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 3822Б1ПМ1

Леонтьев Н.С.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2022

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 6](#_Toc26962564)

[Описание программной реализации 7](#_Toc26962565)

[Подтверждение корректности 9](#_Toc26962566)

[Результаты экспериментов 10](#_Toc26962567)

[Заключение 16](#_Toc26962568)

[Приложение 17](#_Toc26962569)

# Постановка задачи

В лабораторной работе необходимо реализовать четыре сортировки: сортировку выбором (Selection Sort), сортировку вставками (Insertion Sort), сортировку слиянием (Merge Sort), сортировку расчёской (Comb Sort). Все сортировки должны работать с типом данных double. Также необходимо проверить их достоверность и провести исследование: сравнить работу предоставленных алгоритмов сортировки сложностей   
.

Исходный код программы выложить на GitHub, предоставить в данном отчёте руководство пользователю, описание программной реализации.

# Метод решения

**1. Сортировка выбором**

Идея данной сортировки состоит в делении массива на отсортированную (изначально пустую) и неотсортированную часть и нахождении минимального (в случае сортировки по возрастанию) элемента в неотсортированной части N-1 раз, где N – длина массива. После нахождения этот элемент помещается в отсортированную часть справа от остальных (путём swap нужного элемента с первым элементом правой части) и поэтому не рассматривается при последующем выборе минимального.

Поиск минимального N-1 необязателен, но при поиске N раз на N итерации в правой части останется лишь один элемент – максимальный среди всех остальных. Следовательно его необязательно помещать в отсортированную часть, а значит массив отсортирован.

Сложность алгоритма: во всех случаях . Стабильная сортировка при правильной реализации.

**2. Сортировка вставками**

Идея данной сортировки также состоит в делении на отсортированную и неотсортированную часть, однако не производятся сравнения в неотсортированной части. Изначально в отсортированной части 1 элемент (официально 0, но первый элемент всегда будет стоять на своём месте) – первый в массиве. Каждую итерацию в её конец добавляется один элемент, где происходит упорядочение за счёт сравнения элемента с предыдущим и swap в случае неправильного порядка, и так пока элемент не окажется на своём месте. После N-1 итераций массив будет полностью отсортирован.

Сложность алгоритма: в лучшем случае , в среднем и худшем . Стабильная сортировка.

**3. Сортировка слиянием**

Подразумевает разбиение массива на пары, с целью последующего слияния их парами с использованием метода сортировки двух упорядоченных массивов. Это возможно при использовании двух указателей: тогда, используя дополнительную память, возможно сортировать два отсортированных массива, выбирая циклично с какого из них брать и помещать следующий элемент. Легче реализовать рекурсивно, сначала разбив, потом отсортировав от меньших пар массивов к большим.

Требует дополнительный массив N длины. Сложность алгоритма: всегда .

**4. Сортировка расчёской**

Это сортировка на основе сортировки пузырьком. Число, равное , – фактор, необходимый для лучшей скорости сортировки расчёской. Оно приблизительно равно 1,247. Заводится переменная step = N-1, и в случае неупорядоченности элементы i и i+step меняются местами, после чего step делится на фактор. Это повторяется, пока массив не станет полностью отсортированным (пока не окажется ситуации, что step = 1 и ни один элемент не поменяется).

Сложность алгоритма: в лучшем случае, в среднем , в худшем . Нестабильная сортировка.

# Руководство пользователя

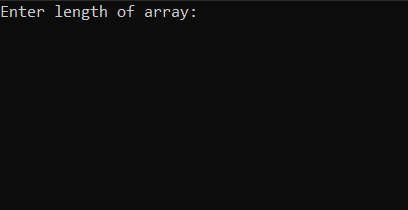
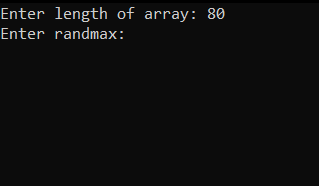
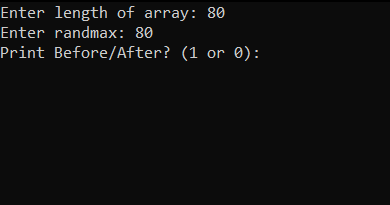
Необходимо запустить программу, код которой представлен внутри репозитория GitHub. Она запросит ввод от пользователя:

