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/ка группы 3824Б1ПМ1

Кудряшова А.А.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2024

**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

Описание задачи: Реализация алгоритмов сортировки Выбором , Расческой, Слиянием, Поразрядная для чисел типа double.

Цель: Разработать и реализовать алгоритмы сортировки для упорядочивания массивов данных.

Задачи:

1. Изучение алгоритмов сортировок:

* Сортировка расческой(Brush Sort)
* Сортировка выбором (Selection Sort)
* Сортировка слиянием (Merge Sort)
* Поразрядная сортировка (Radix Sort)

2. Реализация алгоритмов:

* Написать код для каждого из выбранных алгоритмов на языке программирования C.
* Обеспечить возможность сортировки массивов для вещественных чисел двойной точности с плавающей точкой чисел.

3. Тестирование и валидация:

• Разработать функцию для проверки корректности работы каждого алгоритма.

• Проверить работу сортировок на уже отсортированных массивах.

4. Анализ производительности:

• Измерить время выполнения каждого алгоритма на различных объемах данных.

• Сравнить эффективность алгоритмов по времени выполнения и количеству операций.

• Проанализировать графики и сделать выводы о производительности программы.

# Метод решения

Алгоритм сортировки расческой— это улучшенная версия пузырьковой сортировки, которая использует концепцию "расчески" для более эффективного упорядочивания элементов массива. Он работает, сравнивая и меняя местами элементы, находящиеся на определенном расстоянии друг от друга, что позволяет быстрее устранять сортировать массив.

Описание алгоритма

1. Инициализация шага:

Алгоритм начинает с инициализации переменной step, которая равна половине длины массива. Этот шаг определяет, на каком расстоянии будут производиться сравнения и обмены элементов.

2. Циклы по шагам:

Внешний цикл продолжается до тех пор, пока step больше 1. На каждой итерации step уменьшается вдвое (делится на 2).

3. Сравнение и обмен:

Внутренние циклы выполняют сравнения и обмены элементов:

• Первый внутренний цикл проходит по всем элементам массива с индексом от 0 до step - 1.

• Второй внутренний цикл проходит через массив, начиная с текущего элемента i и перемещается с шагом step, сравнивая элементы на позициях j и j + step.

• Если элемент array[j] больше, чем array[j + step], они меняются местами.

4. Завершение:

Процесс повторяется до тех пор, пока шаг не станет равным 1, после чего массив будет отсортирован.

Сложность алгоритма

• Временная сложность: В среднем и худшем случае временная сложность сортировки расческой составляет O(n log n), что лучше, чем у обычной пузырьковой сортировки (O(n^2)).

Реализация:

int brush\_sort(double\* array, int length) {

int operationCount = 0; // Счетчик операций (обменов)

int step = length / 2; // Начальный шаг равен половине длины массива

// Основной цикл, который уменьшает шаг

for (; step > 1; step /= 2) {

// Внешний цикл по элементам с текущим шагом

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

// Внутренний цикл для сравнения элементов

for (int j = i; j + step < length; j += step) {

// Если текущий элемент больше элемента, находящегося на расстоянии step

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

// Меняем их местами

double t = array[j];

array[j] = array[j + step];

array[j + step] = t;

++operationCount; // Увеличиваем счетчик операций

}

}

}

}

// Проверка, отсортирован ли массив

if (!valid(array, length)) {

// Если массив не отсортирован, выполняем обычную сортировку пузырьком

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

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

// Сравниваем соседние элементы

if (array[j] > array[j + 1]) {

// Меняем их местами

double t = array[j];

array[j] = array[j + 1];

array[j + 1] = t;

operationCount++; // Увеличиваем счетчик операций

}

}

}

}

return operationCount; // Возвращаем количество операций

}

Алгоритм сортировки выбором

Принцип работы:

1. Итерация по массиву: Алгоритм проходит по массиву, начиная с первого элемента.

2. Поиск минимального элемента: На каждой итерации он находит минимальный элемент в оставшейся неотсортированной части массива.

3. Обмен местами: После того как минимальный элемент найден, он меняется местами с первым элементом неотсортированной части.

4. Повторение: Процесс повторяется для следующего элемента, пока весь массив не будет отсортирован.

