Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

На тему: «Внутренние сортировки массивов»

Выполнил студент ИВТ-22-2б:

Мифтахов Марат Ринатович

Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

Пермь 2023

**Постановка задачи**

Задан массив целых чисел. Реализовать сортировку элементов по возрастанию четырьмя методами.

**1. Метод слияния:**

**Алгоритм решения**

1. Массив рекурсивно разбивается на две равные части, и каждая из частей делиться до тех пор, пока размер очередного подмассива не станет равным единице;

2. Выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив;

3. В конце операции слияния элементы перезаписываются из результирующего массива в исходный.

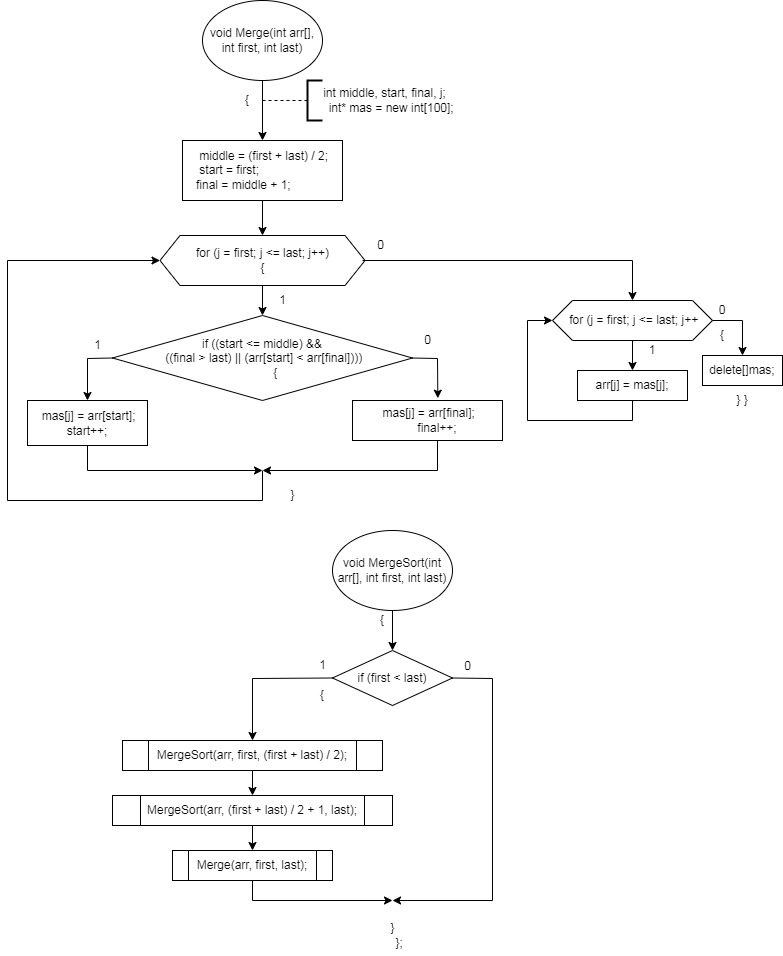
**Блок-схемы**

Рисунок 1 – блок-схемы функций void Merge() и void MergeSort()

**Код программы**

//Сортировка слиянием

void Merge(int arr[], int first, int last)

{

int middle, start, final, j;

int\* mas = new int[100];

middle = (first + last) / 2; // Находим середину массива

start = first; // Устанавливаем начало первой половинки

final = middle + 1; // Устанавливаем начало второй половинки

for (j = first; j <= last; j++) // Слияние половинок

if ((start <= middle) && ((final > last) || (arr[start] < arr[final]))) /\* Если элемент из первой половинки меньше, чем из второй

добавляем его в временный массив\*/

{

mas[j] = arr[start];

start++;

}

else

{

mas[j] = arr[final];// Иначе добавляем элемент из второй половинки

final++;

}

for (j = first; j <= last; j++) arr[j] = mas[j]; // Копируем отсортированный массив из временного массива в исходный

delete[]mas;

}

void MergeSort(int arr[], int first, int last)

{

{

if (first < last)

{

MergeSort(arr, first, (first + last) / 2);

MergeSort(arr, (first + last) / 2 + 1, last);

Merge(arr, first, last);

}

}

}

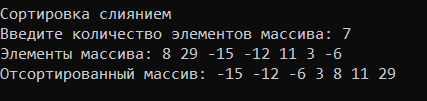
**Вывод программы**

Рисунок 2 – результат работы программы

**2. Метод быстрой сортировки:**

**Алгоритм решения**

1. Отличительной особенностью быстрой сортировки является операция разбиения массива на две части относительно опорного элемента.

