ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Название лабораторной работы (Вашего задания)»

Выполнил работу
Афанасьев Николай Игоревич
Академическая группа №J3113
Принято:
Ходненко Иван Владимирович

Санкт-Петербург
2024

Структура отчёта:

- ***1. Введение***

В этой лабораторной работе мне нужно было выбрать 3 алгоритма сортировки и изучить их свойства и принципы. Я выбрал Shuttle Merge и Counting sort.

Цель: Изучить 3 алгоритма сортировки , понять чем они различаются и подвести итог.

Задачи:

1) Написать красивый код для каждого алгоритма сортировок

2) Подсчет по памяти для 3-х алгоритмов сортировок

3) Подсчет асимптотики для 3-х алгоритмов сортировок

4) Сделать для каждого юнит тесты(в зав от сложности каждого алгоритма)

5) Сделать графики зависимости

6) Сделать вывод по всем алгоритмам и всей работе в целом!

- ***2. Теоретическая подготовка***

Асимптотическая сложность (Asymptotic Complexity)

Асимптотическая сложность — это способ описания того, как время выполнения или потребление памяти алгоритмом растет с увеличением размера входных данных. Она не фокусируется на точных значениях (в секундах или байтах), а скорее на *темпе роста* ресурсов.

Shuttle Sort : (или Сортировка челноком) — это один из алгоритмов сортировки, который напоминает пузырьковую сортировку, но с улучшением за счет того, что элементы могут перемещаться не только вперед, но и назад.

Это делает его похожим на сортировку вставками, но с дополнительной "подгонкой" элементов в процессе.

Сложность:

- В худшем случае ($O(n^2)$): когда массив полностью не отсортирован.
- В лучшем случае ($O(n)$): когда массив уже отсортирован.

Память:

- Основная память: $O(1)$ — сортировка производится на месте, не требуются дополнительные структуры данных.

Merge Sort : (сортировка слиянием) — это популярный алгоритм сортировки, основанный на принципе разделяй и властвуй. Алгоритм делит массив на две половины, рекурсивно сортирует каждую из них и затем сливает две отсортированные половины в один массив.

Сложность:

- Лучший, худший и средний случай: $O(n\log n)$.
- Дополнительная память: $O(n)$ из-за хранения временных массивов для слияния.
- Стабильный алгоритм (не меняет порядок элементов с одинаковыми значениями).
- Подходит для больших данных, но может быть менее эффективным для маленьких массивов из-за дополнительных затрат памяти.

Counting Sort (сортировка подсчетом) — это не сравнивающий алгоритм сортировки, который используется для сортировки чисел или объектов с ключами, представляющими собой целые числа в определенном диапазоне. Вместо сравнения элементов, он подсчитывает количество их вхождений и использует эти данные для упорядочивания.

Временная сложность:

- Лучший, средний и худший случай: $O(n+k)$, где n — размер массива, k — диапазон значений ($k=max-min+1$).

Память:

- Дополнительная память O(n+k):
  - $O(k)$ — для вспомогательного массива `count`.
  - $O(n)$ — для выходного массива, если он создается отдельно.

- ***3 Реализация***

1) Shuttle Sort

1. Реализация Shuttle Sort

- Написана функция `shuttleSort`, которая сортирует массив методом "шаттл-сортировки" (аналог улучшенной версии сортировки вставками):
    - Внешний цикл проходит по всем элементам массива.
    - Внутренний цикл "шаттлит" текущий элемент влево, пока он не окажется на своем месте относительно предыдущих элементов.

2. Проверка массива на отсортированность

- Написана функция `isSorted`, которая проверяет, отсортирован ли массив:
    - Проходит по всем элементам массива.
    - Если обнаруживается, что текущий элемент меньше предыдущего, массив считается не отсортированным (возвращается `false`).
    - Если цикл завершается, значит массив отсортирован, и функция возвращает `true`.

