ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировок»

Выполнил работу:

Зырянов Виталий

Академическая группа:

J3111

Принято:

Ментор, Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

Цель работы:

Реализовать 3 алгоритма сортировка. Проанализировать затрячиваимую память, асимптотику и найти худущие, средние и лучшие случаи для сортировок.

Задачи:

* Реализовать сортировку ставками.
* Реализовать сортировку слиянием.
* Реализовать подрязрядную сортировку.
* Провести тесты над алгоритмами.
* Определить затрачиваемую память.
* Проанализировать алгоритмы посредством графика зависимости скорости работы от входных данных. Сравнить асимптотическую сложность в затрачиваемым временем.

1. Теоретическая подготовка

Используемые типы данных:

* int - целое число, используемый в работе для индексов элементов массивов и в работе самих элементов.
* vector - контейнер, который хранит элементы массива и подмассива.

Используемые функции:

* random\_device -  это компонент стандартной библиотеки C++, который используется для получения случайных чисел.
* chrono::high\_resolution\_clock - функция библиотеки chrono, используемая для получения текущего времени.
* max\_element – функция библиотеки algorithm, позволяет узнать маскимальный элемент в массиве.
* pow – функция стандартной библиотеке iostream, возводит число в заданную степень.

1. Реализация

Первая сортировка должна иметь сложность не меньше квадрата по времени. Я выбрал сортировку вставками. Она работает следующим алгоритмом: сначала начинаем с второго элемента и сравниваем его с предыдущими. Если текущий элемент меньше, перемещаем большие элементы вправо и вставляем текущий элемент на нужную позицию. Повторяем процесс для всех элементов массива.

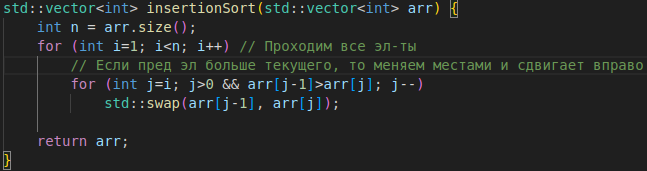


Рисунок. 1 – Сортировка вставками

Вторая сортировка должна работать за O(n logn). В качестве сортировки подобной сложности я выбрал сортировку слиянием. Сначала алгоритм делить исходный массив на две половины, и этот процесс повторяется рекурсивно, пока не останутся подмассивы размером один элемент. Отсортированные подмассивы объединяются в один отсортированный массив.

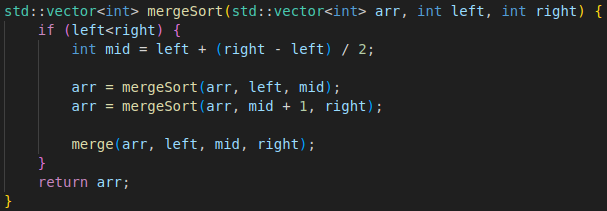


Рисунок 2 –Сортировка слиянием

Третья сортировка должна работать за линейную сложность O(n+k) и по памяти O(n+k). Для этого идеально подходит подразрядная сортировка. Алгоритм работает следующим образом: находим максимальный элемент, чтобы определить количество разрядов, по которым будем сортировать, начинаем сортировку с младшего разряда и используем сортировку для упорядочивания элементов по каждому разряду. Процесс повторяется для всех разрядов, пока не будет отсортирован весь массив.

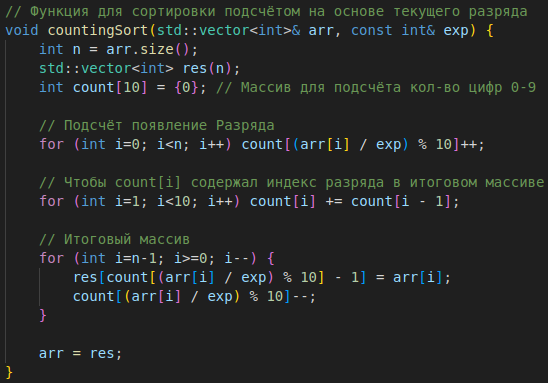


Рисунок 3 –Сортировка на основе разряда

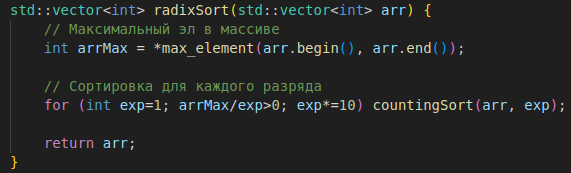


Рисунок 4 – Подразрядная сортировка.

