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 3824Б1ПМ1-1

Безносов А. С.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2024

**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 14](#_Toc26962568)

[Приложение 15](#_Toc26962569)

# Постановка задачи

Необходимо реализовать несколько алгоритмов сортировки массива, а именно сортировку слиянием (Merge sort), сортировку выбором (Selection Sort), сортировку расческой (Comb Sort) и поразрядную сортировку (Radix Sort) с заданными типом данных float. Также необходимо показать соответствие моей реализации алгоритмов и теоретической алгоритмической сложности.

# Метод решения

**Сортировка слиянием (Merge Sort)**

Алгоритм сортировки, который использует метод "разделяй и властвуй". Он делит массив на подмассивы, сортирует их и затем объединяет в единый отсортированный массив.

Исходный массив делится на два подмассива. Далее эти два подмассива делятся на два более маленьких подмассива и так далее. Это деление продолжается рекурсивно до тех пор, пока не останутся подмассивы размером в 1 элемент. Если массив содержит один или ноль элементов, то он считается отсортированным, и алгоритм завершает свою работу. На каждом уровне рекурсии каждая половина массива сортируется отдельно. После того, как две половины отсортированы, они сливаются в один отсортированный массив. Для данной манипуляции используется дополнительный массив, в который будут помещаться элементы из двух половин в отсортированном порядке.

Сложность: в лучшем случае – , в среднем случае – , в худшем случае -

**Сортировка выбором (Selection Sort)**

Алгоритм сортировки, который работает по принципу последовательного выбора минимального элементы (*но это в том случае, если нужна сортировка по возрастанию, иначе выбираем максимальный элемент*) из неотсортированной части массива и перемещаем его в отсортированную часть.

Алгоритм начинает свою работу с первой позиции (*элемента с нулевым индексом*) в массиве, который считается началом отсортированной части. Далее в неотсортированной части ищется минимальный элемент. Найденный минимальный элемент меняется местами с первым элементом неотсортированной части массива. Таким образом, он становится частью отсортированной части. Дальше граница отсортированной и неотсортированной частями массива сдвигается на один шаг вправо. Процесс повторяется для остальной неотсортированной части массива до тех пор, пока не останется элементов для сортировки.

Сложность: в лучшем случае – , в среднем случае – в худшем случае -

**Сортировка расческой (Comb Sort)**

Алгоритм сортировки, являющийся улучшенной версией сортировки пузырьком, который устраняет недочеты, например, медленную скорость при сортировке почти отсортированных массивов. Данный алгоритм использует концепцию “расчески”, что позволяет постепенно уменьшать расстояние между сравниваемыми элементами, более эффективно перемещая большие элементы к концу массива.

В начале определяется значение шага, равное размеру массива, и т.н. коэффициент уменьшения шага, равный 2. Далее устанавливается специальный флаг, изначально равный 1, отслеживающий обмен элементов в текущем проходе. В основном цикле рассчитывается новое значение шага, равное предыдущему значению шага, деленного на коэффициент уменьшения. Пока шаг не станет равен 1, мы будем его каждый раз делить на наш коэффициент. Флаг становится равным 0. Затем мы проходимся по массиву с использованием текущего шага. Мы сравниваем элементы, находящиеся на расстоянии “шага” друг от друга. Если элемент с индексом i больше элемента с индексом i + ”текущий шаг”, то они меняются местами и флаг становится равным 1. Если после какого-то прохода не было произведено обменов (*то есть флаг остался равен 0*), то массив отсортирован и алгоритм завершает свою работу.

Сложность: в лучшем случае – в среднем случае – , в худшем случае - .

**Поразрядная сортировка (Radix Sort)**

Алгоритм сортировки, который работает, обрабатывая данные по отдельным разрядам (или байтам). В случае сортировки массива чисел типа float мы имеем дело с 32-битными числами.

