Министерство науки и высшего образования Российской Федерации
Калужский филиал
федерального государственного бюджетного
образовательного учреждения высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *ИУК "Информатика и управление"*

КАФЕДРА *ИУК5 "Системы обработки информации"*

ЛАБОРАТОРНАЯ РАБОТА №4

«Сортировки.»

ДИСЦИПЛИНА: «Вычислительные алгоритмы»

Выполнил: студент гр.ИУК5-41Б _____ (____Шиндин А.О.___)
(Подпись) (Ф.И.О.)

Проверил: _____ (____ Вершинин В.Е__)
(Подпись) (Ф.И.О.)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка:
- Оценка:

Калуга, 2023

Цель: формирование практических навыков разработки алгоритма сортировки.

Задачи:

1. Создать блок-схему алгоритма
2. Реализовать заданный алгоритм
3. Измерить время выполнения алгоритма
4. Исследовать вычислительную сложность алгоритма

Задание:

Вариант 13:

Написать программу, сортирующую элементы двумерного массива: в нечетных столбцах по возрастанию, в четных – по убыванию. Использовать сортировку подсчетом. Диапазон целых чисел: [0; N]

Ход работы:

1) Блок схема алгоритма сортировки подсчётом:

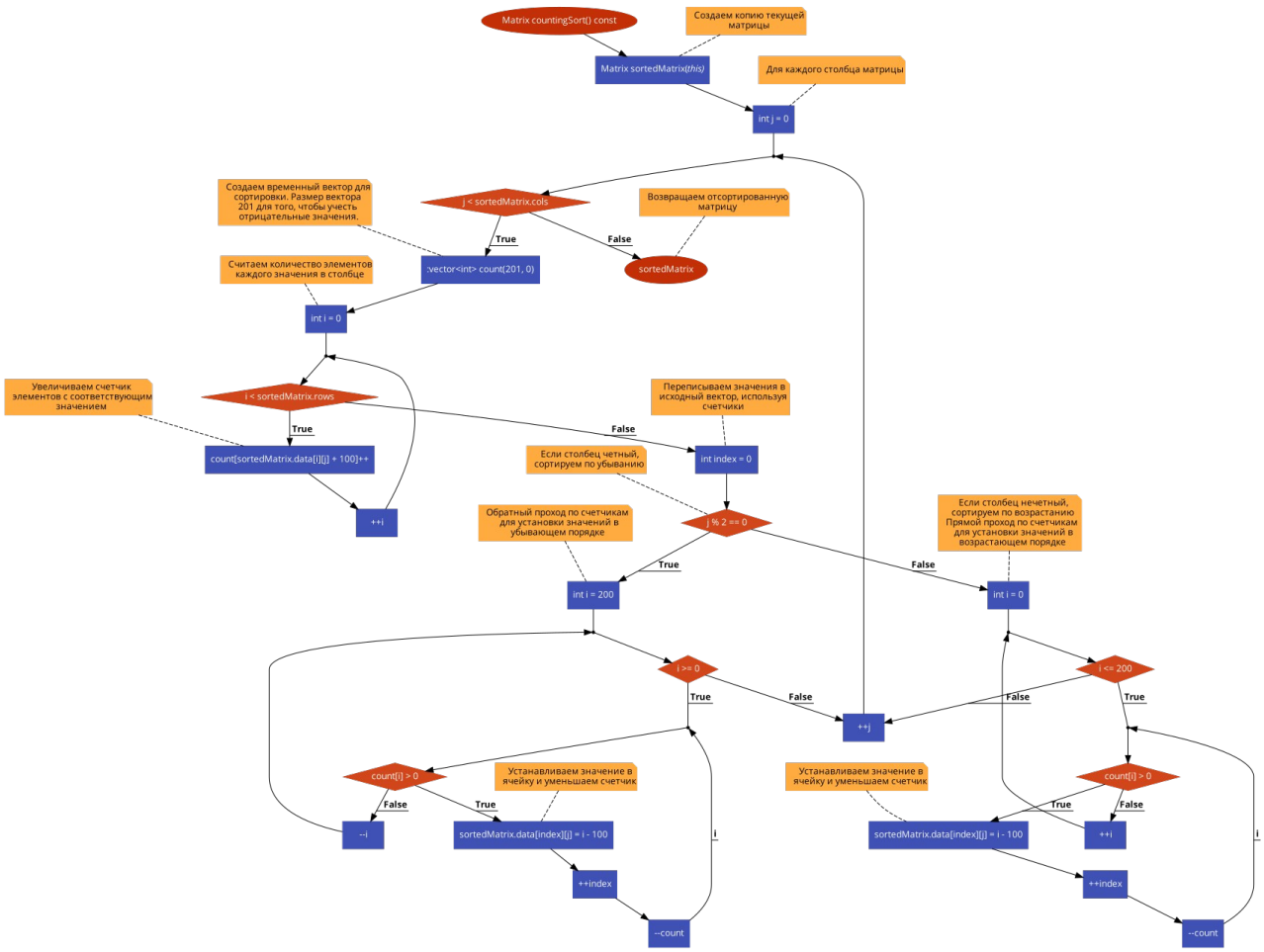
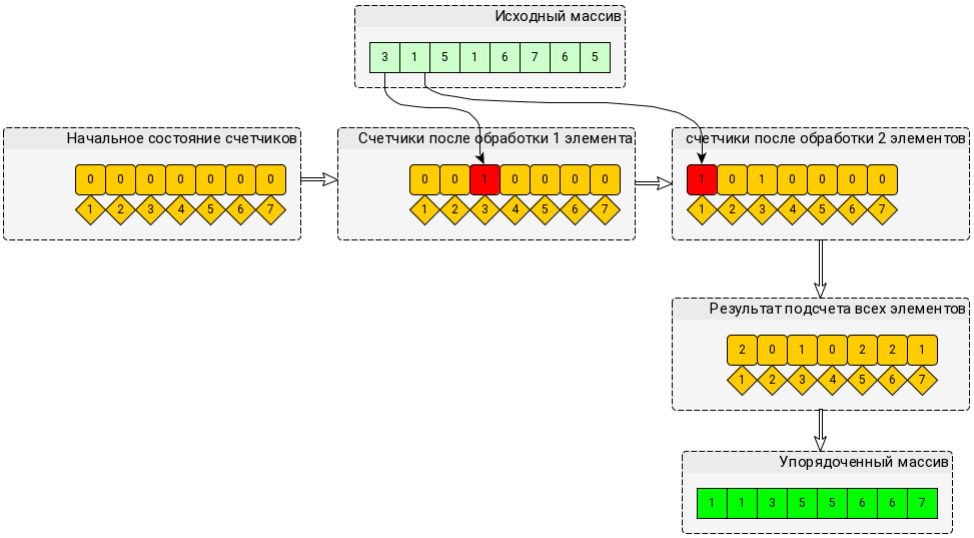