1. Экспериментальная часть

На данном этапе работы проводилась оценка затрачиваемой памяти, асимптотики и времени программы. Замер времени производился с помощью библиотеки chrono. На основе времени был построен график зависимости размера входного массива и времени выполнения.

Подсчёт памяти:

1. Insertation sort:

Вес(vector<int> arr) = n \* size\_of(int) = 4n байт

1. Merge sort:

Вес(vector<int> arr) = n \* size\_of(int) = 4n байт

Вес(vector<int> merge\_arr) = n \* size\_of(int) = 4n байт

1. Radix sort:

Вес(vector<int> arr) = n \* size\_of(int) = 4n байт

Вес(vector<int> count) = r \* size\_of(int) = 4r байт

Подсчёт асимптотики:

1. Insertation sort:

O(n^2) – два вложенных цикла (Рисунок 1)

1. Merge sort:

O(n logn) – цикл и рекурсивный алгоритм (Рисунок 2)

1. Radix sort:

O(n\*r), где r – количество разрядов, – два вложенных цикла (Рисунок 3, 4)

Для тестирования алгоритма была собрана статистика, на основе которой построили графики зависимости входных данных и времени выполнения.

Таблица 1 - Подсчёт сложности для сортировки вставками.

|  |  |  |
| --- | --- | --- |
| Размер входного массива, n: | Время выполнения, с: | O(n^2), c: |
| 1000 | 0.00440735 | 0.003969 |
| 5000 | 0.0846491 | 0.099225 |
| 10000 | 0.37245 | 0.3969 |
| 25000 | 2.40204 | 2.480625 |
| 50000 | 9.36198 | 9.9225 |
| 100000 | 40.0208 | 39.69 |

Рисунок. 5 - График работы алгоритма для сортировки ставками.

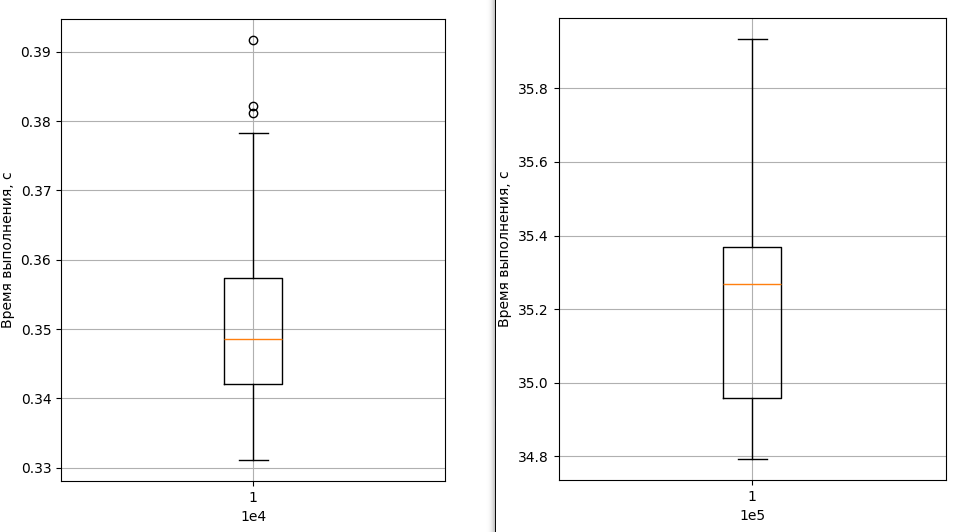
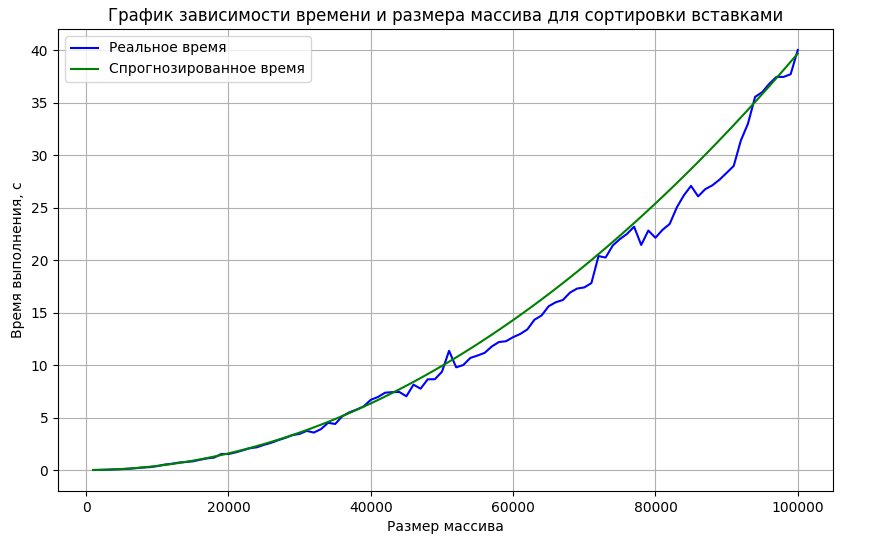