рис. 3. «Выводить процесс (до и после) сортировки? (1 или 0)»

рис. 2. «Введите верхнюю границу для элементов массива»

рис. 1. «Введите длину массива»

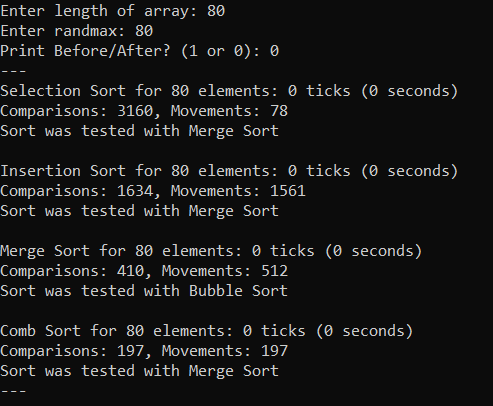


рис. 4. Результат работы программы

После ввода длины массива, его верней границы, типа вывода, программа выведет результаты четырёх разных сортировок, после чего она завершит свою работу.

# Описание программной реализации

Программа представлена в виде одного файла SortsLN.cpp. Чтобы обеспечить взаимодействие с пользователем, была придумана простая функция interface1 с прототипом:

void interface1(int \*arrSize, int \*ranMax, int \*choice);

Она спрашивает у пользователя три переменные: размер массива, верхнюю границу элементов массива и выбор, выводить или нет массивы в процессе.

После вызова интерфейса в куче выделяется память под массив нужного пользователю размера и управление переходит функции StartSort. Важным вопросом стало то, как для четырёх разных сортировок организовать примерно одинаковые действия и одинаковый вывод. Решением проблемы же стала возможность в C создавать указатели на функции с определёнными параметрами:

void(\*sorts[4])(double\*, int) = { selectionSort, insertionSort, mergeSort, combSort };

const char name[4][10] = { "Selection", "Insertion", "Merge", "Comb" };

Это позволило индексировать сортировки, и передавать их в функцию StartSort. Двумерный массив const char же был необходим для корректного вывода в консоль названий сортировок.

void StartSort(void (\*func) (double \*, int), const char\* name, double \*arr, int arrSize, clock\_t \*clock1, int check);

StartSort запрашивает: 1) указатель на сортировку; 2) название сортировки; 3) указатель на массив; 4) его размер; 5) указатель на тип clock\_t для подсчёта времени работы; 6) выводить или нет массивы в процессе (1 или 0).

Поблизости StartSort также были использованы две функции: fillarr и showarr. Первая заполняет массив случайными double элементами (учитывая верхнюю границу ranMax), вторая же выводит их на экран.

Для измерения времени работы каждой сортировки была подключена библиотека “time.h”. Время измеряется в тактах, но ради удобства также дополнительно переведено в секунды. Были использованы глобальные переменные countDo и countCmpr для подсчёта перемещений элементов и их сравнений.

Другие же функции являются функциями сортировки:

void selectionSort(double \*ptr, int len);

void insertionSort(double \*ptr, int len);

void merge(double \*arr, double \*tmp, int l, int m, int r);

void mergeSorting(double \*arr, double \*tmp, int l, int r);

void mergeSort(double \*ptr, int len);

void combSort(double \*ptr, int len);

void bubbleSort(double \*ptr, int len);

Использование всех написанных функций в main() сделало возможным создание программы, подходящей задачам лабораторной работы.

# Подтверждение корректности

Для подтверждения корректности сортировки слиянием был использован простой алгоритм Bubble Sort. Для остальных же сортировок использована уже проверенная сортировка слиянием.

Реализовано это было функцией test. Она имеет такой прототип:

int test(double \*arr, int len, int isMerge);

Принимая отсортированный массив, его длину, и проверку на сортировку слиянием, функция создаёт копию массива arr, после чего сортирует её проверенной сортировкой. После этого каждый элемент исходного массива сравнивается с каждым элементом точно отсортированного массива. Если находятся несоответствия, то функция возвращает 0, что говорит о некорректности сортировки. Если несоответствия не были найдены, функция вернёт 1. Тогда сортировка корректна.