2. Если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному.

3. Вне зависимости от того, какой элемент выбран в качестве опорного, массив будет отсортирован, но все же наиболее удачным считается ситуация, когда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов.

4. Если длина какой-то из получившихся в результате разбиения частей превышает один элемент, то для нее нужно рекурсивно выполнить упорядочивание, т. е. повторно запустить алгоритм на каждом из отрезков.

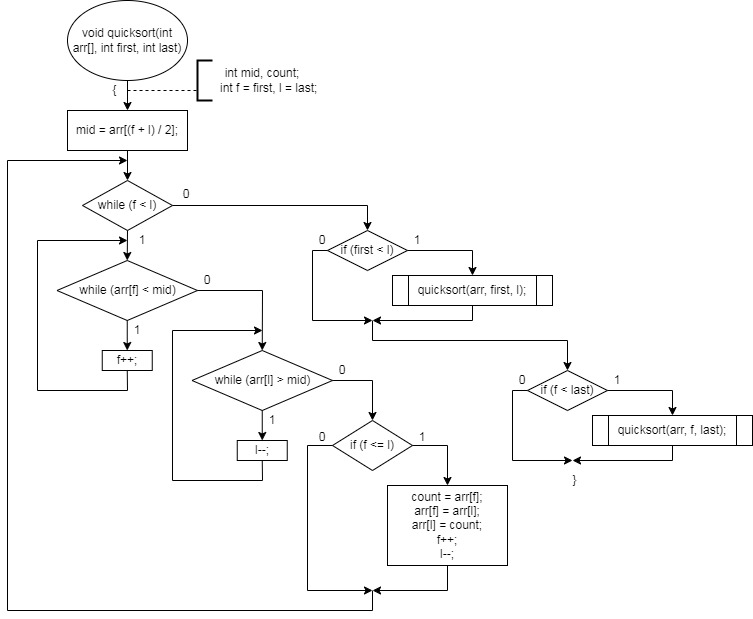
**Блок-схема**

Рисунок 3 – блок-схема функции void quicksort()

**Код программы**

//Быстрая сортировка

void quicksort(int arr[], int first, int last)

{

int mid, count;

int f = first, l = last;

mid = arr[(f + l) / 2]; //Нахождение центра массива

do

{ //Сравнение элемнтов массива с центральным значением

while (arr[f] < mid) f++;

while (arr[l] > mid) l--;

if (f <= l)

{

//Сортировка элементов

count = arr[f];

arr[f] = arr[l];

arr[l] = count;

f++;

l--;

}

} while (f < l); //Выполнение сортировки для частей массива

if (first < l) quicksort(arr, first, l);

if (f < last) quicksort(arr, f, last);

}

**Вывод программы**

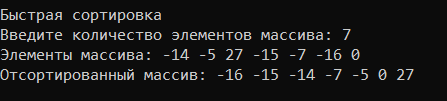


Рисунок 4 – результат работы программы

**3. Метод сортировки подсчётом:**

**Алгоритм решения**

Главная идея алгоритма — посчитать, сколько раз встречается каждый элемент в массиве, а потом заполнить исходный массив результатами этого подсчёта. Для этого нам нужен вспомогательный массив, где мы будем хранить результаты подсчёта. Даже если нам надо отсортировать миллион чисел, мы всё равно знаем диапазон этих чисел заранее, например, от 1 до 100. Это значит, что во вспомогательном массиве будет не миллион элементов, а сто.

В общем виде всё работает так:

Мы создаём вспомогательный массив и на старте заполняем его нулями.

Проходим по всему исходному массиву и смотрим очередное значение в ячейке.

Берём содержимое этой ячейки и увеличиваем на единицу значение вспомогательного массива под этим номером. Например, если мы встретили число 5, то увеличиваем на единицу пятый элемент вспомогательного массива. Если встретили 13 — тринадцатый.

После цикла во вспомогательном массиве у нас хранятся данные, сколько раз встречается каждый элемент.

Теперь мы проходим по вспомогательному массиву, и если в очередной ячейке лежит что-то больше нуля, то мы в исходный массив столько же раз отправляем номер этой ячейки. Например, в первой ячейке вспомогательного массива лежит число 7. Это значит, что в исходный массив мы отправляем единицу 7 раз подряд.

В итоге мы получаем отсортированный массив без сравнения элементов.

