i и позиции minI.**

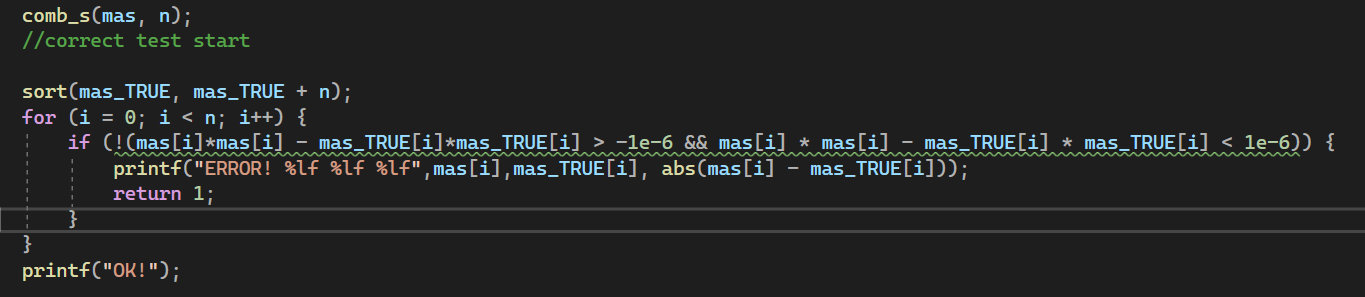
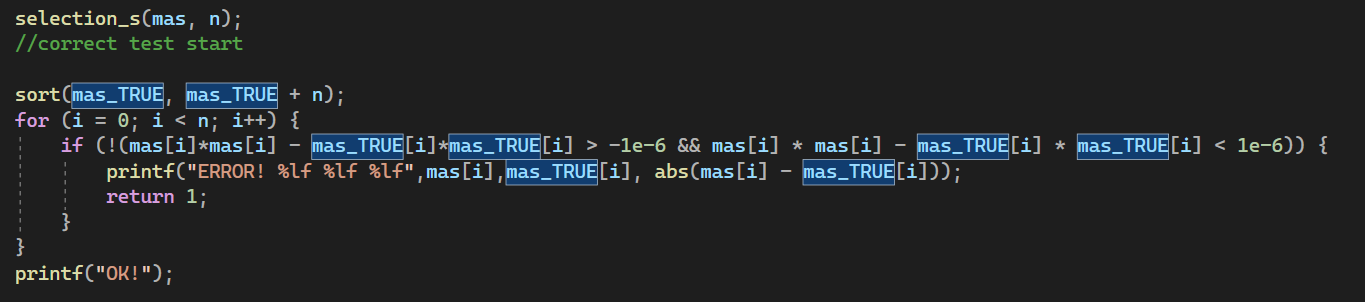
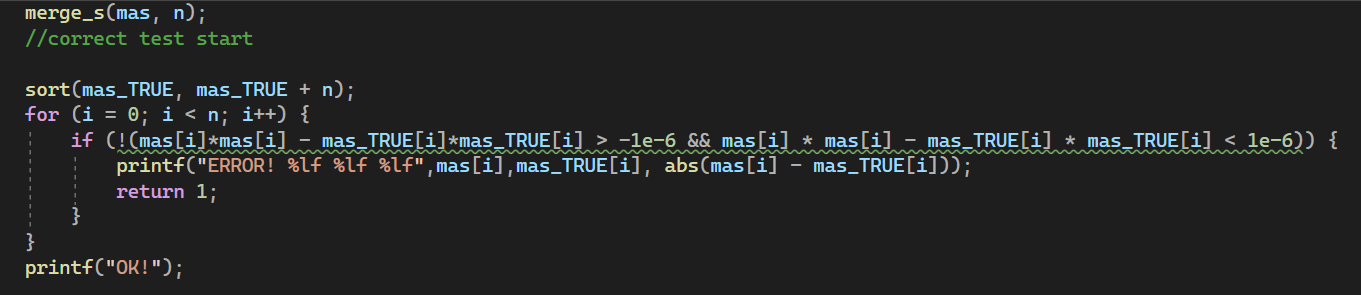
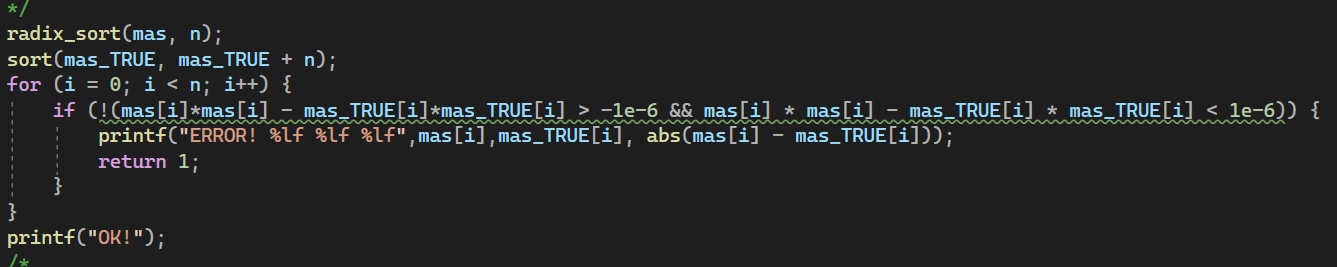
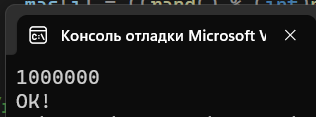
* **Radix\_sort\_double содержит 4 функции: int main(), void swap(), void foffset(), void radix\_sort().**

