ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Реализация алгоритмов сортировки»

Выполнил работу

Эль-Сибаи Амир

Академическая группа №J3112

Принято

Ассистент (квалификационная категория "ассистент"), Дунаев Максим

Санкт-Петербург

2024

Введение

**Целью** данной работы является изучение алгоритмов сортировок

**Задачи** лабораторной работы:

* Изучить алгоритмы сортировок
* Реализация алгоритмов
* Подсчёт асимптотики и используемой оперативной памяти
* Тестировка и вывод результата

**Теоретическая подготовка**

Для реализации лабораторной работы потребовалось:

* Знание языка C++
* Умение оптимизировать работу алгоритма и расход памяти
* Знание и умение использования разных видов сортировок

**Реализация**

Для выполнения задачи был разработан код на C++ с использованием следующих сортировок:

* shakerSort для работы с массивами
* insertionSort для математических вычислений
* radixSort для работы с предельными значениями

Экспериментальная часть

**ShakerSort**

Алгоритм (приложение 1) сравнивает попарно числа по индексу [i] и [i + 1], меняя их местами по возрастанию и убыванию, проходя, соответственно, слева направо и справа налево. Чтобы убедиться, что вообще были какие-нибудь перестановки, используется флаг.

**Асимптотика и используемая память:**

Т.к. shakerSort – это модифицированная пузырьковая сортировка, то худший и средний случаи O(n^2), лучший случай – O(n), пространственная сложность O(1).

**Приложение 1**

#include <iostream>  
#include <vector>  
using namespace std;  
  
void shakerSort(vector<int>& arr) {  
 int left = 0; // Индекс начала неотсортированного участка  
 int right = arr.size() - 1; // Индекс конца неотсортированного участка  
 bool swapped = true; // Флаг, указывающий на наличие обменов  
  
 while (left < right && swapped) {  
 swapped = false;  
  
 // Проход слева направо  
 for (int i = left; i < right; ++i) {  
 if (arr[i] > arr[i + 1]) {  
 swap(arr[i], arr[i + 1]); // Обмен элементов, если текущий больше следующего  
 swapped = true; // Устанавливаем флаг: был обмен  
 }  
 }  
 --right; // Уменьшаем правую границу, т.к. последний элемент теперь на своём месте  
  
 // Проход справа налево  
 for (int i = right; i > left; --i) {  
 if (arr[i] < arr[i - 1]) {  
 swap(arr[i], arr[i - 1]); // Обмен элементов, если текущий меньше предыдущего  
 swapped = true; // Устанавливаем флаг: был обмен  
 }  
 }  
 ++left; // Увеличиваем левую границу, т.к. первый элемент теперь на своём месте  
 }  
}  
  
int main() {  
 vector<int> arr = {64, 34, 25, 12, 22, 11, 90};  
  
 cout << "Исходный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 shakerSort(arr);  
  
 cout << "Отсортированный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 return 0;  
}

**InsertionSort**

Алгоритм (приложение 2) перебирает по одному числу и сравнивает его с предыдущими. Как только попадаем в ситуацию x1 < x < x2, число вставляем в промежуток, все числа, что правее, сдвигаются.

**Асимптотика и используемая память:**

Худший и средний случаи O(n^2), лучший случай – O(n), пространственная сложность O(1).

**Приложение 2**

#include <iostream>  
#include <vector>  
  
using namespace std;  
  
void insertionSort(vector<int>& arr) {  
 int n = arr.size();  
 for (int i = 1; i < n; i++) {  
 int key = arr[i]; // Текущий элемент  
 int j = i - 1;  
  
 // Перемещаем элементы, которые больше key, вправо  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j--;  
 }  
  
 // Вставляем текущий элемент в правильное место  
 arr[j + 1] = key;  
 }  
}  
  
int main() {  
 vector<int> arr = {3414, -1, 2337, 55, 22, 147, 2201, 11};  
  
 cout << "Исходный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 insertionSort(arr);  
  
 cout << "Отсортированный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 return 0;  
}

**RadixSort**

Алгоритм (приложение 3) рассматривает каждое число поразрядно, приводя их постепенно к отсортированному виду.