Алгоритм проходит через каждый байт, начиная с младшего до старшего, и использует сортировку подсчетом Fcount для упорядочивания элементов. В начале мы приводим указатель на массив float к указателю на unsigned char. Это позволяет работать с каждым байтом числа float по отдельности. В функции Fcount cоздается массив counter, хранящий число вхождений каждого возможного значения байта. В цикле мы проходимся по каждому элементу основного массива и увеличиваем счетчик для соответствующего байта. Переменная offset указывает нам, какой именно байт обрабатывается. Далее преобразуется массив counter так, чтобы он содержал фактические индексы для размещения элементов в выходном массиве. Это делается с помощью накопления значений в массиве. Потом, в основной функции RadixSort, создается массив counter и массив mas1 для хранения отсортированных элементов. В цикле проходимся через каждый байт числа float. Вызов функции Fcount выполняет подсчет вхождений для текущего байта. Далее в цикле элементы массива mas перемещаются во временный массив mas1 (*цикл идет в обратном порядке для стабильности сортировки*). Используем значения из массива counter для определения позиции каждого элемента. После распределения всех элементов в mas1 они копируются в оригинальный массив mas. Данный алгоритм не работает с отрицательными числами, поэтому я написал дополнительную функцию RadixSortWithSign. Она создает два дополнительных массива для положительных и отрицательных чисел и распределяет все элементы массива. Потом я вызываю функцию RadixSort для каждого дополнительного массива. Каждый элемент в массиве отрицательных чисел сортируется в порядке возрастания (то есть от наименьшего отрицательного до наибольшего), а массив с положительными сортируется как положено. Далее в цикле я копирую элементы двух массивов в исходный массив (*отрицательные числа копируются в обратном порядке*).

Сложность: в лучшем случае – в среднем случае – в худшем случае -

# Руководство пользователя

Для работы с программой необходимо ввести номер сортировки, результат работы которой вы хотите увидеть. Номер сортировки указан выше поля ввода (рис. 1). После выбора сортировки необходимо указать желаемый размер массива (рис 2). После ввода всех необходимых данных на экране появится время работы алгоритма в секундах и число операций, совершенных в алгоритме за время его работы (рис. 3).

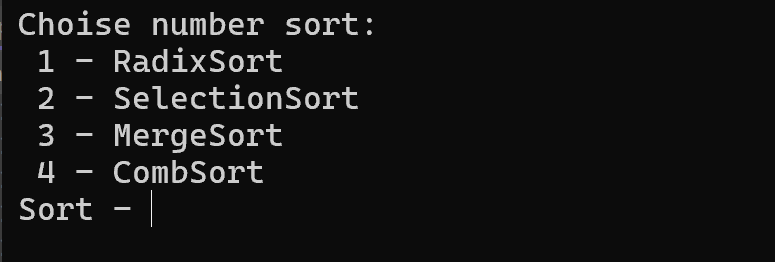


Рис. 1 Выбор сортировки

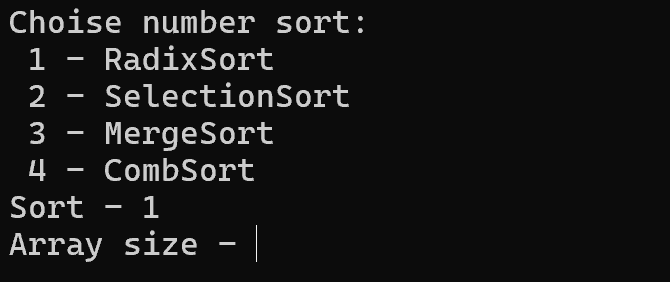


Рис. 2 Выбор размера массива

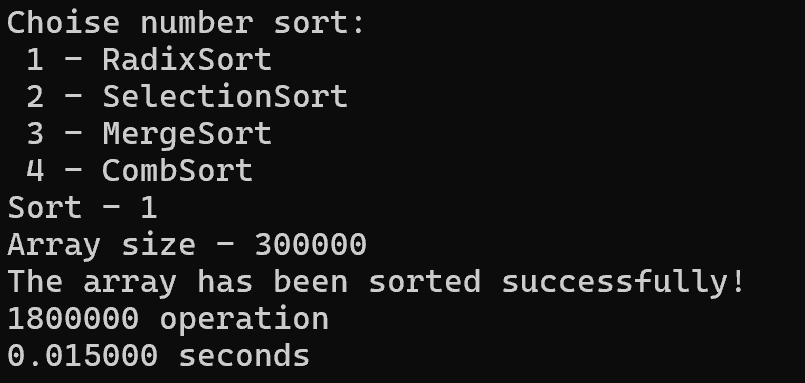


Рис. 3 Вывод

# Описание программной реализации

Вся моя программа включает в себя 3 файла. В первом файле “main.cpp” содержится главная функция main, в которой написан интерфейс программы. Во втором файле “source.cpp” находятся заголовки функций, реализованных в проекте. В третьем файле “Sourсe.h” представлены все основные функции программы. Здесь содержатся алгоритмы сортировок (RadixSortWithSign, MergeSort, SelectionSort, CombSort), функции подсчета времени (TimeOfWork, TimeOfWorkMerge), сложности работы алгоритмов (ComplexityOfWork, ComplexityOfWorkMerge, ComplexityOfWorkRadix) и функция подтверждения правильности сортировки (ConfirmationOfCorrectness, ConfirmationOfCorrectnessMerge). Для подсчета времени используется библиотека “time.h”.