**Функция void foffset(unsigned char\* mas, int size, int offset, int count[257]) принимает на вход 4 аргумента: unsigned char\* mas - указатель на начало массива байтов, в который разложили изначальный массив, int size - длина изначального массива, int offset - номер байта каждого числа, int count[257] - массив для подсчета значений байт. Внутри переменные int i, tmp используются для хранения позиции начала элемента, формирования префикс-суммы в count[257] соответственно. Первый for(..;..;..){...} зануляет массив count. Следующий for идет по элементам (изначального массива) и в count подсчитывается количество байтов позиции offset каждого числа. Финальным шагом является построение префикс-суммы в count[257].**

**Функция void radix\_sort(double \*mas, int size) вызывается из main() и принимает 2 аргумента: указатель на начало сортируемого массива double \* mas и размер сортируемого массива int size. Unsigned char\* pm - массив байт массива mas, int count - массив подсчета для реализации “сортировки подсчетом” по разрядам, int i, j - номер байта каждого числа и номер числа соответственно, double \* mas2 - массив, в котором остается результат после “сортировки подсчетом” на каждой итерации внутреннего цикла for, unsigned int negativeDigits - количество отрицательных чисел, unsigned long long int byteP - представление числа в байтах для подсчета отрицательных чисел. Дальше идет цикл for(..;..;..){...} бегущий по байтам, для double в настоящий момент это от 0 до 8. На каждый байт вызывается функция foffset, формирующая массив count. Дальше в внутреннем цикле for реализуется “сортировка подсчетом” по i-му байту и результат записывается в массив mas2. После внутреннего цикла for меняем местами адреса mas и mas2, чтобы в mas лежали актуальные значения, а также меняем представление в байтах pm на актуальное. После внешнего for реализован подсчет отрицательных чисел, выделением бита, отвечающего за знак (это позволяет избежать операции сравнения). Следующими for складываем в mas2 отсортированные элементы в правильном порядке (т.к положительные и отрицательные элементы не отсортированы относительно друг друга) и перекопируем все в массив mas.**

# Подтверждение корректности

Для подтверждения корректности работы программ, использовалось сравнение отсортированного массива с отсортированным функцией sort() из #include <algorithm> массивом.

Для каждой программы проверка прошла успешно для достаточно большого количества входных данных (radix и merge - 1e6, selection - 1e5), исключение составила сортировка расческой, которая из-за погрешности вычисления коэффициента уменьшения шага не смогла корректно отсортировать большое кол-во чисел.

# Результаты экспериментов

По данным экспериментов видно, что для небольшого количества чисел (дальше n) сортировка выбором выполняется быстрее остальных, но уже к n = 100 самой быстрой сортировкой становится сортировка слиянием, которая при n<10000 будет не на много быстрее сортировки расческой. Начиная с n=500 самой быстрой сортировкой становится поразрядная сортировка.