# Результаты экспериментов

Необходимо проверить, действительно ли сортировки в написанной программе имеют сложности, указанные ранее.

Для этого проанализируем время, затраченное при разном количестве элементов, у каждого из алгоритмов сортировки.

Построим зависимость количества тиков (time.h) от количества элементов в тысячах.

Пока не совсем видно, что это сложность , поэтому разделим количество тиков на сложность алгоритма – N2.

С увеличением числа элементов выясняется, что такой график стремится к константе. Следовательно, сложность алгоритма правильна.

Сортировка вставками проверена так же.

Изменим обозначение для сортировки слиянием: снизу графика будут десятки тысяч элементов, вместо тысяч.

Разделим на N\*log(N):

В итоге график всё же стремится к константе.

Сортировка расчёской крайне неоднозначна, так что рассмотрим отдельные случаи: лучший, худший. Возьмём за единицу сотню тысяч элементов.

Лучший (массив уже отсортирован):

Разделим на N2:

Худший (массив обратно отсортирован):

Это доказывает, что все 4 сортировки: выбором, вставками, слиянием, расчёской – работают правильно.

# Заключение

В итоге поставленные задачи были выполнены. Была написана программа для исполнения заданных сортировок (код см. в Приложении), описаны взаимодействия программы с пользователем, структура программы, способ проверки корректности, приведены результаты экспериментов, согласующиеся с теоретическими данными.

# Приложение

https://github.com/NeiroYT/mp1-3822B1PM1

void combSort(double \*ptr, int len) {

int flag = 1;

double factor = 1.2473;

int step = len - 1;

while (flag == 1 || step != 1) {

flag = 0;

for (int i = 0; i + step < len; i++) {

if (ptr[i] > ptr[i + step]) {

swap(&ptr[i], &ptr[i + step]);

flag = 1;

countDo++; countCmpr++;

}

}

if (step > 1) {

step /= factor;

}

}

}

void mergeSort(double \*ptr, int len) {

double \*tmp = (double \*)malloc(len \* sizeof(double));

mergeSorting(ptr, tmp, 0, len - 1);

free(tmp);

}

void merge(double \*arr, double \*tmp, int l, int m, int r) {

int pi1, pi2;

int tmpindex = 0;

pi1 = l;

pi2 = m + 1;

while (pi1 <= m && pi2 <= r) {

countCmpr++;

if (arr[pi1] <= arr[pi2]) {

tmp[tmpindex++] = arr[pi1++];

}

else {

tmp[tmpindex++] = arr[pi2++];

}

}

while (pi1 > m && pi2 <= r) {

tmp[tmpindex++] = arr[pi2++];

}

while (pi2 > r && pi1 <= m) {

tmp[tmpindex++] = arr[pi1++];

}

for (int i = 0; i < tmpindex; i++) {

arr[l + i] = tmp[i];

countDo++;

}

}

void mergeSorting(double \*arr, double \*tmp, int l, int r) {

int m = l + (r - l) / 2;

if (l == r) {

return;

}

mergeSorting(arr, tmp, l, m);

mergeSorting(arr, tmp, m + 1, r);

merge(arr, tmp, l, m, r);

}

void insertionSort(double \*ptr, int len) {

for (int i = 1; i < len; i++) {

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

countCmpr++;

if (ptr[j] < ptr[j - 1]) {

swap(&ptr[j], &ptr[j - 1]);

countDo++;

}

else {

break;

}

}

}

}

void selectionSort(double \*ptr, int len) {

int minIndex;

for (int i = 0; i < len - 1; i++) {

minIndex = i;

for (int j = i + 1; j < len; j++) {

if (ptr[j] < ptr[minIndex]) {

minIndex = j;

}

countCmpr++;

}

if (minIndex != i) {

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

countDo++;

}

}

}

void swap(double \*p1, double \*p2) {

double temp = \*p1;

\*p1 = \*p2;

\*p2 = temp;

}