схема работы примитивного алгоритма сортировки подсчётом:



2) Программа написанная на языке c++:

```
#include <iostream>

#include <vector>
#include <string>
#include <chrono>
#include <cmath>


class Matrix {
public:
std::vector<std::vector<int>> data;
std::string name;
int rows;
int cols;


Matrix(int rows, int cols, std::string name) : rows(rows), cols(cols), name(name) {
// Инициализация матрицы нулями
data.resize(rows, std::vector<int>(cols, 0));
}


Matrix(const Matrix& other) : rows(other.rows), cols(other.cols), name(other.name) {
// Копирование данных из другой матрицы
data = other.data;
}
```

```
void printMatrix() const {
std::cout << "Matrix " << name << ":\n";
if ((rows < 15)&&(cols < 15)) {
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
std::cout << data[i][j] << '\t';
}
std::cout << "\n";
}
} else {
std::cout << "\nPrinting terminated! The matrix size is too large!\n";
}
}

void randomizeMatrix() {
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
data[i][j] = rand() % 201 - 100;
}
}
}

Matrix copyMatrix(std::string _name) const {
// Создаем новый объект матрицы
Matrix copiedMatrix(rows, cols, _name);

// Копируем данные из текущей матрицы в новую
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
copiedMatrix.data[i][j] = data[i][j];
}
}

return copiedMatrix;
}

// Метод сортировки выбором
```

```
Matrix selectionSort() const {
// Создаем копию текущей матрицы
Matrix sortedMatrix(*this);


// Для каждого столбца матрицы
for (int j = 0; j < sortedMatrix.cols; ++j) {
// Если столбец четный, сортируем по убыванию
if (j % 2 == 0) {
// Проходим по всем строкам столбца, кроме последней
for (int i = 0; i < sortedMatrix.rows - 1; ++i) {
// Ищем индекс максимального элемента в столбце
int max_index = i;
// Проходим по оставшимся строкам для поиска максимума
for (int k = i + 1; k < sortedMatrix.rows; ++k) {
// Если текущий элемент больше максимума, обновляем индекс максимума
if (sortedMatrix.data[k][j] > sortedMatrix.data[max_index][j]) {
max_index = k;
}
}
// Обмениваем местами элементы, если нужно
if (max_index != i) {
int temp = sortedMatrix.data[i][j];
sortedMatrix.data[i][j] = sortedMatrix.data[max_index][j];
sortedMatrix.data[max_index][j] = temp;
}
}
} else { // Если столбец нечетный, сортируем по возрастанию
// Проходим по всем строкам столбца, кроме последней
for (int i = 0; i < sortedMatrix.rows - 1; ++i) {
// Ищем индекс минимального элемента в столбце
int min_index = i;
// Проходим по оставшимся строкам для поиска минимума
for (int k = i + 1; k < sortedMatrix.rows; ++k) {
// Если текущий элемент меньше минимума, обновляем индекс минимума
if (sortedMatrix.data[k][j] < sortedMatrix.data[min_index][j]) {
min_index = k;
}
}
// Обмениваем местами элементы, если нужно
```

```
if (min_index != i) {
int temp = sortedMatrix.data[i][j];
sortedMatrix.data[i][j] = sortedMatrix.data[min_index][j];
sortedMatrix.data[min_index][j] = temp;
}
}
}
}
// Возвращаем отсортированную матрицу
return sortedMatrix;
}

// Метод сортировки LSD Radix
Matrix lsdRadixSort() const {
// Создаем копию текущей матрицы для сортировки
Matrix sortedMatrix(*this);

// Реализуем сортировку LSD Radix Sort для каждого столбца
for (int j = 0; j < sortedMatrix.cols; ++j) {
// Подготавливаем корзины для каждой цифры
std::vector<std::vector<int>> buckets(10);

// Находим максимальное число разрядов в столбце
int maxDigits = 1;
for (int i = 0; i < sortedMatrix.rows; ++i) {
int num = sortedMatrix.data[i][j];
int digits = 1 + static_cast<int>(log10(abs(num)));
maxDigits = std::max(maxDigits, digits);
}

// Сортировка по каждому разряду
for (int digit = 0; digit < maxDigits; ++digit) {
// Распределение чисел по корзинам
for (int i = 0; i < sortedMatrix.rows; ++i) {
int num = sortedMatrix.data[i][j];
int digitValue = (abs(num) / static_cast<int>(pow(10, digit))) % 10;
buckets[digitValue].push_back(num);
```

```
}

// Сборка чисел обратно в матрицу
int row = 0;
if (j % 2 == 0) { // Четный столбец: сортировка по убыванию
for (int k = 9; k >= 0; --k) {
for (int num : buckets[k]) {
sortedMatrix.data[row++][j] = num;
}
}
} else { // Нечетный столбец: сортировка по возрастанию
for (int k = 0; k < 10; ++k) {
for (int num : buckets[k]) {
sortedMatrix.data[row++][j] = num;
}
}
}

// Очистка корзин перед следующей итерацией
for (auto& bucket : buckets) {
bucket.clear();
}
}
}

// Возвращаем отсортированную матрицу
return sortedMatrix;
}

// Метод сортировки четных и нечетных столбцов матрицы с использованием пузырьковой сортировки
Matrix bubbleSort() const {
// Создаем копию текущей матрицы для сортировки
Matrix sortedMatrix(*this);

// Для каждого столбца матрицы
for (int j = 0; j < sortedMatrix.cols; ++j) {
```

```
// Если индекс столбца четный, сортируем по убыванию
if (j % 2 == 0) {
// Сортировка по убыванию с помощью алгоритма сортировки пузырьком
for (int i = 0; i < sortedMatrix.rows - 1; ++i) {
for (int k = 0; k < sortedMatrix.rows - i - 1; ++k) {
// Сравниваем элементы и меняем местами, если необходимо