3. Генерация массива

- Написана функция `generateArray`, которая создает массив определенного типа:
    - Лучший случай (type == 1): Массив уже отсортирован. Последовательно заполняется числами от 1 до n.
    - Средний случай (type == 2): Массив заполняется случайными числами (используется `rand()`).
    - Худший случай (type == 3): Массив заполняется числами в обратном порядке, что является наиболее неэффективным случаем для большинства сортировок.

4. Тестирование алгоритма

- Написана функция `runTests`, которая проверяет корректность работы `shuttleSort`:
    - Лучший случай: Генерируется отсортированный массив, сортируется, затем проверяется с помощью `assert` на отсортированность.
    - Средний случай: Генерируется массив случайных чисел, сортируется и проверяется.
    - Худший случай: Генерируется массив в обратном порядке, сортируется и проверяется.
    - Если массив после сортировки отсортирован, выводится сообщение об успешном прохождении теста.

5. Основная программа

- В функции `main` происходит:

- Установка локали на русский (`setlocale`), чтобы корректно выводить сообщения на русском языке.
- Инициализация генератора случайных чисел через `srand(time(0))`, чтобы случайные числа в массиве были разными при каждом запуске.
- Вызов функции `runTests` для проверки работы алгоритма.
- Если все тесты проходят, выводится сообщение о завершении.
-

В последующем я не буду повторять пункты с генерацией, юнит тестами и Мейном

2) `mergeSort`

1. Функция для слияния двух подмассивов

Функция `merge`:

- Принимает вектор `arr` и индексы: `left`, `mid`, `right`.
- Делит массив на две части:
  - Левая часть — от `left` до `mid`.
  - Правая часть — от `mid + 1` до `right`.
- Создает временные массивы `L` и `R` для хранения элементов левой и правой половин.
- Копирует элементы из `arr` в `L` и `R`.
- Выполняет слияние, попутно сортируя:
  - Сравнивает текущие элементы из `L` и `R` и вставляет меньший в `arr`.
  - Оставшиеся элементы из `L` или `R` копируются в `arr`.

2. Рекурсивная сортировка (Merge Sort)

Функция `mergeSort`:

- Рекурсивно делит массив на две части:
  - Сортирует левую половину.
  - Сортирует правую половину.
- Сливает две отсортированные части, используя функцию `merge`.

Условие выхода из рекурсии: если массив состоит из одного элемента или пуст.

3. Проверка отсортированности массива

Функция `isSorted`:

- Проверяет, является ли массив отсортированным.
- Если хотя бы один элемент больше следующего, возвращает `false`.

3. Counting Sort

1.Функция `countingSort`:

- Входные данные: массив `arr` и максимальное значение `maxValue`.
- Шаги алгоритма:
    - Создается массив `count`, где каждый индекс соответствует числу из диапазона [0, maxValue], а значение — количеству его вхождений в `arr`.
    - Заполняется массив `count` подсчетом вхождений каждого элемента из `arr`.
    - Используется массив `output` для записи отсортированных значений:
        - Проходит по массиву `count`.
        - Для каждого индекса (числа) с ненулевым количеством вхождений добавляются числа в `output`.
    - Копирует отсортированный массив `output` обратно в `arr`.

2. Генерация массива

Функция `generateArray`:

- Заполняет массив `arr` случайными числами в диапазоне [0, maxValue].
- Использует `rand()` для генерации чисел.

3. Проверка отсортированности

Функция `isSorted`:

- Проходит по массиву и проверяет, является ли каждый следующий элемент больше или равен предыдущему.
- Возвращает `true`, если массив отсортирован, иначе `false`.

- ***Экспериментальная часть***

В этом разделе вам необходимо привести результаты работы вашего алгоритма, с таблицами и графиками, демонстрирующими выполнения алгоритма с различными условиями и наборами данных. Оценивается производительность и сравниваются результаты с теоретическими оценками.