Реализация:

void selection\_sort(double\* array, int length) {

for (int i = 0; i < length - 1; i++) { // Внешний цикл: перебираем элементы массива

int min = i; // Предполагаем, что текущий элемент — минимальный

for (int j = i + 1; j < length; j++) { // Внутренний цикл: ищем минимальный элемент в оставшейся части массива

if (array[j] < array[min]) // Если найден меньший элемент

min = j; // Обновляем индекс минимального элемента

}

// Обмен местами текущего элемента и найденного минимального

double t = array[i]; // Сохраняем текущее значение

array[i] = array[min]; // Заменяем текущее значение на минимальное

array[min] = t; // Помещаем сохраненное значение на место минимального

}

}

Сложность алгоритма

• Временная сложность: O(n²) , где n — количество элементов в массиве. Это связано с тем, что для каждого элемента требуется пройти по остальным элементам для поиска минимального.

Алгоритм сортировки слиянием (Merge Sort) — это эффективный, стабильный и основанный на принципе "разделяй и властвуй" алгоритм сортировки. Он разбивает массив на две половины, рекурсивно сортирует каждую из них и затем объединяет (сливает) отсортированные половины в один отсортированный массив.

Основные шаги алгоритма сортировки слиянием:

1. Разделение: Если массив содержит более одного элемента, он делится на две половины.

2. Рекурсивная сортировка: Каждая половина сортируется рекурсивно с помощью того же алгоритма.

3. Слияние: Две отсортированные половины объединяются в один отсортированный массив.

Реализация:

void merge\_sort(double\* array, int left, int right, int\* operationCount) {

if (left < right) {

int mid = left + (right - left) / 2; // Находим середину массива

merge\_sort(array, left, mid, operationCount); // Рекурсивно сортируем левую половину

merge\_sort(array, mid + 1, right, operationCount); // Рекурсивно сортируем правую половину

merge(array, left, mid, right, operationCount); // Сливаем две отсортированные половины

}

}

void count(double mas[], int counter[], int lenght, int offset) {

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

b = b + offset;

memset(counter, 0, 256 \* sizeof(int)); // Инициализация счетчика нулями

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

counter[(unsigned char)\*b]++; // Увеличиваем счетчик для текущего байта

b += sizeof(double); // Переходим к следующему элементу массива

}

int s = 0, tmp;

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

tmp = counter[i];

counter[i] = s; // Сохраняем текущее количество элементов до i

s += tmp; // Обновляем сумму

}

}

Сложность алгоритма

• Временная сложность: Во всех случаях временная сложность сортировки расческой составляет O(n log n).

Алгоритм поразрядной сортировки (Radix Sort)

Поразрядная сортировка — это алгоритм сортировки, который сортирует числа по отдельным разрядам. Он обычно применяется к целым числам или строкам, но в данном случае он адаптирован для сортировки чисел с плавающей точкой (типа double). Основная идея заключается в следующем:

Сортировка по разрядам: Сначала сортируем числа по младшему разряду (LSB ), затем по следующему разряду и так далее, пока не отсортируем по старшему разряду (MSB).

int radix\_sort(double\* array, int length) {

int operationCount = 0; // Счетчик операций для анализа производительности

int counter[256]; // Массив для подсчета количества элементов (для 256 возможных значений байтов)

double\* b = (double\*)malloc(length \* sizeof(double)); // Временный массив для хранения отсортированных значений

char\* c = (char\*)array; // Приведение указателя на массив double к указателю на char для доступа к байтам

for (int i = 0; i < sizeof(double); i++) { // Проходим по каждому байту double

count(array, counter, length, i); // Вызываем функцию count для подсчета количества элементов по текущему байту

for (int j = 0; j < length; j++) {

operationCount++; // Увеличиваем счетчик операций

b[counter[(unsigned char)\*(c + i + j \* sizeof(double))]++] = array[j]; // Сортируем элементы в массив b

}

for (int j = 0; j < length; j++) {

operationCount++; // Увеличиваем счетчик операций

array[j] = b[j]; // Копируем отсортированные элементы обратно в исходный массив

}

}

int index;

for (index = 0; index < length; index++) {

if (b[index] < 0.0) { // Находим индекс первого отрицательного числа

operationCount++; // Увеличиваем счетчик операций

break;

}

}

for (int i = 0; i < length - index; i++) {

operationCount++; // Увеличиваем счетчик операций

array[i] = b[length - 1 - i]; // Копируем отрицательные числа в исходный массив в обратном порядке

}

for (int i = length - index; i < length; i++) {

operationCount++; // Увеличиваем счетчик операций

array[i] = b[i - length + index]; // Копируем положительные числа в исходный массив

}

free(b); // Освобождаем память, выделенную для временного массива

return operationCount; // Возвращаем количество операций

}

Сложность алгоритма

• Временная сложность:В общем случае, алгоритм поразрядной сортировки имеет линейную сложность O(n), если количество разрядов и диапазон значений фиксированы и не зависят от количества элементов.

