

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**

### **«Поразрядная сортировка для целых чисел с четно- нечетным слиянием Бэтчера»**

**Выполнил:**

студент группы 381706-2  
Зинков А. С.

**Проверил:**

Доцент кафедры МОСТ,  
кандидат технических наук,  
Сысоев А.В.

## Содержание

Введение.....	3
Постановка задачи .....	4
Метод решения.....	5
Схема распараллеливания .....	6
Описание программной реализации.....	7
Подтверждение корректности .....	8
Результаты экспериментов .....	9
Заключение.....	10
Литература .....	11
Приложение.....	12

## **Введение**

Сортировка данных имеет очевидное практическое применение во многих областях (решение систем линейных уравнений, упорядочивание графов, базы данных и др.). Проблема, однако, в том, что таким, скорее всего весьма простым алгоритмом, не удастся воспользоваться при решении прикладных задач большой размерности, где объемы упорядочиваемых данных исчисляются десятками миллионов элементов. В современном мире число таких задач и их важность неуклонно растут. Для достижения большей эффективности необходимо выполнить вычисления параллельно.

Целью данной работы является реализация параллельного алгоритма поразрядной сортировки для целых чисел со слиянием Бэтчера.

## **Постановка задачи**

В рамках данной работы были поставлены следующие задачи:

1. Реализация последовательного алгоритма поразрядной сортировки для целых чисел.
2. Реализация параллельного алгоритма поразрядной сортировки для целых чисел со слиянием Бэтчера.
3. Проведение вычислительных экспериментов.
4. Сравнение эффективности алгоритмов.

## Метод решения

Поразрядная сортировка (англ. *radix sort*) — алгоритм сортировки, который выполняется за линейное время.

Массив несколько раз перебирается, и элементы перегруппировываются в зависимости от того, какая цифра находится в определённом разряде. После обработки разрядов (всех или почти всех) массив оказывается упорядоченным. При этом разряды могут обрабатываться в противоположных направлениях - от младших к старшим или наоборот.

Будем использовать побайтовую реализацию поразрядной сортировки как наиболее эффективную.

Для  $i$  прохода по массиву необходимо выполнить следующие операции:

1. Проход по исходному массиву и подсчет количества  $i$ -ых байт, сохранение результата в массив.
2. Проход по массиву подсчетов и вычисление смещений.
3. Проход по исходному массиву и копирование элементов в результирующий массив соответствующий их смещению.

Чётно-нечётное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечётной и чётной позициях. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Чётно-нечётное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние  $n$  массивов могут выполнять  $n$  параллельных потоков. На следующем шаге слияние  $n/2$  полученных массивов будут выполнять  $n/2$  потоков и т.д. На последнем шаге два массива будут сливать 2 потока.

## Схема распараллеливания

Сначала мы разделяем исходный массив на количество частей равному количеству процессов. Далее передаем каждому процессу свою часть элементов из исходного массива и сортируем их с помощью поразрядной сортировки.

Следом нужно слить все эти массивы с помощью слияния Бэтчера. Для этого необходимо произвести  $\lceil \log_2 k \rceil$  цикловслияний, где  $k$ - количество процессов.

На  $i$ -ой итерации будут попарно сливаться процессы, для которых выполняется  $j \% 2^i = 0$  с процессами, для которых  $j + 2^{i-1}$ , где  $j$ - ранг процесса. Для каждого слияния необходимо выполнить следующие действия:

1. Разделить массивы в первом и втором сливаемых процессах на четные и нечетные. Второй процесс передает первому четные элементы, а первый второму нечетные.
2. Первый процесс сливает в один массив четные элементы, а второй нечетные.
3. Второй процесс отправляет первому свой отсортированный массив и первый эти массивы объединяет.
4. Первый процесс проходит по массиву и сравнивает четные и нечетные элементы.

На последней итерации слияния необходимо вместо шага 4 нужно пересортировать массив так чтобы четные и нечетные элементы были на своих местах, а затем так же сделать проход по массиву и сравнить нечетные элементы с четными.

После всех итераций отсортированный массив будет находится в 0 процессе.

## Описание программной реализации

Программа содержит 8 функций:

`std::vector<int> getRandomVector(int size)` – создает рандомный вектор.

`std::vector<int> merge_batcher(std::vector<int> global_vec, int size_vec)` – параллельный алгоритм поразрядной сортировки со слиянием Бэтчера.

`std::vector<int> merge_even(const std::vector<int>& vec1, const std::vector<int>& vec2)` – сливает четные элементы.

`std::vector<int> merge_odd(const std::vector<int>& vec1, const std::vector<int>& vec2)` – сливает нечетные элементы.