if (sortedMatrix.data[k][j] < sortedMatrix.data[k + 1][j]) {
// Обмен значениями
int temp = sortedMatrix.data[k][j];
sortedMatrix.data[k][j] = sortedMatrix.data[k + 1][j];
sortedMatrix.data[k + 1][j] = temp;
}
}
}
} else { // Если индекс столбца нечетный, сортируем по возрастанию
// Сортировка по возрастанию с помощью алгоритма сортировки пузырьком
for (int i = 0; i < sortedMatrix.rows - 1; ++i) {
for (int k = 0; k < sortedMatrix.rows - i - 1; ++k) {
// Сравниваем элементы и меняем местами, если необходимо
if (sortedMatrix.data[k][j] > sortedMatrix.data[k + 1][j]) {
// Обмен значениями
int temp = sortedMatrix.data[k][j];
sortedMatrix.data[k][j] = sortedMatrix.data[k + 1][j];
sortedMatrix.data[k + 1][j] = temp;
}
}
}
}
}


// Возвращаем отсортированную матрицу
return sortedMatrix;
}


// Метод сортировки подсчетом
Matrix countingSort() const {
// Создаем копию текущей матрицы
Matrix sortedMatrix(*this);
```

```
// Для каждого столбца матрицы
for (int j = 0; j < sortedMatrix.cols; ++j) {
// Создаем временный вектор для сортировки. Размер вектора 201 для того, чтобы учесть отрицательные значения.
std::vector<int> count(201, 0);

// Считаем количество элементов каждого значения в столбце
for (int i = 0; i < sortedMatrix.rows; ++i) {
// Увеличиваем счетчик элементов с соответствующим значением
count[sortedMatrix.data[i][j] + 100]++;
}

// Переписываем значения в исходный вектор, используя счетчики
int index = 0;
// Если столбец четный, сортируем по убыванию
if (j % 2 == 0) {
// Обратный проход по счетчикам для установки значений в убывающем порядке
for (int i = 200; i >= 0; --i) {
while (count[i] > 0) {
// Устанавливаем значение в ячейку и уменьшаем счетчик
sortedMatrix.data[index][j] = i - 100;
++index;
--count[i];
}
}
} else { // Если столбец нечетный, сортируем по возрастанию
// Прямой проход по счетчикам для установки значений в возрастающем порядке
for (int i = 0; i <= 200; ++i) {
while (count[i] > 0) {
// Устанавливаем значение в ячейку и уменьшаем счетчик
sortedMatrix.data[index][j] = i - 100;
++index;
--count[i];
}
}
```

```
}
}

// Возвращаем отсортированную матрицу
return sortedMatrix;
}

};

int main() {
int rows, cols;
std::cout << "Enter the number of rows and columns for the matrix: ";
std::cin >> rows >> cols;

// Создание матрицы с заданными размерами и инициализированной нулями
Matrix originalMatrix(rows, cols, "originalMatrix");

originalMatrix.randomizeMatrix();
originalMatrix.printMatrix();

////////////////////////////////////////////////////////////////////////////////////////////////////
// Создание копии матрицы и сортировка копии с помощью LSD Radix
std::cout << "\nSorted Matrix using LSD Radix Sort:\n";
Matrix copiedMatrix = originalMatrix.copyMatrix("LSD Radix");
auto start1 = std::chrono::high_resolution_clock::now();
Matrix sortedLSDMatrix = copiedMatrix.lsdRadixSort();
auto end1 = std::chrono::high_resolution_clock::now();
auto duration1 = std::chrono::duration_cast<std::chrono::microseconds>(end1 -
start1);
sortedLSDMatrix.printMatrix();



// Создание копии матрицы и сортировка копии с помощью Selection Sort
std::cout << "\nSorted Matrix using Selection Sort:\n";
copiedMatrix = originalMatrix.copyMatrix("Selection Sort");
```

```
auto start2 = std::chrono::high_resolution_clock::now();
Matrix sortedSelectionMatrix = copiedMatrix.selectionSort();
auto end2 = std::chrono::high_resolution_clock::now();
auto duration2 = std::chrono::duration_cast<std::chrono::microseconds>(end2 -
start2);
sortedSelectionMatrix.printMatrix();



// Создание копии матрицы и сортировка копии с помощью Counting Sort
std::cout << "\nSorted Matrix using Counting Sort:\n";
copiedMatrix = originalMatrix.copyMatrix("Counting Sort");
auto start3 = std::chrono::high_resolution_clock::now();
Matrix sortedCountingMatrix = copiedMatrix.countingSort();
auto end3 = std::chrono::high_resolution_clock::now();
auto duration3 = std::chrono::duration_cast<std::chrono::microseconds>(end3 -
start3);
sortedCountingMatrix.printMatrix();



// Создание копии матрицы и сортировка копии с помощью Bubble Sort
std::cout << "\nSorted Matrix using Bubble Sort:\n";
copiedMatrix = originalMatrix.copyMatrix("Bubble Sort");
auto start4 = std::chrono::high_resolution_clock::now();
Matrix sortedBubbleMatrix = copiedMatrix.bubbleSort();
auto end4 = std::chrono::high_resolution_clock::now();
auto duration4 = std::chrono::duration_cast<std::chrono::microseconds>(end4 -
start4);
sortedBubbleMatrix.printMatrix();

////////////////////////////////////////////////////////////////////////////////////////////////////
std::cout << std::endl << std::endl;
std::cout << "Selection sort took -> " << duration1.count() << " microseconds\n";
std::cout << "LSD Radix sort took -> " << duration2.count() << " microseconds\n";
std::cout << "Counting sort took -> " << duration3.count() << " microseconds\n";
```

```
std::cout << "Bubble sort took -> " << duration4.count() << " microseconds\n";


return 0;
}
```