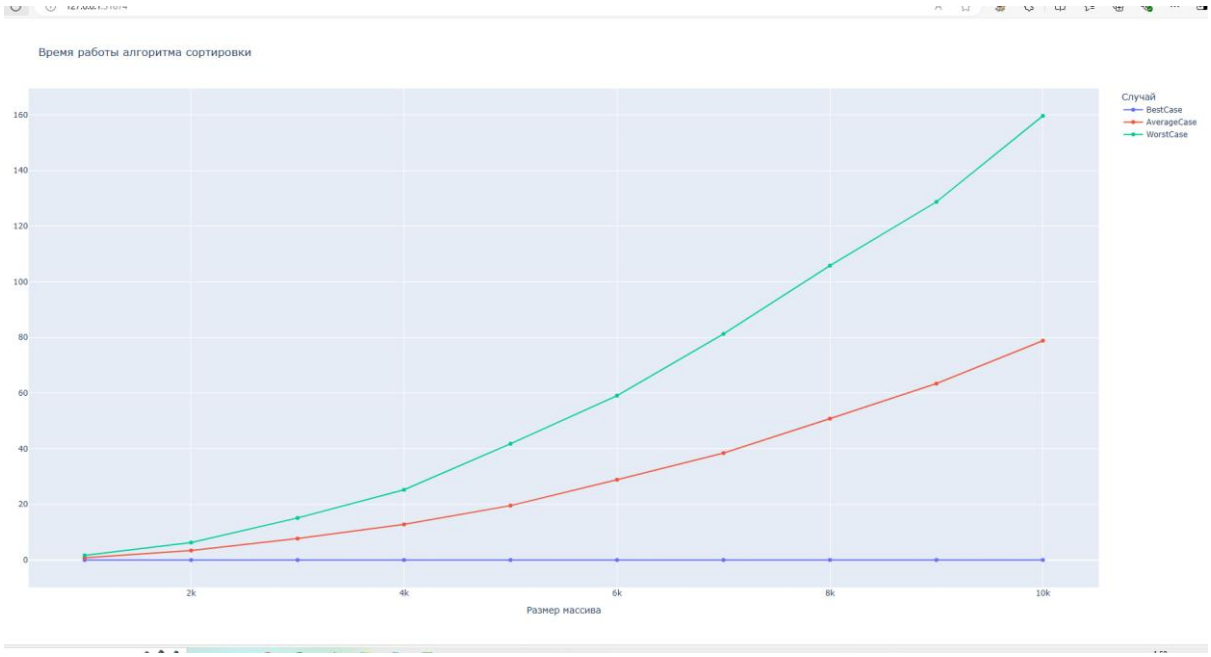
Таблица - Подсчёт сложности shuttle sort