`std::vector<int> transpos(std::vector<int> vec, int even_size, int odd_size)` – проход по массиву и сравнение нечетных и четных элементов.

`std::vector<int> merge(std::vector<int> vec, int even_size, int odd_size)` – пересортирует массив так чтобы четные и нечетные элементы были на своих местах, а затем так же делает проход по массиву и сравнить нечетные элементы с четными.

`std::vector<int> shuffle(std::vector<int> vec)` – разделяет массив на четные и нечетные элементы.

`std::vector<int> radix_sort_bit(std::vector<int> vec)` – поразрядная сортировка.

## Подтверждение корректности

Для подтверждения корректности в программе реализован набор тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework:

`Test_Disordered_Vector` – проверка корректности работы алгоритма для полностью неупорядоченного вектора.

`Test_Odd_Size_Vector` – проверка корректности работы алгоритма для вектора нечетной длины.

`Test_Identical_Elements_Vector` – проверка корректности работы алгоритма для вектора состоящего из одинаковых элементов.

`Test_Ordered_Vector` – проверка корректности работы алгоритма заранее упорядоченного вектора.

`Test_Big_Size_Vector` – проверка корректности работы алгоритма для вектора большого размера.

`Test_Const_Vector` – проверка корректности работы алгоритма для заранее заданного вектора.

`Test_One_Elements_Vector` – проверка корректности работы алгоритма для вектора состоящего из одного элемента.

`Test_Merge` – проверка корректности работы самого алгоритма.



## Результаты экспериментов

Эксперименты проводились на ПК со следующими параметрами:

1. Операционная система: Windows 10 Pro.
2. Процессор: AMD Ryzen 5 2600 Six-Core Processor 3.90 GHz.
3. ОЗУ 16 гб.
4. Версия Visual Studio: 2019.

Эксперимент проводился на 100 000 000 элементов. Количество процессов равно 1 - 8.

Количество процессов	Время последовательного алгоритма (сек.)	Время параллельного алгоритма (сек.)	Ускорение
1	4.5936	4.6835	0.9808
2	4.5876	3.7837	1,2124
3	4.6161	3.4735	1,3289
4	4.6130	2.8788	1,6024
5	4.6135	3.1213	1,4780
6	4.5999	2.7864	1,6508
7	4.6090	2.6545	1,7362
8	4.6463	2.4942	1,8628
9	4.6217	3.2214	1,4346
10	4.6103	2.8815	1,5999

Таблица 1. Сравнение времени работы программы при параллельном и последовательном алгоритме.

По данным экспериментам видно, что при увеличении количества процессов увеличивается ускорение, следовательно, алгоритм работает корректно. Также можно заметить, что наибольшая эффективность достигается при количестве процессов близкому к количеству ядер и равному степени двойки, так как количество циклов слияний в этом случае наиболее оптимально. Для количества процессов больше восьми эффективность уменьшается, так как возрастают накладные расходы.

Можем сделать вывод, что использование поразрядной сортировки с четно-нечетным слиянием Бэтчера наиболее эффективно, чем последовательная поразрядная сортировка.

## **Заключение**

В ходе работы была написана два алгоритма сортировки массивов: поразрядная сортировки и алгоритм поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера, использующий технологию MPI.

Корректность работы подтверждается с помощью библиотеки модульного тестирования Google C++ Testing Framework.

По данным вычислительных экспериментов можно сделать вывод, на достаточно большом количестве элементов параллельный алгоритм наиболее эффективен, по сравнению с последовательным.

## Литература

### Книги:

- Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие – Нижний

### Internet-ресурсы:

- Национальный Открытый Университет «ИНТУИТ». Академия: ИнтернетУниверситет Суперкомпьютерных Технологий. Курс: Теория и практика параллельных вычислений. Автор: Виктор Гергель. ISBN: 978-5-9556-0096-3. URL: <https://www.intuit.ru/studies/courses/1156/190/info>
- Четно-нечетная сортировка слиянием Бэтчера: Хабр, сообщество IT-специалистов. - <https://habr.com/ru/post/261777/>

