Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

Выполнил студент группы КС-38 (Казанцев Леонард Антонович)

Ссылка на репозиторий: (https://github.com/SimpleMaking/Algorithms-6-sem-/tree/main/Lab\_1(сортировка))

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: (17.02.2023)

Оглавление

[Описание задачи. 2](#_Toc127270410)

[Описание метода/модели. 3](#_Toc127270411)

[Выполнение задачи. 4](#_Toc127270412)

[Заключение. 4](#_Toc127270413)

# Описание задачи.

В данной лабораторной работе необходимо было реализовать один из видов сортировки. Мне досталась сортировка вставками. Эта сортировка считается одной из самых универсальных: время работы и реализация.

Помимо реализации еще нужно было провести тесты работы алгоритма для всех значений N из списка (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000), при этом:

* в каждом тесте необходимо по 20 раз генерировать вектор, состоящий из N элементов
* каждый элемент массива заполняется случайным числом с плавающей запятой от -1 до 1

По окончанию всех тестов необходимо нанести все точки, полученные в результате замеров времени на график где на ось абсцисс(Х) нанести N, а на ось ординат(Y) нанести значения времени на сортировку. По полученным точкам построить график лучшего (минимальное время для каждого N), худшего (максимальное время для каждого N) и среднего (среднее время для каждого N) случая.

В качестве дополнительного задания, необходимо построить график худшего случая, и график O(c \* g(N)), где g(N) соответствует асимптотической сложности рассматриваемого метода сортировки, подобрав такое значение C, что бы начиная с N ~ 1000 график асимптотической сложности возрастал быстрее чем полученное худшее время, но при этом был различим на графике.

# Описание метода/модели.

Сортировка вставками

Задача заключается в следующем: есть часть массива, которая уже отсортирована, и требуется вставить остальные элементы массива в отсортированную часть, сохранив при этом упорядоченность. Для этого на каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной части массива, до тех пор пока весь набор входных данных не будет отсортирован. Метод выбора очередного элемента из исходного массива произволен, однако обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.

Так как в процессе работы алгоритма могут меняться местами только соседние элементы, каждый обмен уменьшает число инверсий на единицу. Следовательно, количество обменов равно количеству инверсий в исходном массиве вне зависимости от реализации сортировки. Максимальное количество инверсий содержится в массиве, элементы которого отсортированы по не возрастанию. Число инверсий в таком массиве .

Алгоритм работает за O(n + k), где k — число обменов элементов входного массива, равное числу инверсий. В среднем и в худшем случае — за . Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие — когда они расположены в обратном порядке.

# Выполнение задачи.

Для решения поставленной задачи был использован язык программирования С++ и Python. Первый использовался для получения максимального, минимального и среднего времени выполнения алгоритма для каждого из 8 проведенных тестов. В каждом тесте производилось 20 попыток.

Алгоритм:

Функция алгоритма paste\_sort принимает первым аргументом указатель на сортируемый массив, вторым аргументом – размерность массива.

В самой функции с помощью двух циклов, один из которых -вложенный, передвигаемся по массиву и, если пред. элемент удовлетворяет условию сортировки (больше текущего элемента или меньше), то меняем их местами. Производим эти операции по всему массиву.

Код функции:

double\* paste\_sort(double\* data, size\_t n)

{

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

for (int j = i; j > 0 && data[j - 1] > data[j]; j--) // пока j>0 и элемент j-1 > j, x-массив int

swap(data[j - 1], data[j]); // меняем местами элементы j и j-1

return data;

}

Код main функции:

int main(void)

{

std::ofstream out;

double min[8], max[8], avg[8] = { 0 };

for (size\_t i = 0; i < 8; i++)

{

min[i] = 999999;

max[i] = -999999;

}

out.open("output.txt");

size\_t counter = 0;

// get times for all tests

for (auto value : CONST\_SIZE\_ARR)

{

double\* data = new double[value];

for (size\_t i = 0; i < 20; i++)