Размер входного набора	2000	4000	5000	7000	8000	10000

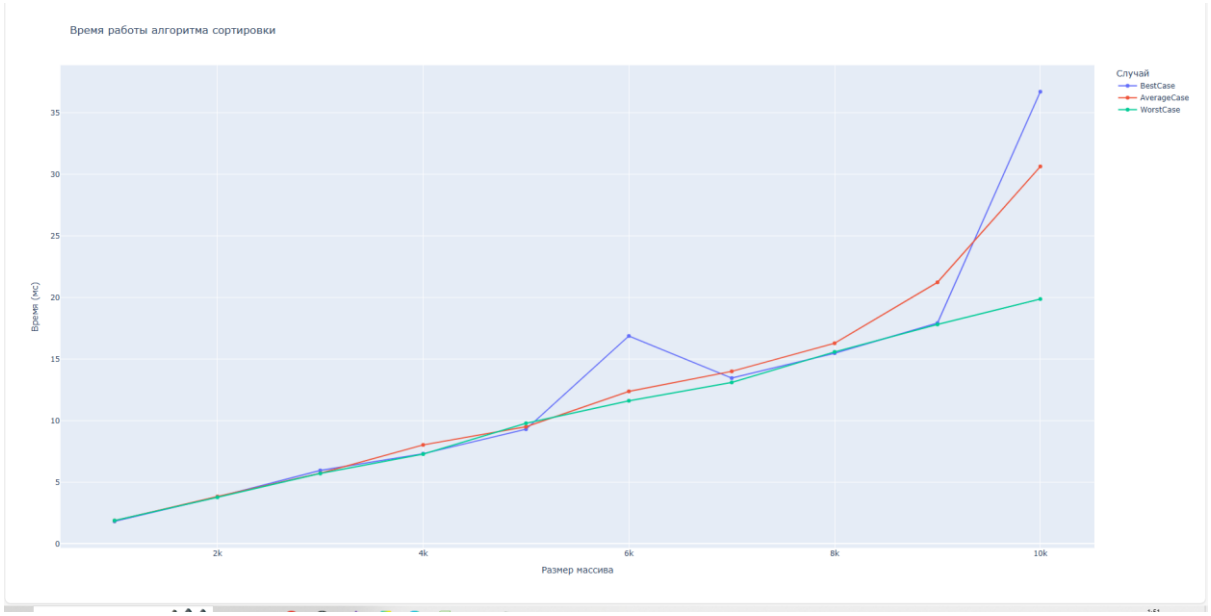
Время выполнения программы, с	0,0049	0,0081	0,0102	0,0142	0,0165	0,203
O(N), лучший случай,с	2000	4000	5000	7000	8000	10000
Время выполнения программы,с	6,2	25,4	41,9	81,2	105,7	159
Худший o(n**2)	40000000	160000000	250000000	490000000	640000000	1000000000
Память 1)для лучшего байт	2000*4	4000*4	5000*4	7000*4	8000*4	10000*4
Память 2)для худшего байт	40000000*4	160000000*4	250000000*4	490000000*4	640000000*4	1000000000*4

Тоже самое делаем и с остальными , но для merge - сложность n* log(n)

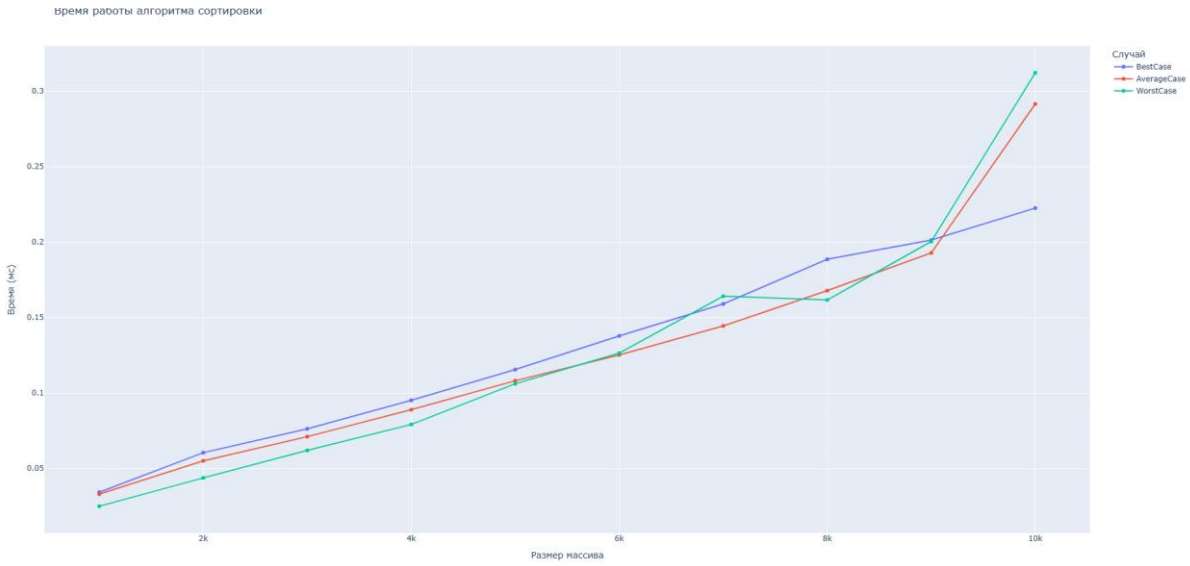
График представляющий визуально удобный формат данных из таблицы представлен на изображении.