# Приложение

## main.cpp

```
// Copyright 2019 Zinkov Artem
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include <numeric>
#include <algorithm>
#include "../radix_sort_merge_batcher.h"

TEST(Radix_Sort_Merge_Batcher, Test_Merge) {
    int rank;
    double t1, t2;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec;
    const int size_vector = 100000000;
    if (rank == 0) {
        global_vec = getRandomVector(size_vector);
        t1 = MPI_Wtime();
    }

    std::vector<int> parralel = merge_batcher(global_vec, size_vector);
    if (rank == 0) {
        t1 = MPI_Wtime() - t1;
        std::cout << "time parral = " << t1 << std::endl;
        t2 = MPI_Wtime();
        // std::sort(global_vec.begin(), global_vec.end());
        global_vec = radix_sort_bit(global_vec);
        t2 = MPI_Wtime() - t2;
        std::cout << "time seq = " << t2 << std::endl;
        ASSERT_EQ(parralel, global_vec);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_Disordered_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec(50), res(50);
    if (rank == 0) {
        for (size_t i = 0; i < global_vec.size(); i++) {
            global_vec[i] = global_vec.size() - i;
        }
        std::iota(res.begin(), res.end(), 1);
    }

    std::vector<int> parralel = merge_batcher(global_vec, global_vec.size());
    if (rank == 0) {
        ASSERT_EQ(parralel, res);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_Odd_Size_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec;
    const int size_vector = 225;
    if (rank == 0) {
        global_vec = getRandomVector(size_vector);
    }
}
```

```

std::vector<int> parralel = merge_batcher(global_vec, size_vector);
if (rank == 0) {
    global_vec = radix_sort_bit(global_vec);
    ASSERT_EQ(parralel, global_vec);
}
}

TEST(Radix_Sort_Merge_Batcher, Test_Identical_Elements_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec(200, 5);

    std::vector<int> parralel = merge_batcher(global_vec, global_vec.size());
    if (rank == 0) {
        global_vec = radix_sort_bit(global_vec);
        ASSERT_EQ(parralel, global_vec);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_Ordered_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec(200);
    if (rank == 0) {
        std::iota(global_vec.begin(), global_vec.end(), 1);
    }

    std::vector<int> parralel = merge_batcher(global_vec, global_vec.size());
    if (rank == 0) {
        ASSERT_EQ(parralel, global_vec);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_Big_Size_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec;
    const int size_vector = 10000;
    if (rank == 0) {
        global_vec = getRandomVector(size_vector);
    }

    std::vector<int> parralel = merge_batcher(global_vec, size_vector);
    if (rank == 0) {
        global_vec = radix_sort_bit(global_vec);
        ASSERT_EQ(parralel, global_vec);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_Const_Vect) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> global_vec = {57, 39, 26, 163, 4, 273, 14, 2, 356, 37, 93, 3, 678, 256, 83, 17, 26};
    std::vector<int> res = {2, 3, 4, 14, 17, 26, 26, 37, 39, 57, 83, 93, 163, 256, 273, 356, 678};

    std::vector<int> parralel = merge_batcher(global_vec, global_vec.size());
    if (rank == 0) {
        ASSERT_EQ(parralel, res);
    }
}

TEST(Radix_Sort_Merge_Batcher, Test_One_Elements_Vector) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

std::vector<int> global_vec;
const int size_vector = 1;
if (rank == 0) {
    global_vec = getRandomVector(size_vector);
}

std::vector<int> parralel = merge_batcher(global_vec, size_vector);
if (rank == 0) {
    global_vec = radix_sort_bit(global_vec);
    ASSERT_EQ(parralel, global_vec);
}
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}

```

## radix\_sort\_merge\_batcher.h

```

// Copyright 2019 Zinkov Artem
#ifdef
MODULES_TASK_3_ZINKOV_RADIX_SORT_MERGE_BATCHER_RADIX_SORT_MERGE_BATCHER_H_
#define
MODULES_TASK_3_ZINKOV_RADIX_SORT_MERGE_BATCHER_RADIX_SORT_MERGE_BATCHER_H_

#include <mpi.h>
#include <vector>

std::vector<int> getRandomVector(int size);
std::vector<int> merge_batcher(std::vector<int> global_vec, int size_vec);
std::vector<int> merge_even(const std::vector<int>& vec1, const std::vector<int>& vec2);
std::vector<int> merge_odd(const std::vector<int>& vec1, const std::vector<int>& vec2);
std::vector<int> transpos(std::vector<int> vec, int even_size, int odd_size);
std::vector<int> merge(std::vector<int> vec, int even_size, int odd_size);
std::vector<int> shuffle(std::vector<int> vec);
std::vector<int> radix_sort_bit(std::vector<int> vec);

#endif //
MODULES_TASK_3_ZINKOV_RADIX_SORT_MERGE_BATCHER_RADIX_SORT_MERGE_BATCHER_H_