Рисунок 6 - Стабильность сортировки ставками

Таблица 2 - Подсчёт сложности для сортировки слиянием.

|  |  |  |
| --- | --- | --- |
| Размер входного массива, n: | Время выполнения, с: | O(n logn), c: |
| 1000 | 0.00109981 | 0.089692 |
| 5000 | 0.013106 | 0.6143856 |
| 10000 | 0.087764 | 1.328771 |
| 25000 | 0.984029 | 3.65241 |
| 50000 | 3.91058 | 7.80482 |
| 100000 | 17.6026 | 16.60964 |

Рисунок 7 - График работы алгоритма для сортировки слиянием.

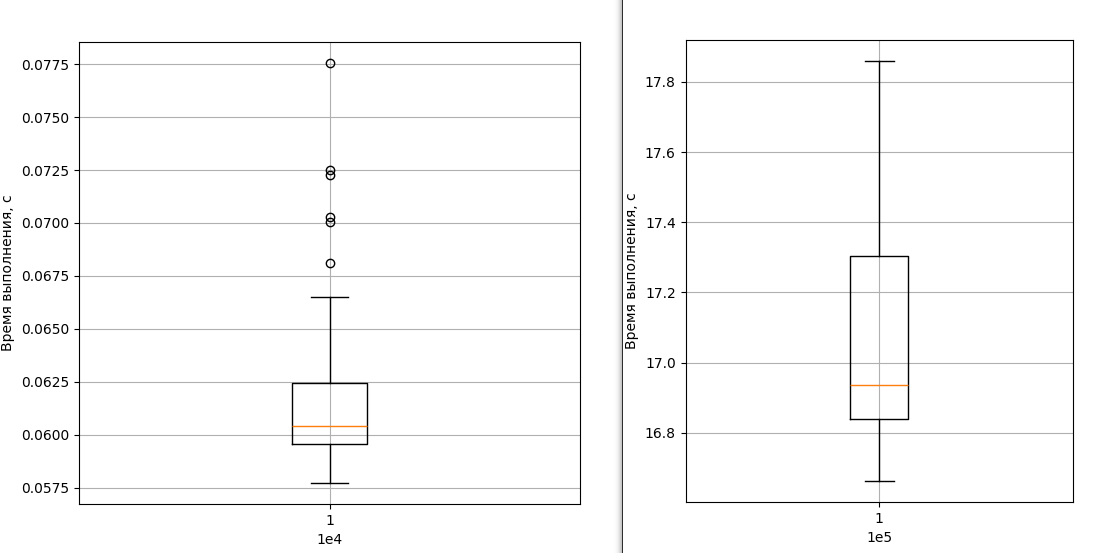
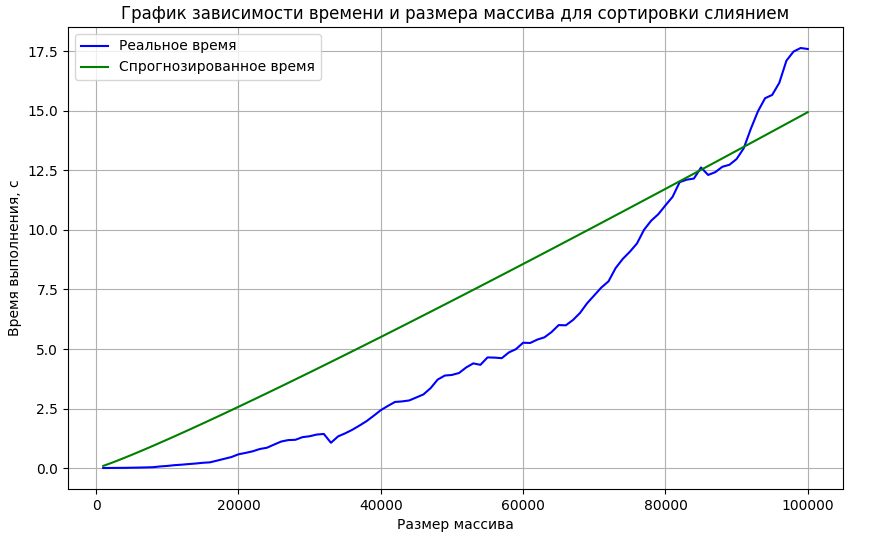