Замер количества тиков процессора, сделанных за время исполнения сортировок проводился с помощью библиотеки #include "intrin.h":

a = \_\_rdtsc();

radix\_sort(mas, n);//замер тиков для поразрядной сортировки

b = \_\_rdtsc();

printf\_s("%I64d ticks\n", b - a);

После замера были получены следующие значения для разных сортировок:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***n*** | ***10*** | ***100*** | ***500*** | ***1000*** | ***10000*** | ***50000*** | ***100000*** |
| ***comb*** | ***11624*** | ***25638*** | ***92586*** | ***187334*** | ***2619224*** | ***15555400*** | ***33526141*** |
| ***merge*** | ***3534*** | ***17480*** | ***93606*** | ***195248*** | ***2366202*** | ***13723162*** | ***29101154*** |
| ***selection*** | ***2132*** | ***26202*** | ***359114*** | ***1152456*** | ***108181147*** | ***2634393511*** | ***10510856430*** |
| ***radix*** | ***26832*** | ***46070*** | ***87034*** | ***127360*** | ***968022*** | ***4472002*** | ***9387952*** |

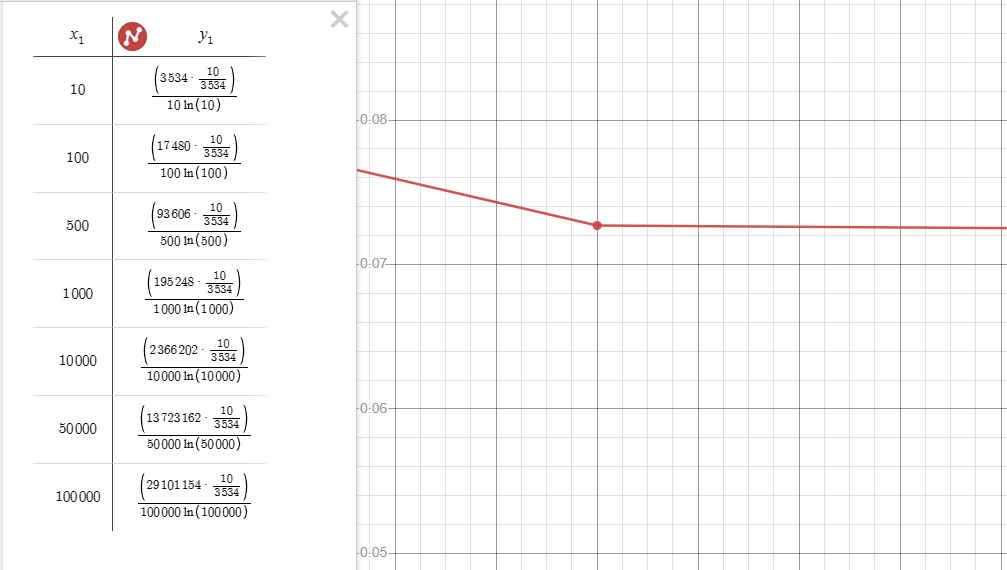
Построим графики используя формулу y = (n\*10/m)/(n\*ln(n)), где m - кол-во тиков при 10 n. Количество тиков умножается на коэффициент равный первому n деленному на кол-во тиков при первом n, это позволяет уменьшить разницу в масштабах осей. Знаменатель дроби основан на предположении, что сложность логарифмическая.

Проверяем данное предположение для сортировки расческой:



