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.

Нижний Новгород

2019

**Содержание**

[Введение 3](#_Toc29654316)

[Постановка задачи 4](#_Toc29654317)

[Метод решения 5](#_Toc29654318)

[Схема распараллеливания 6](#_Toc29654319)

[Описание программной реализации 7](#_Toc29654320)

[Подтверждение корректности 8](#_Toc29654321)

[Результаты экспериментов 9](#_Toc29654322)

[Заключение 10](#_Toc29654323)

[Литература 11](#_Toc29654324)

[Приложение 12](#_Toc29654325)

# 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.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *radix sort*) — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), 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-ыx 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 слияний, где k- количество процессов.

На i-ой итерации будут попарно сливаться процессы, для которых выполняется с процессами, для которых , 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 | Время параллельного алгоритма | 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 написано 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 последовательном.

# 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

#ifndef 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;

}