Рисунок 8 - Стабильность сортировки слиянием.

Таблица 3 - Подсчёт сложности для подразрядной сортировки.

|  |  |  |
| --- | --- | --- |
| Размер входного массива, N: | Время выполнения, с: | O(n+r), c: |
| 1000 | 0.000250547 | 0.00018 |
| 5000 | 0.00104916 | 0.0009 |
| 10000 | 0.00154352 | 0.0018 |
| 25000 | 0.00470855 | 0.0045 |
| 50000 | 0.00833091 | 0.009 |
| 100000 | 0.0188761 | 0.018 |

Рисунок 9 - График работы алгоритма для подразрядной сортировки.

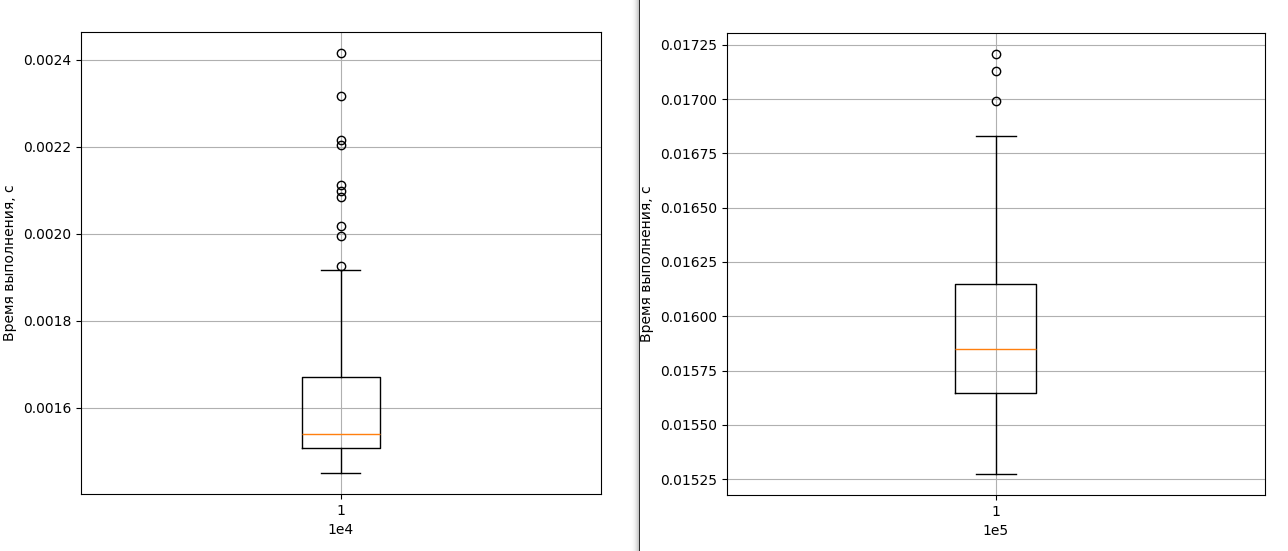


Рисунок 10 - Стабильность подразрядной сортировки.

Анализируя графики можно увидеть, что сложность сортировок была определена корректно.

1. Заключение

В ходе выполнения лабораторной работы мною был реализован алгоритм три алгоритма сортировок разной асимптотической сложности. Проведённое тестирование алгоритмов подтвердило теоретические оценки. Самой эффективной по времени оказалась radix sort, самой худшей стала insertion sort. Самый оптимальный алгоритм по времени и памяти стал merge sort.