Изображение №1 - График работы алгоритма shuttle sort



Изображение №2 - График работы алгоритма merge sort



Изображение №3 - График работы алгоритма counting sort

Проанализировав данные графики можно сказать , что у всех алгоритмов кроме шаттл сорт - схожее время выполнения которое не зависит от количества элементов!!!

- ***Заключение***

Данная лабораторная работа мне очень понравилась!!! Я научился работать с алгоритмами сортировок а также их сложностью и понимать какой куда лучше использовать. Больше всего мне понравилось строить графики зависимости с помощью Питона и искать зависимости. Вообще сама по себе это очень нужная тема при углублении в которую можно многому научиться. Я познакомился с тем как можно записывать данные из с++ в эксель а потом графически переводить их на питон - и это не может не радовать!!!

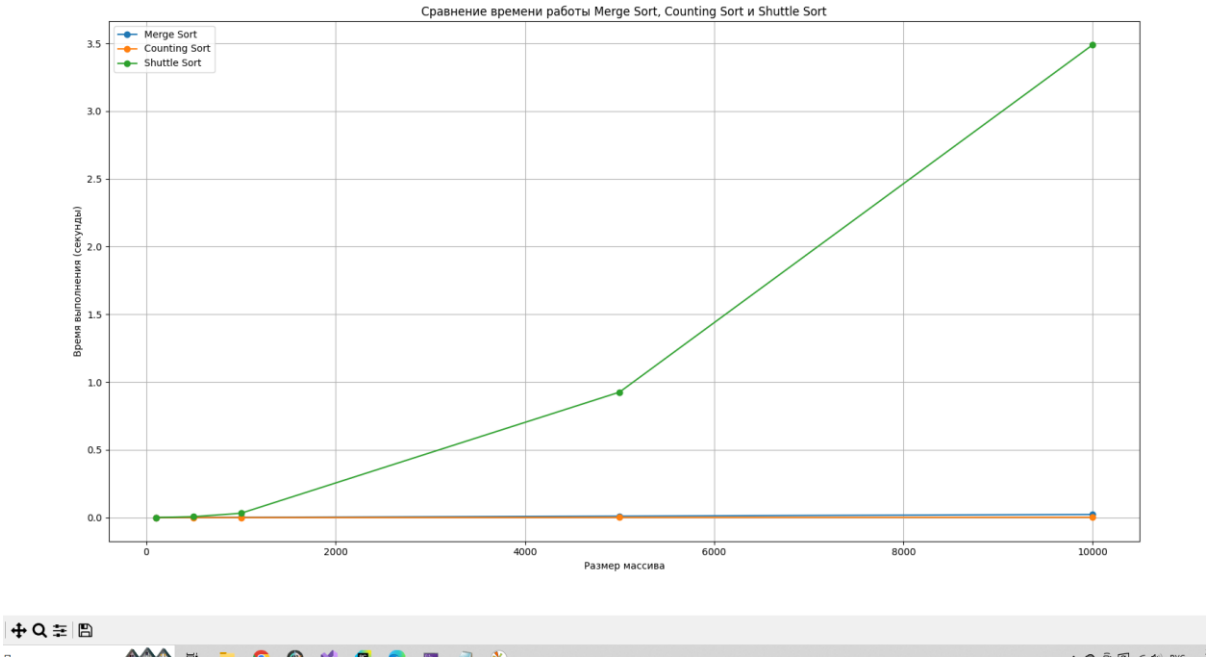
ПРИЛОЖЕНИЕ А

1.Линейное сравнение

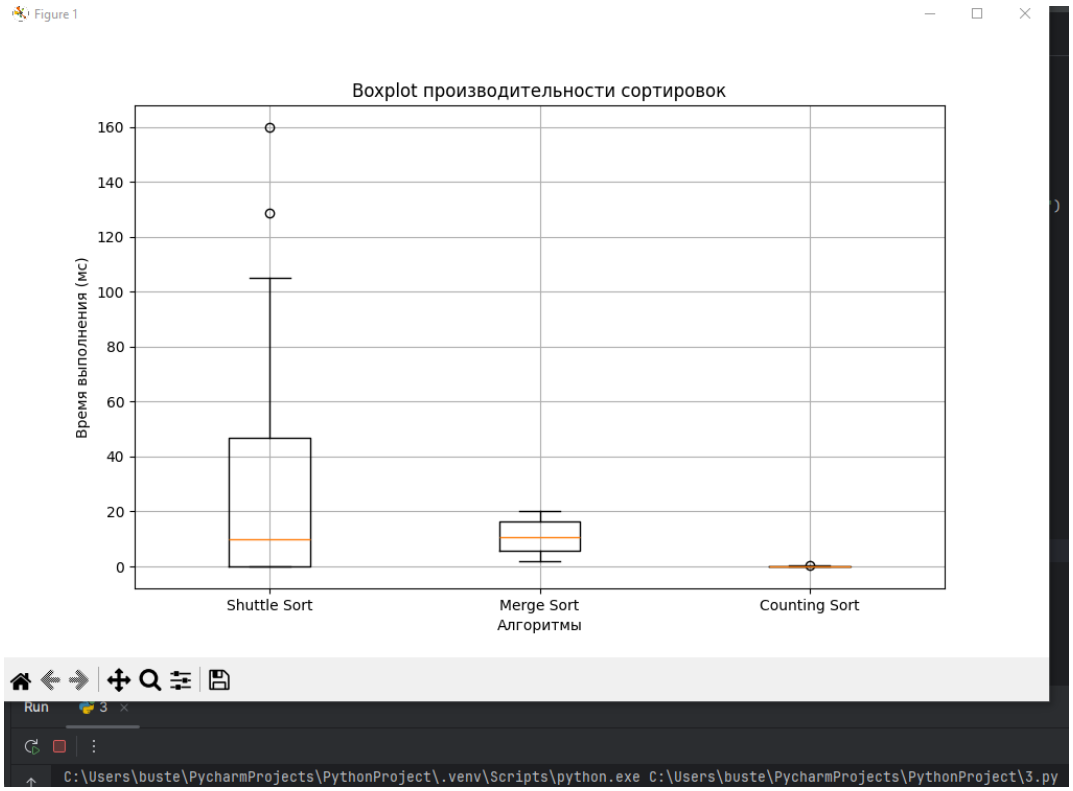
2.Boxplotграфики

3.Юнит тесты и скрины

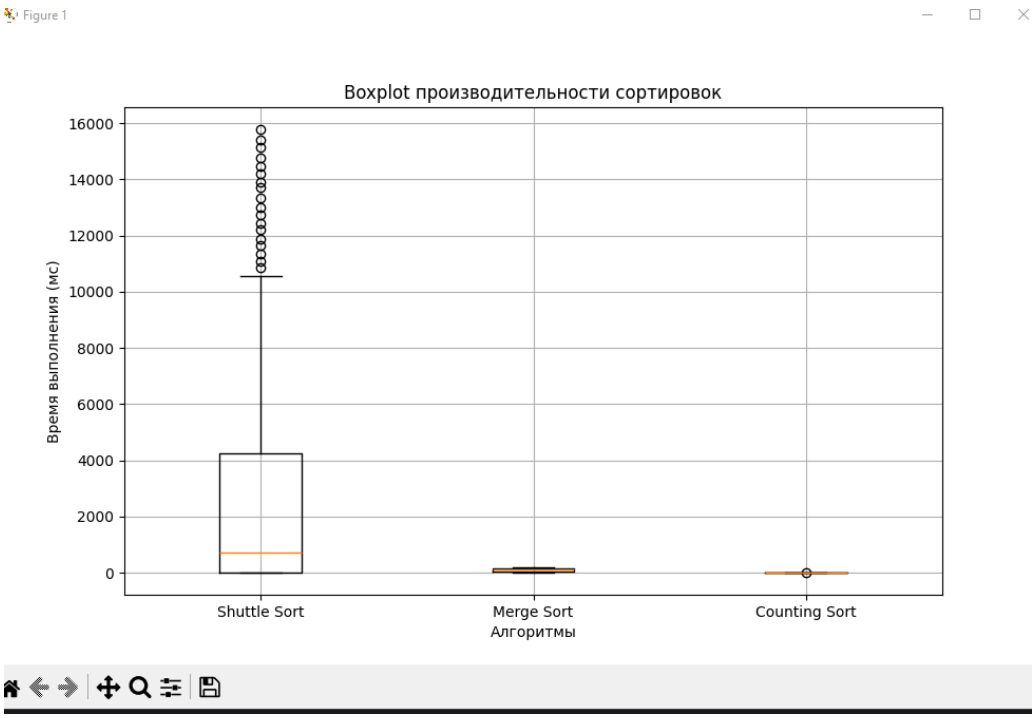
1):