```

## radix\_sort\_merge\_batcher.cpp

```

// Copyright 2019 Zinkov Artem
#include <mpi.h>
#include <vector>
#include <random>
#include <ctime>
#include <algorithm>
#include <iostream>
#include <utility>
#include "../modules/task_3/zinkov_radix_sort_merge_batcher/radix_sort_merge_batcher.h"

```

```

std::vector<int> getRandomVector(int length) {
    if (length < 1)
        throw "WRONG_LEN";

    std::vector<int> vec(length);
    std::mt19937 gen;
    gen.seed(static_cast<unsigned int>(time(0)));

    for (auto& val : vec) {
        val = gen() % 1000000;
    }

    return vec;
}

std::vector<int> merge_batcher(std::vector<int> global_vec, int size_vec) {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int delta = size_vec / size;
    const int residue = size_vec % size;
    std::vector<int> local_vec;
    if (size_vec < size || size == 1) {
        if (rank == 0)
            global_vec = radix_sort_bit(global_vec);
        return global_vec;
    }

    if (rank == 0) {
        local_vec.reserve(size_vec);
        local_vec.resize(delta + residue);
    } else {
        local_vec.resize(delta);
    }

    int* sendcounts = new int[size];
    int* displs = new int[size];

    for (int i = 0; i < size; i++) {
        displs[i] = 0;
        if (i == 0) {
            sendcounts[i] = delta + residue;
        } else {
            sendcounts[i] = delta;
        }
        if (i > 0) {
            displs[i] = displs[i - 1] + sendcounts[i - 1];
        }
    }

    MPI_Scatterv(global_vec.data(), sendcounts, displs, MPI_INT,
        &local_vec.front(), sendcounts[rank], MPI_INT, 0, MPI_COMM_WORLD);

    int num_merge = 1;
    while (pow(2, num_merge) < size)
        num_merge++;
    //std::sort(local_vec.begin(), local_vec.end());
    local_vec = radix_sort_bit(local_vec);
    local_vec = shuffle(local_vec);

    int merged_proc = 2, displs_proc = 1, length_send, length_recv;
    std::vector<int> res, even, odd;

```

```

int k = 2;
if (rank != 0) {
    for (int i = 2; i < num_merge - 1; i++) {
        k = pow(2, i);
        if (rank % k == 0) {
            local_vec.reserve(size_vec / size * k + 10);
            even.reserve(size_vec / size / 2 * k + 10);
            odd.reserve(size_vec / size / 2 * k + 10);
            res.reserve(size_vec / size / 4 * k + 10);
        }
    }
}

if (rank == 0) {
    even.reserve(size_vec / 2 + 10);
    odd.reserve(size_vec / 2 + 10);
    res.reserve(size_vec / 4 + 10);
}

MPI_Status status;

for (int i = 0; i < num_merge; i++) {
    if (rank % merged_proc == 0 && rank + displs_proc < size) {
        length_send = local_vec.size() / 2;

        MPI_Sendrecv(&length_send, 1, MPI_INT, rank + displs_proc, 0,
                     &length_recv, 1, MPI_INT, rank + displs_proc, 0, MPI_COMM_WORLD, &status);
        res.resize(length_recv / 2 + length_recv % 2);
        MPI_Sendrecv(&local_vec[local_vec.size() / 2 + local_vec.size() % 2], length_send,
                     MPI_INT, rank + displs_proc, 0, &res.front(), length_recv / 2 + length_recv % 2,
                     MPI_INT, rank + displs_proc, 0, MPI_COMM_WORLD, &status);

        even = merge_even(local_vec, res);

        odd.resize(length_recv / 2 + local_vec.size() / 2);
        MPI_Recv(&odd.front(), length_recv / 2 + local_vec.size() / 2, MPI_INT,
                 rank + displs_proc, 0, MPI_COMM_WORLD, &status);

        local_vec.resize(even.size() + odd.size());
        std::copy(even.begin(), even.end(), local_vec.begin());
        std::copy(odd.begin(), odd.end(), local_vec.begin() + even.size());

        if (i + 1 != num_merge)
            local_vec = transpos(local_vec, even.size(), odd.size());
        else
            local_vec = merge(local_vec, even.size(), odd.size());
    }
    if (rank - displs_proc >= 0 && (rank - displs_proc) % merged_proc == 0) {
        length_send = local_vec.size();
        MPI_Sendrecv(&length_send, 1, MPI_INT, rank - displs_proc, 0, &length_recv, 1,
                     MPI_INT, rank - displs_proc, 0, MPI_COMM_WORLD, &status);
        res.resize(length_recv);
        MPI_Sendrecv(local_vec.data(), length_send / 2 + length_send % 2, MPI_INT,
                     rank - displs_proc, 0, &res.front(), length_recv, MPI_INT,
                     rank - displs_proc, 0, MPI_COMM_WORLD, &status);
        odd = merge_odd(local_vec, res);

        MPI_Send(odd.data(), odd.size(), MPI_INT, rank - displs_proc, 0, MPI_COMM_WORLD);
    }
    merged_proc *= 2;
    displs_proc *= 2;
}
return local_vec;
}