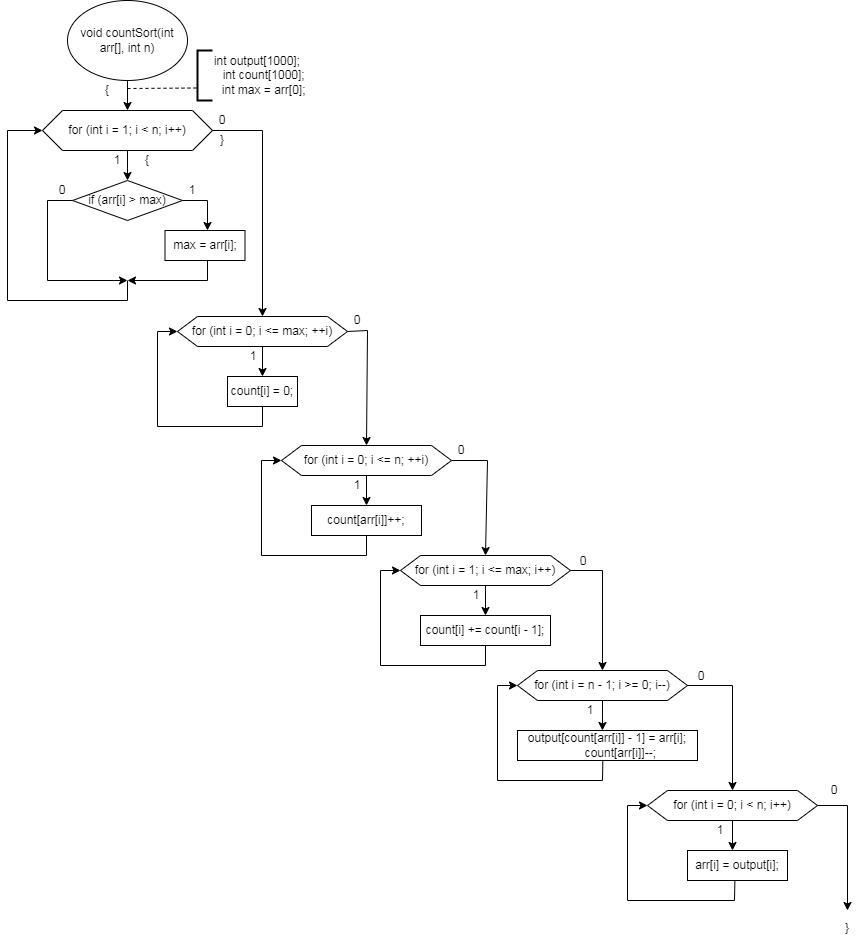
**Блок-схема**

Рисунок 5 – блок-схема функции void countSort()

**Код программы**

//Сортировка подсчетом

void countSort(int arr[], int n) {

// Находим минимальное значение в массиве

int min = arr[0];

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

if (arr[i] < min) {

min = arr[i];

}

}

// "Смещаем" значения входного массива, чтобы все значения были неотрицательными

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

arr[i] -= min;

}

// Сортировка подсчетом на измененном массиве

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

int\* count = new int[max + 1] {0};

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

count[arr[i]]++;

}

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

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

}

int\* output = new int[n];

for (int i = n - 1; i >= 0; i--) {

output[count[arr[i]] - 1] = arr[i];

count[arr[i]]--;

}

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

arr[i] = output[i];

}

// Восстанавливаем значения в отсортированном массиве

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

arr[i] += min;

}

delete[] output;

delete[] count;

}

**Вывод программы**

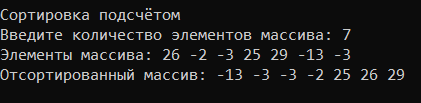
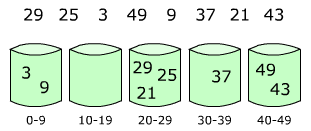


Рисунок 6 – результат работы программы

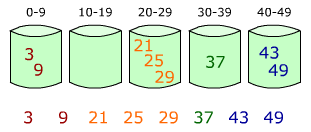
**4. Блочная сортировка:**

**Алгоритм решения**

Блочная сортировка (другие названия: карманная сортировка, корзинная сортировка, англ. Bucket sort) - алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив. Этот тип сортировки может обладать линейным временем исполнения.