# Руководство пользователя

Для работы с программой необходимо:

\* Запустить исполняемый файл.

\* Программа запросит ввод размера массива. Необходимо ввести целое число и нажать Enter.

\* Программа сгенерирует случайный массив чисел с плавающей точкой и отсортирует его каждым из реализованных алгоритмов.

\* Для каждого алгоритма будет выведено сообщение о корректности сортировки, количество выполненных операций и время выполнения.

# Описание программной реализации

Проект состоит из одного файла main.c, который содержит:

\* Функции сортировки: brush\_sort, selection\_sort, merge\_sort, radix\_sort.

\* Вспомогательные функции: valid, merge, count.

\* Функцию main, которая отвечает за ввод размера массива, генерацию случайных данных, вызов функций сортировки, измерение времени выполнения и вывод результатов.

Подключение библиотек

#include <stdio.h>

#include <random>

#include <time.h>

• stdio.h: стандартная библиотека для ввода-вывода.

• random: библиотека для генерации случайных чисел.

• time.h: библиотека для работы со временем.

Вспомогательные функции:

Функция count выполняет подсчет количества элементов для каждого значения байта на определенной позиции (разряде) в числах массива. Она используется для сортировки чисел по текущему разряду.

Функция valid проверяет, отсортирован ли массив array в порядке возрастания.

Функция merge объединяет два подмассива в один отсортированный массив. Это часть алгоритма сортировки слиянием (merge sort).

# Подтверждение корректности

Функция valid в коде предназначена для проверки, отсортирован ли массив чисел в порядке возрастания.

Параметры функции

• const double\* array: указатель на массив double, который нужно проверить. Использование const указывает на то, что функция не будет изменять содержимое массива.

• int lenght: длина (количество элементов) массива, который необходимо проверить.

Логика работы функции

• Функция использует цикл for, который проходит по всем элементам массива от индекса 0 до lenght - 2. Это делается для того, чтобы сравнить каждый элемент с последующим (элемент с индексом i сравнивается с элементом с индексом i + 1).

• Внутри цикла происходит сравнение двух соседних элементов: array[i] и array[i + 1].

• Если текущий элемент (array[i]) больше следующего элемента (array[i + 1]), это означает, что массив не отсортирован. В этом случае функция немедленно возвращает 0 (или false).

• Если цикл завершился без нахождения пары элементов, где текущий элемент больше следующего, это означает, что массив отсортирован. В этом случае функция возвращает 1 (или true).

Функция valid служит для проверки корректности сортировки массива. Она возвращает:

• 1 (или true) — если массив отсортирован в порядке возрастания.

• 0 (или false) — если массив не отсортирован.

Реализация:

bool valid(const double\* array, int lenght) {

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

if (array[i] > array[i + 1])

return 0;

}

return 1;

}

# Результаты экспериментов

Результаты экспериментов

Результаты экспериментов будут зависеть от характеристик компьютера, на котором запускается программа, и от входных данных (размера массива). В общем случае можно ожидать, что:

\* Сортировка выбором будет показывать относительно низкую производительность на больших массивах.

\* Сортировка слиянием будет демонстрировать хорошую производительность на больших массивах, так как имеет сложность O(n log n).

\* Поразрядная сортировка может быть очень эффективной для определенных типов данных, но ее производительность может зависеть от распределения данных.

\* Сортировка расческой, как правило, работает быстрее сортировки пузырьком, но может проигрывать более продвинутым алгоритмам на больших массивах.

# Заключение

В результате выполнения работы были реализованы четыре алгоритма сортировки. Проведена проверка корректности их работы и измерено время выполнения. Полученные результаты позволяют сравнить эффективность различных алгоритмов на практике.

# Приложение

#include <stdio.h>

#include <random>

#include <time.h>

bool valid(const double\* array, int lenght) {

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

if (array[i] > array[i + 1])

return 0;

}

return 1;

}

int brush\_sort(double\* array, int lenght) {

int operationCount = 0;

int step = lenght / 2;

for (; step > 1; step /= 2) {

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

for (int j = i; j + step < lenght; j += step) {

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

double t = array[j];

array[j] = array[j + step];

array[j + step] = t;

++operationCount;

}

}

}

}

if (!valid(array, lenght)) {

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

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

if (array[j] > array[j + 1]) {

double t = array[j];

array[j] = array[j + 1];

array[j + 1] = t;

operationCount++;

}

}

}

}

return operationCount;