Функция TimeOfWork принимает два параметра: указатель на функцию f, которая принимает массив float и его размер и размер массива типа int, который будет создан и передан в функцию f. В функции создается массив, который заполняется элементами типа float. Время начала выполнения функции f записывается в переменную start. Затем вызывается функция f, передавая ей массив и его размер. После завершения функции f, время окончания выполнения записывается в переменную finish. Для итогового подсчета времени находится разность между finish и start, деленная на число тактов в секунду. Результатом работы является время в секундах.

Здесь и далее функции с припиской Merge в названии будут выполнять те же самые действия, что и функции без приписки. Отличие лишь в том, что в этих функциях инициализируется не один массив, а два массива, один из которых заполняется нулями. Этого требует алгоритм сортировки слиянием.

Функция ComplexityOfWork принимает те же два параметра, что и функция TimeOfWork. В функции создается массив, заполняемый элементами типа float. Далее вызывается фунция f. В конце выводится значение счетчика count, который накапливал число операций в функции f.

Функция ConfirmationOfCorrectness принимает указатель на функцию f с двумя параметрами и размер массива. Создается два массива, оба заполняются случайными числами. Массив mas1 копирует значения из mas, чтобы иметь оригинальный неотсортированный массив для дальнейшей проверки. Вызывается функция сортировки f, передавая ей массив mas и его размер. Этот массив будет отсортирован с помощью переданной функции. Массив mas1 сортируется с помощью стандартной функции qsort. Для этого необходимо определить функцию сравнения cmp, которая должна соответствовать требованиям стандартной сортировки. Далее происходит сравнение элементов отсортированного массива mas с отсортированным массивом mas1. Если хотя бы один элемент не совпадает, то устанавливается флаг в 0, что указывает на то, что сортировка прошла некорректно. Если флаг так и остался равен 1, то сортировка произведена успешно и на экран выводится сообщение об успешной сортировке.

Прототипы функций, присутствующих в программе:

void Merge(float\* mas1, int size1, float\* mas2, int size2, float\* mas12);

void MergeSort(float\* mas, int size, float\* mas0);

void MergeComplexityOfWork(float\* mas1, int size1, float\* mas2, int size2, float\* mas12);

void MergeSortComplexityOfWork(float\* mas, int size, float\* mas0);

void SelectionSort(float\* mas, int size);

void SelectionSortComplexityOfWork(float\* mas, int size);

void CombSort(float\* mas, int size);

void CombSortComplexityOfWork(float\* mas, int size);

void Fcount(float\* mas, unsigned int counter[], int n, int offset);

void RadixSort(float\* mas, int size);

void RadixSortWithSign(float\* mas, int size);

void FcountComplexityOfWork(float\* mas, unsigned int counter[], int size, int offset);

void RadixSortComplexityOfWork(float\* mas, int size);

void RadixSortWithSignComplexityOfWork(float\* mas, int size);

void TimeOfWork(void (\*f)(float\* mas, int size), int size);

void TimeOfWorkMerge(void (\*f)(float\* mas, int size,float\* mas0), int size);

void ComplexityOfWork(void (\*f)(float\* mas, int size), int size);

void ComplexityOfWorkMerge(void (\*f)(float\* mas, int size, float\* mas0), int size);

void ComplexityOfWorkRadix(void (\*f)(float\* mas, int size), void (\*f1)(float\* mas, int size), int size);

int cmp(const void\* a, const void\* b);

void ConfirmationOfCorrectness(void (\*f)(float\* mas, int size), int size);

void ConfirmationOfCorrectnessMerge(void (\*f)(float\* mas, int size, float\* mas0), int size);

# Подтверждение корректности

Для подтверждения корректности своих сортировок я написал функцию ConfirmationOfCorrectness (см. Описание программной реализации на стр. 8), которая проверяет правильность алгоритмов сортировки, сравнивая их с результатами работы стандартной сортировки qsort.