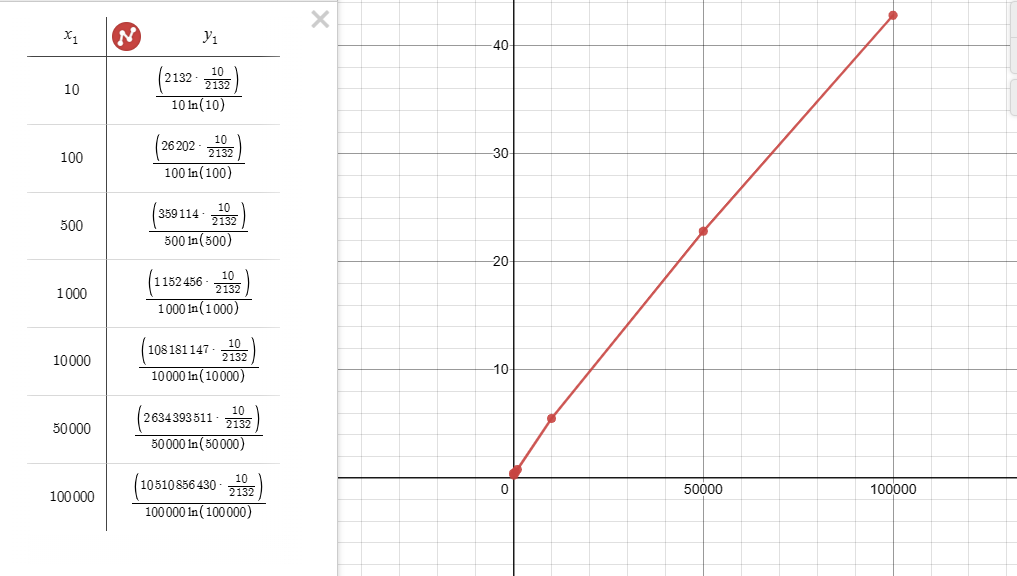
Из графика что функция, начиная с некоторого n стала принимать константу, значит, предположение оказалось верно и сложность алгоритма O(nlog(n))

Проверяем данное предположение для сортировки слиянием:



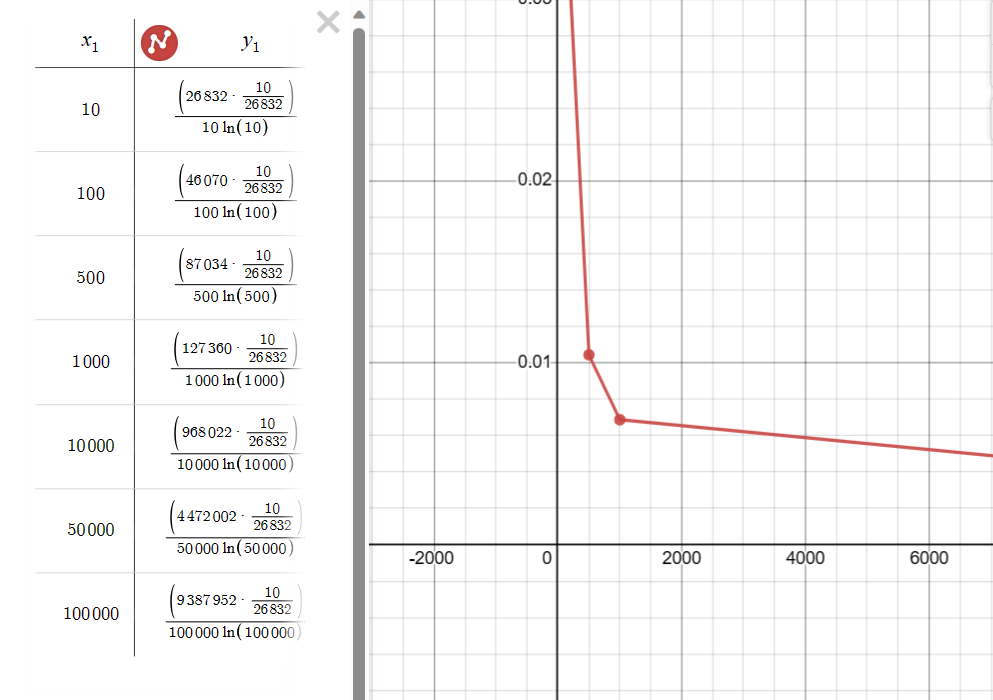
Видно, что начиная с некоторого n график принимает константу, предположение верно, сложность O(nlog(n))

Проверяем данное предположение для сортировки выбором:



Из графика видно, что предположение неверно, соответственно, функция растет быстрее nlog(n), значит это сложность O(n2).