ПРИЛОЖЕНИЕ А

Листинг кода файла InsertionSort.cpp

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <cassert>

// Сортировка вставками

// Использование памяти: O(1) для переменных

// Сложность алгоритма: O(n^2) (т. к два вложеных массива длины n)

std::vector<int> insertionSort(std::vector<int> arr) {

int n = arr.size();

for (int i=1; i<n; i++) // Проходим все эл-ты

// Если пред эл больше текущего, то меняем местами и сдвигает вправо

for (int j=i; j>0 && arr[j-1]>arr[j]; j--)

std::swap(arr[j-1], arr[j]);

return arr;

}

// Функция для создания случайного массива

std::vector<int> arrRand(const int& n, const int& arrMin, const int& arrMax) {

std::vector<int> arr(n);

std::random\_device rd;

std::mt19937 seed(rd());

arr[0] = arrMax;

arr[n - 1] = arrMin;

for (int i=1; i<n-1; ++i) arr[i] = arrMin + seed() % (arrMax - arrMin);

return arr;

}

std::vector<int> arrWorse(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = n - i;

return arr;

}

std::vector<int> arrBett(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = i;

return arr;

}

// Вывод

void display(const std::vector<int>& arr, const std::vector<int>& arrSort, std::chrono::duration<double> duration) {

/\*

int n = arr.size();

std::cout << "\nРазмер массива:" << n;

\*/

std::cout << duration.count() << ", ";

/\*

std::cout << "\nИсходный массив:\n";

for (int i = 0; i < n; i++) std::cout << arr[i] << " ";

std::cout << "\nОтсортированный массив:\n";

for (int i = 0; i < n; i++) std::cout << arrSort[i] << " ";

\*/

}

// Проверка на корректность сортировки

bool check(const std::vector<int>& arr) {

int n = arr.size();

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

if (arr[i - 1]>arr[i]) return false;

}

return true;

}

// Тест для алгоритма сортировки

void test(const int& n = 10, const int& arrMax = 100, const int& arrMin = 0) {

std::vector<int> arr = arrRand(n, arrMin, arrMax);

auto start = std::chrono::high\_resolution\_clock::now();

std::vector<int> arrSort = insertionSort(arr);

assert(check(arrSort));

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

display(arr, arrSort, duration);

}

int main() {

test(1000, 999999, 0);

return 0;

}

Листинг кода файла MergeSort.cpp

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <cassert>

// Слияние двух массивов

void merge(std::vector<int>& arr, const int& left, const int& mid, const int& right) {

// Длина левого и правого массивов

int nLeft = mid - left + 1;

int nRight = right - mid;

std::vector<int> arrLeft(nLeft);

std::vector<int> arrRight(nRight);

// Создаём массивы левой и правой частей массива

for (int i=0; i<nLeft; i++) arrLeft[i] = arr[left + i];

for (int i=0; i<nRight; i++) arrRight[i] = arr[mid + 1 + i];

int i = 0;

int j = 0;

int k = left;

// Слияние двух отсортированных массива в один

while (i<nLeft && j<nRight) {

if (arrLeft[i]<=arrRight[j]) {

arr[k] = arrLeft[i];

i++;

} else {

arr[k] = arrRight[j];

j++;

}

k++;

}

// Добавояем отсавшиеся эл-ты в итоговый массив

while (i<nLeft) {

arr[k] = arrLeft[i];

i++;

k++;

}

while (j<nRight) {

arr[k] = arrRight[j];

j++;

k++;

}

}

// Сортировка слиянием

// Использование памяти: O(n) (т. к создаём 2 массива в общей длины n)

// Сложность алгоритма: O(n log n) (рекурсивно разбиваем log n и выполняем слияние O(n))

std::vector<int> mergeSort(std::vector<int> arr, int left, int right) {

if (left<right) {

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

arr = mergeSort(arr, left, mid);

arr = mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

return arr;

}

// Функция для создания случайного массива

std::vector<int> arrRand(const int& n, const int& arrMin, const int& arrMax) {

std::vector<int> arr(n);

std::random\_device rd;

std::mt19937 seed(rd());

arr[0] = arrMax;

arr[n - 1] = arrMin;

for (int i=1; i<n-1; ++i) arr[i] = arrMin + seed() % (arrMax - arrMin);

return arr;