**Асимптотика и используемая память:**

Т.к. radixSort для своей работы использует ещё coutingSort, сложность будет выше. radixSort рассматривает каждое число поразрядно, поэтому худшая, средняя и лучшая сложности будут равны O(d \* (n + k)), где n – размер массива, d - максимальное количество разрядов, k – максимальное значение в разряде (в 10сс k = 10). Пространственная сложность – O(n + k), т.к. используется 2 массива: входной и для подсчёта вхождений.

**Приложение 3**

#include <iostream>  
#include <vector>  
  
using namespace std;  
  
// Функция сортировки по разряду  
void countingSort(vector<int>& arr, int exp) {  
 vector<int> count(10, 0); // Массив для подсчета вхождений цифр (от 0 до 9)  
 vector<int> output(arr.size()); // Массив для временного хранения отсортированных данных  
  
 // Подсчитываем количество вхождений цифр в разряде  
 for (int num : arr) {  
 count[(num / exp) % 10]++; // Извлекаем цифру на текущем разряде  
 }  
  
 // Преобразуем массив count, чтобы он хранил позицию для каждой цифры  
 for (int i = 1; i < 10; i++) {  
 count[i] += count[i - 1]; // Преобразование: теперь count[i] указывает на индекс последнего вхождения цифры i  
 }  
  
 // Строим отсортированный массив на основе count  
 for (int i = arr.size() - 1; i >= 0; i--) {  
 int digit = (arr[i] / exp) % 10; // Извлекаем цифру на текущем разряде  
 output[count[digit] - 1] = arr[i]; // Помещаем число в правильную позицию  
 count[digit]--; // Уменьшаем счетчик для данной цифры  
 }  
  
 // Копируем отсортированный массив обратно в оригинальный  
 arr = output;  
}  
  
// Основная функция поразрядной сортировки  
void radixSort(vector<int>& arr) {  
 int maxVal = \*max\_element(arr.begin(), arr.end()); // Находим максимальное значение в массиве  
  
 // Проходим по всем разрядам (от младшего к старшему)  
 for (int exp = 1; maxVal / exp > 0; exp \*= 10) {  
 countingSort(arr, exp); // Сортируем массив по текущему разряду  
 }  
}  
  
int main() {  
 // Пример массива для сортировки  
 vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66};  
  
 // Выводим исходный массив  
 cout << "Исходный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 // Сортируем массив с помощью поразрядной сортировки  
 radixSort(arr);  
  
 // Выводим отсортированный массив  
 cout << "Отсортированный массив: ";  
 for (int num : arr) {  
 cout << num << " ";  
 }  
 cout << endl;  
  
 return 0;  
}

**Test**

Для тестирования работы всех алгоритмов сортировки была разработана программа (приложение 4). В ней представлены функции для генерации лучшего, среднего, худшего случая для всех сортировок и запуска тестов.

**Приложение 4**

#include <iostream>

#include <vector>

#include <algorithm> // Для is\_sorted

using namespace std;

void shakerSort(vector<int>& arr) {

int left = 0;

int right = arr.size() - 1;

bool swapped = true;

while (left < right && swapped) {

swapped = false;

for (int i = left; i < right; ++i) {

if (arr[i] > arr[i + 1]) {

swap(arr[i], arr[i + 1]);

swapped = true;

}

}

--right;

for (int i = right; i > left; --i) {

if (arr[i] < arr[i - 1]) {

swap(arr[i], arr[i - 1]);

swapped = true;

}

}

++left;

}

}

void insertionSort(vector<int>& arr) {

int n = arr.size();

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

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

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

vector<int> count(10, 0);

vector<int> output(arr.size());

for (int num : arr) {

count[(num / exp) % 10]++;

}

for (int i = 1; i < 10; i++) {

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

}

for (int i = arr.size() - 1; i >= 0; i--) {

int digit = (arr[i] / exp) % 10;

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

count[digit]--;

}

arr = output;

}

void radixSort(vector<int>& arr) {

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

for (int exp = 1; maxVal / exp > 0; exp \*= 10) {

countingSort(arr, exp);