2) Для 1e4



Для 1e5:



3) 1 - shuttle

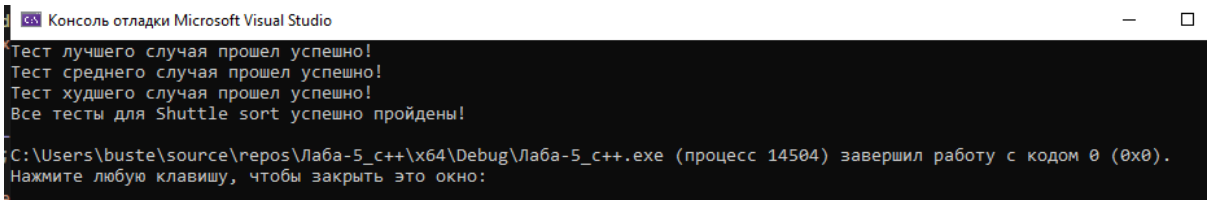
```cpp
// Функция для генерации массива
void generateArray(int arr[], int n, int type) {
    if (type == 1) { // Лучший случай: массив уже отсортирован
        for (int i = 0; i < n; i++) {
            arr[i] = i + 1;
        }
    }
    else if (type == 2) { // Средний случай: случайный массив
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 1000; // Генерируем числа
        }
    }
    else if (type == 3) { // Худший случай: массив в обратном порядке
        for (int i = 0; i < n; i++) {
            arr[i] = n - i;
        }
    }
}

void runTests() {
    const int SIZE = 100; // Размер массива
    int arr[SIZE];

    // Лучший случай
    generateArray(arr, SIZE, 1);
    shuttleSort(arr, SIZE);
    assert(isSorted(arr, SIZE));
    cout << "Тест лучшего случая прошел успешно!" << endl;

    // Средний случай
    generateArray(arr, SIZE, 2);
    shuttleSort(arr, SIZE);
    assert(isSorted(arr, SIZE));
    cout << "Тест среднего случая прошел успешно!" << endl;

    // Худший случай
    generateArray(arr, SIZE, 3);
    shuttleSort(arr, SIZE);
    assert(isSorted(arr, SIZE));
    cout << "Тест худшего случая прошел успешно!" << endl;
}
```



2 – merge

```cpp
// Генерация худшего
void generateWorstCase(vector<int>& arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = n - i; // Массив от N до 1
    }
}

// Генерация среднего
void generateRandomCase(vector<int>& arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }
}

// Юнит-тесты для Merge Sort
void runTests() {
    const int SIZE = 1000000;// можно взять какой хотим
    vector<int> arr(SIZE);

    // лучший
    generateBestCase(arr, SIZE);
    mergeSort(arr, 0, SIZE - 1);
    assert(isSorted(arr)); // Проверка, что массив отсортирован
    cout << "Тест лучшего случая прошел успешно!" << endl;

    //средний
    generateRandomCase(arr, SIZE);
    mergeSort(arr, 0, SIZE - 1);
    assert(isSorted(arr));
    cout << "Тест среднего случая прошел успешно!" << endl;

    //худший
    generateWorstCase(arr, SIZE);
    mergeSort(arr, 0, SIZE - 1);
    assert(isSorted(arr));
    cout << "Тест худшего случая прошел успешно!" << endl;
```

```
Консоль отладки Microsoft Visual Studio
Тест лучшего случая прошел успешно!
Тест среднего случая прошел успешно!
Тест худшего случая прошел успешно!
Все тесты для mergeSort прошли успешно!

C:\Users\buste\source\repos\Лаба-5_c++\x64\Debug\Лаба-5_c++.exe (процесс
Нажмите любую клавишу, чтобы закрыть это окно:
```