{

for (size\_t k = 0; k < value; k++)

data[k] = A + (B - A) \* rand() / (float)RAND\_MAX;

chrono::high\_resolution\_clock::time\_point start = chrono::high\_resolution\_clock::now();

double\* sorted\_data = paste\_sort(data, value);

chrono::high\_resolution\_clock::time\_point end = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> milli\_diff = end - start;

if (milli\_diff.count() < min[counter])

{

min[counter] = milli\_diff.count();

}

if (milli\_diff.count() > max[counter])

{

max[counter] = milli\_diff.count();

}

avg[counter] += milli\_diff.count();

}

avg[counter] /= 20;

counter += 1;

delete[] data;

}

//output data into the file output.txt

out << "N min max avg" << endl;

for (size\_t i = 0; i < 8; i++)

{

out << std::setprecision(3) << CONST\_SIZE\_ARR[i] << " " << min[i] << " " << max[i] << " " << avg[i] << endl;

}

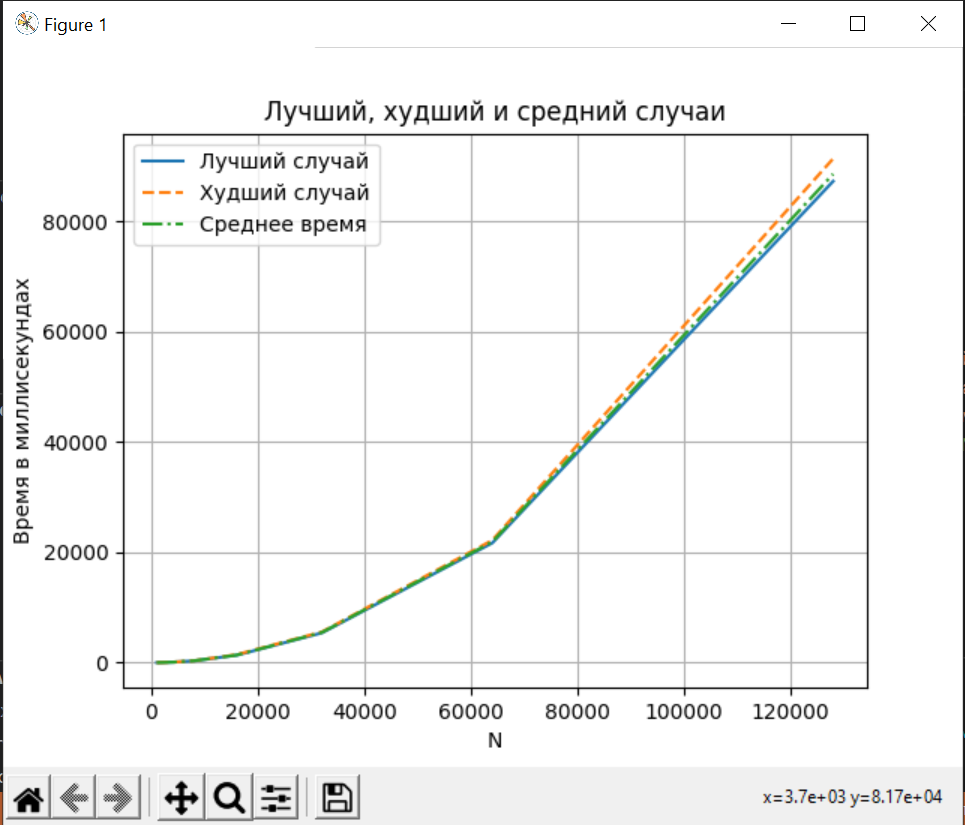
out.close();

system("python graphics.py");

return 0;

}

Вывод графиков для мин. макс. и среднего значений осуществлялся через скрипт graphics.py:



Лучший, худший и средний случаи



Лучший, худший и средний случаи, а также график O(c \* g(N))

# Заключение.

Алгоритм показался мне прост в реализации, также радует время его работы: самый долгий подсчет для 128000 чисел не занял больше 6 миллисекунд. Считаю этот алгоритм универсальным, и буду продолжать его использовать в своих целях.