}

}

void testSortingAlgorithm(void (\*sortFunc)(vector<int>&), const string& name) {

// Лучший случай

vector<int> bestCase = {1, 2, 3, 4, 5};

sortFunc(bestCase);

if (is\_sorted(bestCase.begin(), bestCase.end())) {

cout << name << " passed best case!" << endl;

} else {

cout << name << " failed best case!" << endl;

}

// Худший случай

vector<int> worstCase = {5, 4, 3, 2, 1};

sortFunc(worstCase);

if (is\_sorted(worstCase.begin(), worstCase.end())) {

cout << name << " passed worst case!" << endl;

} else {

cout << name << " failed worst case!" << endl;

}

// Средний случай

vector<int> averageCase = {3, 1, 4, 5, 2};

sortFunc(averageCase);

if (is\_sorted(averageCase.begin(), averageCase.end())) {

cout << name << " passed average case!" << endl;

} else {

cout << name << " failed average case!" << endl;

}

}

int main() {

testSortingAlgorithm(shakerSort, "ShakerSort");

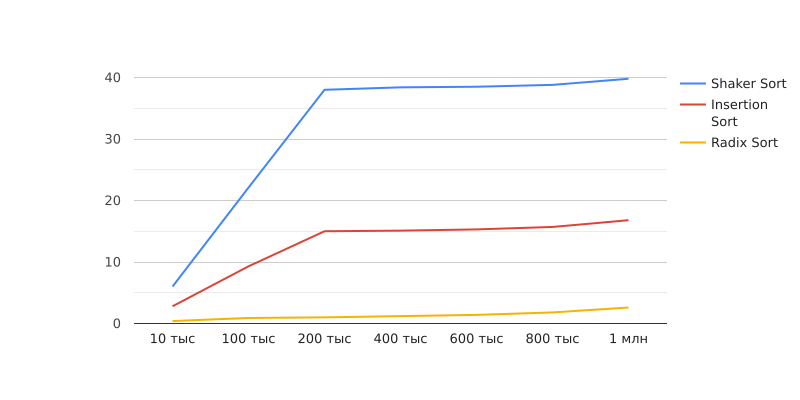
testSortingAlgorithm(insertionSort, "InsertionSort");

testSortingAlgorithm(radixSort, "RadixSort");

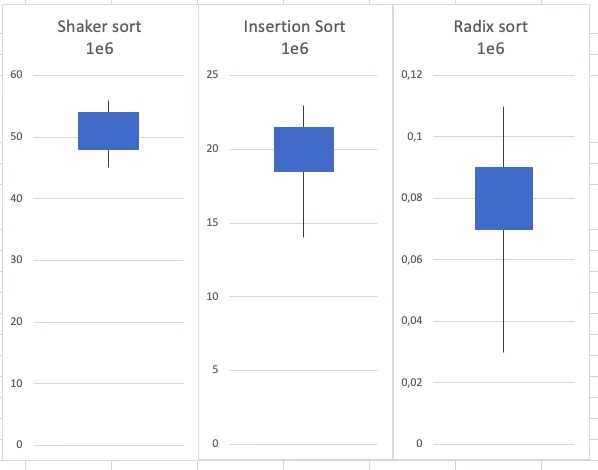
return 0;

}

**Приложение 5 (Линейный график времени работы алгоритмов)**



**Приложение 6 (Каскадный график времени работы алгоритмов)**



Заключение

Ожидаемая асимптотика примерно сошлась с реальным временем. Выбросы на графиках незначительны и объяснимы не идеальностью оборудования, на котором производилось тестирование.

ShakerSort оказалась самой неэффективной, это неудивительно, т.к. асимптотика O(n^2). Заключается это в сложности алгоритма, т.к. он перебирает абсолютно все значения массива с двух сторон.

InsertionSort не самый оптимальный алгоритм. Тоже не самый быстрый из-за асимптотики O(n^2), но работает быстрее, чем ShakerSort из-за другой логики сортирорвки. Может быть применим при не очень больших массивах.

RadixSort значительно быстрее других алгоритмов, об этом говорит лучшая асимптотика O(d \* (n + k)). Является самым универсальным из представленных алгоритмов, подходит для больших массивов.