3 – counting

```cpp
// Юнит-тесты
void runTests() {
    vector<int> arr;

    // Легкий случай: небольшой массив, большой диапазон значений
    arr.resize(10); // Массив из 10 элементов
    generateArray(arr, 10, 1000); // Диапазон значений [0, 1000]
    countingSort(arr, 1000);
    assert(isSorted(arr));
    cout << "Тест легкого случая пройден успешно!" << endl;

    size_t size = 1000;
    // Средний случай: массив и диапазон значений сопоставимы
    arr.resize(size); // Массив из 1000 элементов
    generateArray(arr, size, 2000); // Диапазон значений [0, 2000]
    countingSort(arr, 2000);
    assert(isSorted(arr));
    cout << "Тест среднего случая пройден успешно!" << endl;

    // Сложный случай: большой массив, небольшой диапазон значений
    arr.resize(1000000); // Массив из 1 000 000 элементов
    generateArray(arr, 1000000, 10000); // Диапазон значений [0, 10000]
    countingSort(arr, 10000);
    assert(isSorted(arr));
    cout << "Тест сложного случая пройден успешно!" << endl;
}

int main() {
```

```
Консоль отладки Microsoft Visual Studio
Тест легкого случая пройден успешно!
Тест среднего случая пройден успешно!
Тест сложного случая пройден успешно!
Все тесты для counting sort прошли успешно!

C:\Users\buste\source\repos\Лаба-5_c++\x64\Debug
Нажмите любую клавишу, чтобы закрыть это окно:
```

Требования к оформлению отчёта:

- Размер шрифта и тип шрифта

Шрифт: Times New Roman.

Размер шрифта: 14 pt.

Межстрочный интервал: 1.5.

Отступы: слева — 30 мм, справа — 10 мм, сверху и снизу — 20 мм.

- Абзацы

Абзацный отступ: 1.25 см.

Выравнивание текста: по ширине.

- Нумерация страниц

Нумерация страниц: снизу по середине, начиная со второй страницы.

- Таблицы

Таблицы нумеруются и имеют заголовок.

Название таблицы пишется над таблицей, справа.

Пример:

Таблица 1 - Результаты тестирования алгоритма

№ Студента	IQ
192455	123

В таблице все данные должны быть выравнены по центру.

- Оформление рисунков и графиков

Все рисунки и графики должны быть подписаны, например:

Рисунок 1 – Кошка и собака в состоянии выброса дофамина

Подпись располагается под рисунком, по центру, также, как и рисунок, без отступа.

На все таблицы, рисунки, схемы и пр. должна быть ссылка в тексте, пример:

«Собаки и кошки всегда рады встретить своего хозяина, вернувшегося с работы, в этот момент в их организме происходит выплеск дофамина, который является следствием дофаминовой награды за ожидание вас в течение дня, пример животных, находящихся в таком состоянии представлен на изображении 1.»

- Списки

Маркированные списки использовать только для перечислений.

Для маркированных списков использовать широкое тире.

Нумерованные списки — для последовательностей шагов.

Для нумерованных списков использовать формат ГОСТ, пример:

- Первый уровень
  - Второй уровень
    - Третий уровень

Ссылка на ГОСТ для оформления отчетов

ГОСТ 7.32-2001: "Отчет о научно-исследовательской работе. Структура и правила оформления". URL: https://csr.itmo.ru/education/nir.html