3) Результат работы:

```
Enter the number of rows and columns for the matrix: 10000 5
Matrix originalMatrix:

Printing terminated! The matrix size is too large!

Sorted Matrix using LSD Radix Sort:
Matrix LSD Radix:

Printing terminated! The matrix size is too large!

Sorted Matrix using Selection Sort:
Matrix Selection Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Counting Sort:
Matrix Counting Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Bubble Sort:
Matrix Bubble Sort:

Printing terminated! The matrix size is too large!


Selection sort took ->  58214 microseconds
LSD Radix sort took ->  10424220 microseconds
Counting sort took  ->  8896 microseconds
Bubble sort took    ->  25112338 microseconds
alex@fedora:~/Documents/code/alg$
```

```
Enter the number of rows and columns for the matrix: 5 10000
Matrix originalMatrix:

Printing terminated! The matrix size is too large!

Sorted Matrix using LSD Radix Sort:
Matrix LSD Radix:

Printing terminated! The matrix size is too large!

Sorted Matrix using Selection Sort:
Matrix Selection Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Counting Sort:
Matrix Counting Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Bubble Sort:
Matrix Bubble Sort:

Printing terminated! The matrix size is too large!


Selection sort took ->  169686 microseconds
LSD Radix sort took ->  8954 microseconds
Counting sort took  ->  60264 microseconds
Bubble sort took    ->  10494 microseconds
alex@fedora:~/Documents/code/alg$
```

```
Enter the number of rows and columns for the matrix: 250 250
Matrix originalMatrix:

Printing terminated! The matrix size is too large!

Sorted Matrix using LSD Radix Sort:
Matrix LSD Radix:

Printing terminated! The matrix size is too large!

Sorted Matrix using Selection Sort:
Matrix Selection Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Counting Sort:
Matrix Counting Sort:

Printing terminated! The matrix size is too large!

Sorted Matrix using Bubble Sort:
Matrix Bubble Sort:

Printing terminated! The matrix size is too large!


Selection sort took ->  77740 microseconds
LSD Radix sort took ->  347373 microseconds
Counting sort took  ->  8323 microseconds
Bubble sort took    ->  717634 microseconds
alex@fedora:~/Documents/code/alg$
```