```



```

std::vector<int> shuffle(std::vector<int> vec) {
    std::vector<int> tmp(vec.size());

    for (size_t i = 0; i < vec.size() / 2 + vec.size() % 2; i++) {
        tmp[i] = vec[2 * i];
    }
    for (size_t i = 1; i < vec.size(); i += 2) {
        tmp[vec.size() / 2 + vec.size() % 2 + i / 2] = vec[i];
    }
    for (size_t i = 0; i < vec.size(); i++) {
        vec[i] = tmp[i];
    }
    return vec;
}

std::vector<int> merge_even(const std::vector<int>& vec1, const std::vector<int>& vec2) {
    std::vector<int> res(vec1.size() / 2 + vec1.size() % 2 + vec2.size());
    size_t j = 0, k = 0;
    int l = 0;

    while (j < (vec1.size() / 2 + vec1.size() % 2) && k < (vec2.size())) {
        if (vec1[j] < vec2[k])
            res[l++] = vec1[j++];
        else
            res[l++] = vec2[k++];
    }

    while (j < vec1.size() / 2 + vec1.size() % 2)
        res[l++] = vec1[j++];
    while (k < vec2.size())
        res[l++] = vec2[k++];
    return res;
}

std::vector<int> merge_odd(const std::vector<int>& vec1, const std::vector<int>& vec2) {
    std::vector<int> res(vec1.size() / 2 + vec2.size());
    size_t j = vec1.size() / 2 + vec1.size() % 2, k = 0;
    int l = 0;

    while (j < vec1.size() && k < vec2.size()) {
        if (vec1[j] < vec2[k])
            res[l++] = vec1[j++];
        else
            res[l++] = vec2[k++];
    }

    while (j < vec1.size())
        res[l++] = vec1[j++];
    while (k < vec2.size())
        res[l++] = vec2[k++];
    return res;
}

std::vector<int> transpos(std::vector<int> vec, int even_size, int odd_size) {
    if (even_size - odd_size == 2) {
        std::vector<int> res(vec.size());
        int j = 0, k = 0, l = 0;

        while (j < even_size && k < odd_size) {
            res[l++] = vec[j++];
            res[l++] = vec[even_size + k];
            k++;
        }
    }
}

```

```

    }

    while (j < even_size)
        res[l++] = vec[j++];

    for (size_t i = 1; i < res.size() - 1; i += 2) {
        if (res[i] > res[i + 1])
            std::swap(res[i], res[i + 1]);
    }

    res = shuffle(res);
    return res;

} else {
    for (int i = 0; i < even_size - 1; i++)
        if (vec[1 + i] < vec[even_size + i])
            std::swap(vec[1 + i], vec[even_size + i]);
    return vec;
}
}

std::vector<int> merge(std::vector<int> vec, int even_size, int odd_size) {
    std::vector<int> res(vec.size());
    int j = 0, k = 0, l = 0;

    while (j < even_size && k < odd_size) {
        res[l++] = vec[j++];
        res[l++] = vec[even_size + k];
        k++;
    }

    while (j < even_size)
        res[l++] = vec[j++];

    for (size_t i = 1; i < res.size() - 1; i += 2) {
        if (res[i] > res[i + 1])
            std::swap(res[i], res[i + 1]);
    }

    return res;
}

std::vector<int> radix_sort_bit(std::vector<int> vec) {
    std::vector<int> res(vec.size());
    std::vector<int> count(256, 0);
    int val;

    for (int i = 0; i < 4; i++) {
        for (size_t j = 0; j < vec.size(); j++) {
            val = (vec[j] >> (i * 8)) & 255;
            count[val]++;
        }
        if (i != 3) {
            for (int j = 1; j < 256; j++)
                count[j] += count[j - 1];
        } else {
            for (int j = 128; j < 256; j++)
                count[j] += count[j - 1];
            count[0] += count[255];
            for (int j = 1; j < 129; j++)
                count[j] += count[j - 1];
        }

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

```

```
    val = (vec[j] >> (i * 8)) & 255;
    res[--count[val]] = vec[j];
}
for (size_t j = 0; j < vec.size(); j++)
    vec[j] = res[j];
for (auto& c : count)
    c = 0;
}
return vec;
}
```