Проверяем данное предположение для поразрядной сортировки:



Видно, что график отличается от nlog(n), убывает с асимптотами в осях координат, значит это то, что растет медленнее nlog(n) (при больших n), значит, сложность - O(n).

# Заключение

Реализованные алгоритмы были проверены на корректность и на временную сложность. В ходе исследования было выявлено:

* Сортировка расчёской требует большой точности вычисления коэффициента уменьшения шага, чем больше n, тем большая точность требуется.
* Для небольших n выгодно использовать простые алгоритмы, работающие за O(n2).
* Самая быстрая сортировка для достаточно больших n - поразрядная, работающая за O(n), но несмотря на линейную сложность, алгоритм требует дополнительных действий для формирования результата, т.к алгоритм сортирует поразрядно, а значение последнего рассматриваемого разряда отрицательных чисел больше, чем у положительных.

Полученные результаты доказывают, что, чем медленнее растет функция временной сложности алгоритма, тем быстрее сам алгоритм.

# Приложение

Сортировка выбором:

void selection\_s(double\* ptr, int n) {

int i = 0, minI, j;

for (i = 0; i < n - 1; i++) { //run for elements to swap

minI = i;

for (j = i; j < n; j++) { //finding the min element to the right of the selected

if (ptr[minI] > ptr[j]) {

minI = j;

}

}

swap(&ptr[i], &ptr[minI]);

}

}

Сортировка расчёской:

void comb\_s(double\* mas, int n) {

int step = n - 1;

double basis = (1 + sqrt(5)) / 2.0;

double factor = 1.0 / (1.0 - exp(-basis)); //"magic digit" - constant for step reduction

int j;

while (step >= 1) {

for (j = 0; j < n - step; j++) { //run for (0 to n-step) and compare values with step

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

swap(&mas[j], &mas[j + step]);

}

}

step /= factor;

}

}

Сортировка слиянием:

void merge(double\* a, double\* b, double\* res, int an, int bn)

{

int ia = 0, ib = 0, ir = 0;

while (an > ia && ib < bn)

{

if (a[ia] < b[ib]) {

res[ir++] = a[ia++];

}

else {

res[ir++] = b[ib++];

}

}

for (; ia < an; ia++) {

res[ir++] = a[ia];

}

for (; ib < bn; ib++) {

res[ir++] = b[ib];

}

}

void merge\_s(double\* mas, int n) {

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

int step = 1, i;

double\* temp;

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

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

merge(&mas[i], &mas[i + step], &tmpdata[i], max(0, min(step, n - i)), max(0, min(step, n - step - i)));

}

temp = mas;

mas = tmpdata;

tmpdata = temp;

}

}

Поразрядная сортировка:

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

int i = 0;

int tmp;

for (i = 0; i < 257; count[i++] = 0) {};

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

count[ //сounts byte values

mas[offset + i \* sizeof(double)] //i-byte every digit

]++;

}

tmp = count[0]; //bytes == 0

count[0] = 0;

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

swap(&tmp, &count[i + 1]);

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

}

}

void radix\_sort(double\* mas, int size) {

//start variables

unsigned char\* pm = (unsigned char\*)mas; //Representation of a number in bytes

int count[257];

int i = 0, j = 0;

double\* tempp;

double\* mas2 = (double\*)malloc(size \* sizeof(double));

unsigned int negativeDigits = 0;

unsigned long long int byteP;

//end variables

for (; i < sizeof(double); i++) { //run for bytes (double -- 8byte (now))

foffset(pm, size, i, count);

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

mas2[

count[

pm[j \* sizeof(double) + i]

]++

] = mas[j];

}

//start swap arrays

tempp = mas;

mas = mas2;

mas2 = tempp;

//end swap arrays

pm = (unsigned char\*)mas; //Representation of a number in bytes

}

//start counting negative elements

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

byteP = (unsigned long long int)mas[i];

negativeDigits += byteP >> 63;

}

//end counting negative elements

//start pushing elements in mas2

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

mas2[i] = mas[size - i - 1];

}

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

mas2[i] = mas[i - negativeDigits];

}

//end pushing elements in mas2

//start copy mas2 to mas

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

mas[i] = mas2[i];

}

//end copy mas2 to mas

free(mas2);

}