}

int selection\_sort(double\* array, int lenght) {

int operationCount = 0;

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

int min = i;

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

if (array[j] < array[min]) {

min = j;

operationCount++;

}

}

double t = array[i];

array[i] = array[min];

array[min] = t;

}

return operationCount;

}

void merge(double\* array, int left, int mid, int right, int\* operationCount) {

double\* lArray = (double\*)malloc(sizeof(double) \* (mid - left + 1));

double\* rArray = (double\*)malloc(sizeof(double) \* (right - mid));

for (int i = 0; i < mid - left + 1; i++)

lArray[i] = array[left + i];

for (int i = 0; i < right - mid; i++)

rArray[i] = array[mid + 1 + i];

int lIndex = 0, rIndex = 0, mIndex = left;

while (lIndex < mid - left + 1 && rIndex < right - mid) {

if (lArray[lIndex] < rArray[rIndex]) {

array[mIndex] = lArray[lIndex];

lIndex++;

(\*operationCount)++;

}

else {

array[mIndex] = rArray[rIndex];

rIndex++;

(\*operationCount)++;

}

mIndex++;

}

while (lIndex < mid - left + 1) {

array[mIndex] = lArray[lIndex];

lIndex++;

mIndex++;

(\*operationCount)++;

}

while (rIndex < right - mid) {

array[mIndex] = rArray[rIndex];

rIndex++;

mIndex++;

(\*operationCount)++;

}

free(lArray);

free(rArray);

}

void merge\_sort(double\* array, int left, int right, int\* operationCount) {

if (left < right) {

int mid = left + (right - left) / 2;

merge\_sort(array, left, mid, operationCount);

merge\_sort(array, mid + 1, right, operationCount);

merge(array, left, mid, right, operationCount);

}

}

void count(double mas[], int counter[], int lenght, int offset) {

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

b = b + offset;

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

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

counter[(unsigned char)\*b]++;

b += sizeof(double);

}

int s = 0, tmp;

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

tmp = counter[i];

counter[i] = s;

s += tmp;

}

}

int radix\_sort(double\* array, int lenght) {

int operationCount = 0;

int counter[256];

double\* b = (double\*)malloc(lenght \* sizeof(double));

char\* c = (char\*)array;

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

count(array, counter, lenght, i);

for (int j = 0; j < lenght; j++) {

operationCount++;

b[counter[(unsigned char)\*(c + i + j \* sizeof(double))]++] = array[j];

}

for (int j = 0; j < lenght; j++) {

operationCount++;

array[j] = b[j];

}

}

int index;

for (index = 0; index < lenght; index++) {

if (b[index] < 0.0) {

operationCount++;

break;

}

}

for (int i = 0; i < lenght - index; i++) {

operationCount++;

array[i] = b[lenght - 1 - i];

}

for (int i = lenght - index; i < lenght; i++) {

operationCount++;

array[i] = b[i - lenght + index];

}

free(b);

return operationCount;

}

int main() {

srand(time(0));

int lenght, operation;

printf("Enter array size: ");

scanf\_s("%d", &lenght);

clock\_t start, end;

double\* array = (double\*)malloc(sizeof(double) \* lenght);

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

array[i] = (double)rand() / (double)(RAND\_MAX / 100);

}

start = clock();

operation = brush\_sort(array, lenght);

end = clock();

printf("\nThe brush sorting is done correctly: %d, operation count: %d, time: %f\n", valid(array, lenght), operation, (float)(end - start) / CLOCKS\_PER\_SEC);

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

array[i] = (double)rand() / (double)(RAND\_MAX / 100);

}

start = clock();

operation = selection\_sort(array, lenght);

end = clock();

printf("\nThe selection sorting is done correctly: %d, operation count: %d, time: %f\n", valid(array, lenght), operation, (float)(end - start) / CLOCKS\_PER\_SEC);

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

array[i] = (double)rand() / (double)(RAND\_MAX / 100);

}

start = clock();

operation = 0;

merge\_sort(array, 0, lenght, &operation);

end = clock();

printf("\nThe merge sorting is done correctly: %d, operation count: %d, time: %f\n", valid(array, lenght), operation, (float)(end - start) / CLOCKS\_PER\_SEC);

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

array[i] = (double)rand() / (double)(RAND\_MAX / 100);

}

start = clock();

operation = radix\_sort(array, lenght);

end = clock();

printf("\nThe radix sorting is done correctly: %d, operation count: %d, time: %f\n", valid(array, lenght), operation, (float)(end - start) / CLOCKS\_PER\_SEC);

free(array);

}