Элементы распределяются по корзинам



Затем элементы в каждой корзине сортируются

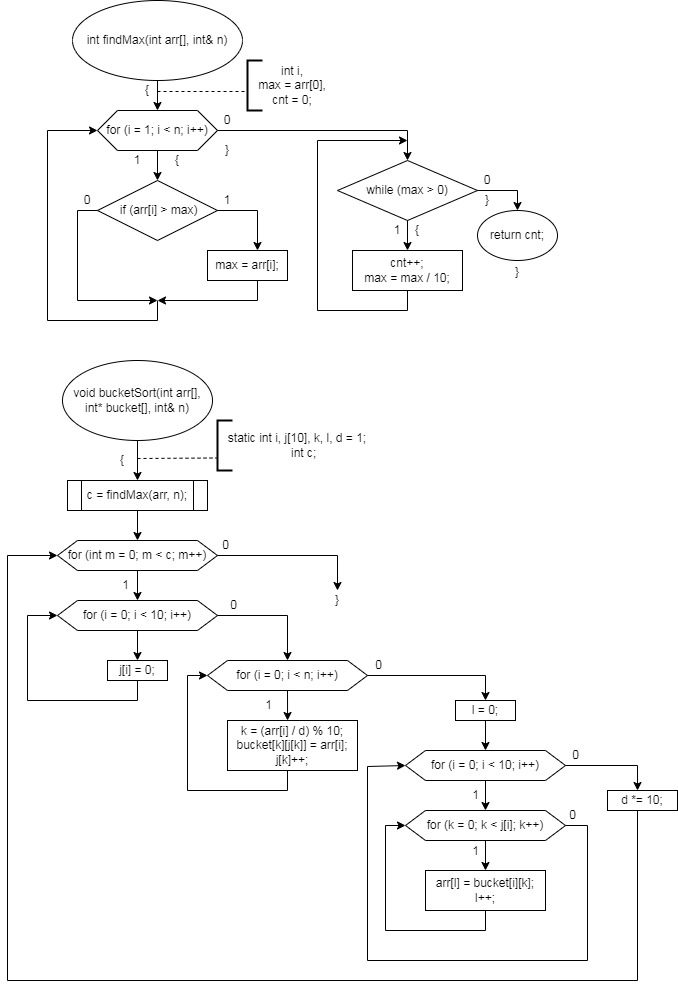
**Блок-схемы**

Рисунок 7 – блок-схемы функций int findMax() и void bucketSort()

**Код программы**

//Блочная сортировка

const int ARRAY\_SIZE = 10;

const int BUCKET\_SIZE = 10;

void bucketSort(int arr[], int size) {

int max\_val = arr[0];

int min\_val = arr[0];

for (int i = 1; i < size; i++) {// Найти максимальный и минимальный элементы в массиве

if (arr[i] > max\_val) {

max\_val = arr[i];

}

if (arr[i] < min\_val) {

min\_val = arr[i];

}

}

float range = (float)(max\_val - min\_val + 1) / BUCKET\_SIZE; // Вычислить размер каждого блока

int buckets[BUCKET\_SIZE][ARRAY\_SIZE] = { 0 }; // Создать массив блоков и массив размеров блоков

int bucket\_sizes[BUCKET\_SIZE] = { 0 };

for (int i = 0; i < size; i++) { // Распределить элементы по блокам

int bucket\_index = (int)floor((arr[i] - min\_val) / range); // // Вычислить индекс блока, к которому относится текущий элемент

if (bucket\_index == BUCKET\_SIZE) {

bucket\_index--;

}

int j = 0; // Найти позицию для вставки элемента в блоке

while (j < bucket\_sizes[bucket\_index] && arr[i] > buckets[bucket\_index][j]) {

j++;

}

for (int k = bucket\_sizes[bucket\_index] - 1; k >= j; k--) { // Сдвинуть элементы в блоке, чтобы освободить место для вставки нового элемента

buckets[bucket\_index][k + 1] = buckets[bucket\_index][k];

}

buckets[bucket\_index][j] = arr[i]; // Вставить элемент в блок и увеличить размер блока

bucket\_sizes[bucket\_index]++;

}

int index = 0; // Собрать отсортированный массив из блоков

for (int i = 0; i < BUCKET\_SIZE; i++) {

for (int j = 0; j < bucket\_sizes[i]; j++) {

arr[index++] = buckets[i][j];

}

}

}

**Вывод программы**

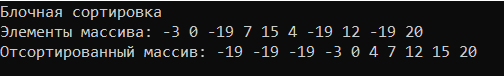


Рисунок 8 – результат работы программы