4) проанализируем алгоритмы сортировки, в коде, с точки зрения их временной сложности (Big O) и использования памяти:

1. LSD Radix Sort:
   - Временная сложность: O(d * (n + k)), где d - количество разрядов в числах, n - количество элементов в матрице, k - размер корзины (обычно 10 для десятичной системы).
   - Память: O(n * k), так как требуется дополнительное пространство для хранения корзин.

2. Selection Sort:
   - Временная сложность: O(n^2), где n - количество элементов в матрице.
   - Память: O(1), поскольку сортировка выполняется на месте без дополнительных структур данных.

3. Counting Sort:
   - Временная сложность: O(n + k), где n - количество элементов в матрице, k - разница между максимальным и минимальным элементами.
   - Память: O(k), где k - разница между максимальным и минимальным элементами.

4. Bubble Sort:
   - Временная сложность: O(n^2), где n - количество элементов в матрице.
   - Память: O(1), так как сортировка выполняется на месте без дополнительных структур данных.

Исходя из этого, LSD Radix Sort обладает наилучшей временной сложностью O(d * (n + k)) и требует O(n * k) памяти, где d - количество разрядов в числах, n - количество элементов в матрице, k - размер корзины. Однако, если разряды чисел и/или их количество невелики, Selection Sort или Bubble Sort могут быть более эффективными в некоторых случаях, учитывая их более низкую асимптотическую сложность по памяти.

Вывод: в результате выполнения лабораторной работы были приобретены практические навыки работы в разработки алгоритма сортировки.