# Результаты экспериментов

У того или иного алгоритма есть своя алгоритмическая сложность. Дабы убедиться в соответствии моей реализации алгоритмов сортировок и теоретической алгоритмической сложности этих сортировок, были составлены графики зависимости функций вида (T(n) / O(f(n))), где O(f(n)) – сложность работы соответствующего алгоритма. На рис. 1 представлен результат эксперимента с поразрядной сортировкой. Известно, что сложность работы этого алгоритма – O(n), то есть при устремлении числа элементов массива к бесконечности количество операций над массивом растет линейно. Поэтому если мы поделим число операций, совершенных при заданном n, на функцию, определяющую нашу сложность, то увидим на графике сначала некоторые колебания, а потом прямую линию. Например, при размере массива в 1 миллион элементов количество операций равно 6 миллионов при поразрядной сортировке. Поделив 6 миллионов на 1 миллион получаем 6. Эта тенденция будет продолжаться и при большем числе элементов. Из этого можно сделать вывод, что алгоритм сортировки соответствует своей теоретической алгоритмической сложности. Аналогично поступаем и с остальными алгоритмами сортировок. На рис. 2, 3, 4 представлены результаты работы с Merge Sort, Selection Sort и Comb Sort соответственно.

Рис. 1 Radix Sort

Рис. 2 Merge Sort

Рис. 3 Selection Sort

Рис. 4 Comb Sort

# Заключение

Я реализовал 4 алгоритма сортировки массивов с заданными типом данных float и показал соответствие моей реализации алгоритмов и теоретической алгоритмической сложности. Количество операций в каждой сортировке было успешно замерено, и они показали соответствие заявленным ожиданиям. Корректность сортировок также была подтверждена.

# Приложение

void Merge(float\* mas1, int size1, float\* mas2, int size2, float\* mas12) {

int j = 0, k = 0;

for (int i = 0; i < size1 + size2; i++) {

if (j == size1) {

mas12[i] = mas2[k++];

}

else {

if (k == size2) {

mas12[i] = mas1[j++];

}

else {

if (mas1[j] < mas2[k]) {

mas12[i] = mas1[j++];

}

else {

mas12[i] = mas2[k];

}

}

}

}

}

void MergeSort(float\* mas, int size, float\* mas0) {

if (size <= 1) {

return;

}

MergeSort(&mas[0], size / 2, &mas0[0]);

MergeSort(&mas[size / 2], size - size / 2, &mas0[size / 2]);

Merge(&mas[0], size / 2, &mas[size / 2], size - size / 2, &mas0[0]);

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

mas[i] = mas0[i];

}

}

void SelectionSort(float\* mas, int size) {

int i\_min;

float min;

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

min = mas[i];

i\_min = i;

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

if (min > mas[j]) {

i\_min = j;

min = mas[j];

}

}

float t = mas[i];

mas[i] = mas[i\_min];

mas[i\_min] = t;

}

}

void CombSort(float\* mas, int size) {

int gap = size;

int ratio = 2;

int flag = 1;

while (gap > 1 || flag) {

gap /= ratio;

if (gap < 1) {

gap = 1;

}

flag = 0;

for (int i = 0; i < size - gap; i++) {

if (mas[i] > mas[i + gap]) {

float t = mas[i];

mas[i] = mas[i + gap];

mas[i + gap] = t;

flag = 1;

}

}

}

}

void Fcount(float\* mas, unsigned int counter[], int size, int offset) {

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

memset(counter, 0, 256 \* sizeof(unsigned int));

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

counter[b[i \* sizeof(float) + offset]]++;

}

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

counter[i] += counter[i - 1];

}

}

void RadixSort(float\* mas, int size) {

unsigned int counter[256];

float\* mas1 = (float\*)malloc(size \* sizeof(float));

if (mas1 == NULL) {

return;

}

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

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

Fcount(mas, counter, size, i);

for (int j = size - 1; j >= 0; j--) {

mas1[counter[b[j \* sizeof(float) + i]] - 1] = mas[j];

counter[b[j \* sizeof(float) + i]]--;

}

memcpy(mas, mas1, size \* sizeof(float));

}

free(mas1);

}

void RadixSortWithSign(float\* mas, int size) {

float\* pos = (float\*)malloc(size \* sizeof(float));

float\* neg = (float\*)malloc(size \* sizeof(float));

int posCount = 0, negCount = 0;

if (pos == NULL || neg == NULL) {

return;

}

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

if (mas[i] >= 0) {

pos[posCount++] = mas[i];

}

else {

neg[negCount++] = mas[i];

}

}

RadixSort(neg, negCount);

RadixSort(pos, posCount);

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

mas[i] = neg[negCount - i - 1];

}

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

mas[negCount + i] = pos[i];

}

free(pos);

free(neg);

}