}

std::vector<int> arrWorse(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = n - i;

return arr;

}

std::vector<int> arrBett(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = i;

return arr;

}

// Вывод

void display(const std::vector<int>& arr, const std::vector<int>& arrSort, std::chrono::duration<double> duration) {

/\*

int n = arr.size();

std::cout << "\nРазмер массива:" << n;

\*/

std::cout << duration.count() << ", ";

/\*

std::cout << "\nИсходный массив:\n";

for (int i = 0; i < n; i++) std::cout << arr[i] << " ";

std::cout << "\nОтсортированный массив:\n";

for (int i = 0; i < n; i++) std::cout << arrSort[i] << " ";

\*/

}

// Проверка на корректность сортировки

bool check(const std::vector<int>& arr) {

int n = arr.size();

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

if (arr[i - 1]>arr[i]) return false;

}

return true;

}

// Тест для алгоритма сортировки

void test(const int& n = 10, const int& arrMax = 100, const int& arrMin = 0) {

std::vector<int> arr = arrRand(n, arrMin, arrMax);

auto start = std::chrono::high\_resolution\_clock::now();

std::vector<int> arrSort = mergeSort(arr, 0, n - 1);

assert(check(arrSort));

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

display(arr, arrSort, duration);

}

int main() {

test(1000, 999999, 0);

return 0;

}

Листинг кода файла RadixSort.cpp

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <algorithm>

#include <cassert>

// Функция для сортировки подсчётом на основе текущего разряда

void countingSort(std::vector<int>& arr, const int& exp) {

int n = arr.size();

std::vector<int> res(n);

int count[10] = {0}; // Массив для подсчёта кол-во цифр 0-9

// Подсчёт появление Разряда

for (int i=0; i<n; i++) count[(arr[i] / exp) % 10]++;

// Чтобы count[i] содержал индекс разряда в итоговом массиве

for (int i=1; i<10; i++) count[i] += count[i - 1];

// Итоговый массив

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

res[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

arr = res;

}

// Подразрядная сортировка

// Использование памяти: O(n \* r) r - разряд числа

// Сложность алгоритма: O(n \* r) r - разряд числа

std::vector<int> radixSort(std::vector<int> arr) {

// Максимальный эл в массиве

int arrMax = \*max\_element(arr.begin(), arr.end());

// Сортировка для каждого разряда

for (int exp=1; arrMax/exp>0; exp\*=10) countingSort(arr, exp);

return arr;

}

// Функция для создания случайного массива

std::vector<int> arrRand(const int& n, const int& arrMin, const int& arrMax) {

std::vector<int> arr(n);

std::random\_device rd;

std::mt19937 seed(rd());

arr[0] = arrMax;

arr[n - 1] = arrMin;

for (int i=1; i<n-1; ++i) arr[i] = arrMin + seed() % (arrMax - arrMin);

return arr;

}

std::vector<int> arrWorse(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = std::pow(10, i);

return arr;

}

std::vector<int> arrBett(const int& n) {

std::vector<int> arr(n);

for (int i=0; i<n; ++i) arr[i] = i;

return arr;

}

// Вывод

void display(const std::vector<int>& arr, const std::vector<int>& arrSort, std::chrono::duration<double> duration) {

/\*

int n = arr.size();

std::cout << "\nРазмер массива:" << n;

\*/

std::cout << duration.count() << ", ";

/\*

std::cout << "\nИсходный массив:\n";

for (int i = 0; i < n; i++) std::cout << arr[i] << " ";

std::cout << "\nОтсортированный массив:\n";

for (int i = 0; i < n; i++) std::cout << arrSort[i] << " ";

\*/

}

// Проверка на корректность сортировки

bool check(const std::vector<int>& arr) {

int n = arr.size();

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

if (arr[i - 1]>arr[i]) return false;

}

return true;

}

// Тест для алгоритма сортировки

void test(const int& n = 10, const int& arrMax = 100, const int& arrMin = 0) {

std::vector<int> arr = arrRand(n, arrMin, arrMax);

auto start = std::chrono::high\_resolution\_clock::now();

std::vector<int> arrSort = radixSort(arr);

assert(check(arrSort));

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

display(arr, arrSort, duration);

}

int main() {

test(1000, 999999, 0);

return 0;

}