Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт по лабораторной работе №3

По дисциплине: «Анализ алгоритмов»

Тема: «Трудоемкость алгоритмов»

Студент Мередова Айджахан

Группа ИУ7-56Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва - 2020 г.

Введение

Цель работы - анализ алгоритмов сортировки массивов. Для этого:

1) реализовать три различных алгоритма сортировки;

2) теоретически вычислить эффективность алгоритмов;

3) провести замеры процессорного времени работы реализаций алгоритмов;

1. Аналитическая часть

Сортировка - весьма важная операция, особенно в обработке данных. Также она идеальная тема для демонстарции широкого разнообразия алгоритмов, решаюших одну и ту же задачу, причем многие из них в каком-нибудь смысле оптимальны, и почти из них имеет какие-нибудь преимущества перед остальными. Поэтому здесь хорошо видно, зачем нужен анализ эффективности алгоритмов. Более того, на примере сортировок становится понятно, что можно получить весьма существенный выигрыш в эффективности, разрабатывая хитроумные алгоритмы, даже если уже имеются очевидные методы.

1.1. Описание задачи

Алгоритм сортировки - это алгоритм для упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

1.1.1. Сортировка пузырьком

Простая сортировка выбором, она же пузырьковая использует сравнение и обмен пар соседних элементов до тех пор, пока не будут отсортированы все элементы.

Здесь выполняются повторные проходы по массиву, причем каждый раз наименьший элемент оставшегося множества просеивается в направлении левого конца массива. Если для разнообразия считать массив расположенным вертикально, а не горизонтально, и вообразить, что элементы - это пузырьки в сосуде с водой, причем их вес равен значению ключа, тогда проходы по массиву приводят к подъему пузырька на уровень, соответствующий его весу. Отсюда и название этого алгоритма сортировки.

Сложность по времени:

- худший случай (элементы массива отсортированы в обратном порядке): $O(n^2)$
- средний случай (рандомное расположение элементов массива): $O(n^2)$
- лучший случай (массив отсортирован): $O(n)$

1.1.2. Шейкер-сортировка

Этот алгоритм является улучшенной версией пузырьковой сортировки. Можно встретить другие названия этой сортировки: сортировка перемешиванием, пульсирующая сортировка, двунаправленная сортировка пузырьком.

Этот алгоритм сортировки отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево. Шейкер-сортировка выгодна в тех случаях, когда элементы уже стоят в почти правильном порядке, - но это редко случается на практике. **Сложность по времени:**

- худший случай (элементы массива отсортированы в обратном порядке): $O(n^2)$
- средний случай (рандомное расположение элементов массива): $O(n^2)$
- лучший случай (массив отсортирован): $O(n)$

1.1.3. Сортировка Шелла

В 1959 г. Шелл предложил ускорить сортировку простыми вставками.

В этом алгоритме очень остроумный подход в определении того, какую именно часть массива считать отсортированной. В простых вставках все просто: от текущего элемента всё что слева — уже отсортировано, всё что справа — ещё не отсортировано. В отличие от простых вставок сортировка Шелла не пытается слева от элемента сразу формировать строго отсортированную часть массива. Она создаёт слева от элемента почти отсортированную часть массива и делает это достаточно быстро.

Сортировка Шелла закидывает текущий элемент в буфер и сравнивает его с левой частью массива. Если находит большие элементы слева, то сдвигает их вправо, освобождая место для вставки. Но при этом берёт не всю левую часть, а только некоторую группу элементов из неё, где элементы разнесены друг от друга на некоторое расстояние. Такая система позволяет быстро вставлять элементы примерно в ту область массива, где они должны находиться.

С каждой итерацией основного цикла это расстояние постепенно уменьшается и когда оно становится равным единице, то сортировка Шелла в этот момент превращается в классическую сортировку простыми вставками, которой дали на обработку почти отсортированный массив. А почти отсортированный массив сортировка вставками в полностью отсортированный преобразует быстро.

Сложность по времени:

- худший случай (если неправильно выбраны промежутки): $\mathrm{O}(n(log_n)^2)$
- средний случай (рандомное расположение элементов массива):$\mathrm{O}(n(log_n)^2)$ или $n^{3/2}$
- лучший случай (массив отсортирован): $\mathrm{O}(n)$

2. Конструкторская часть

Рассмотрим сортироку пузырьком, шейкер-сортировку и сортировку Шелла.

2.1. Схемы алгоритмов

На рисунке 1 изображена схема алгоритма сортировки пузырьком.

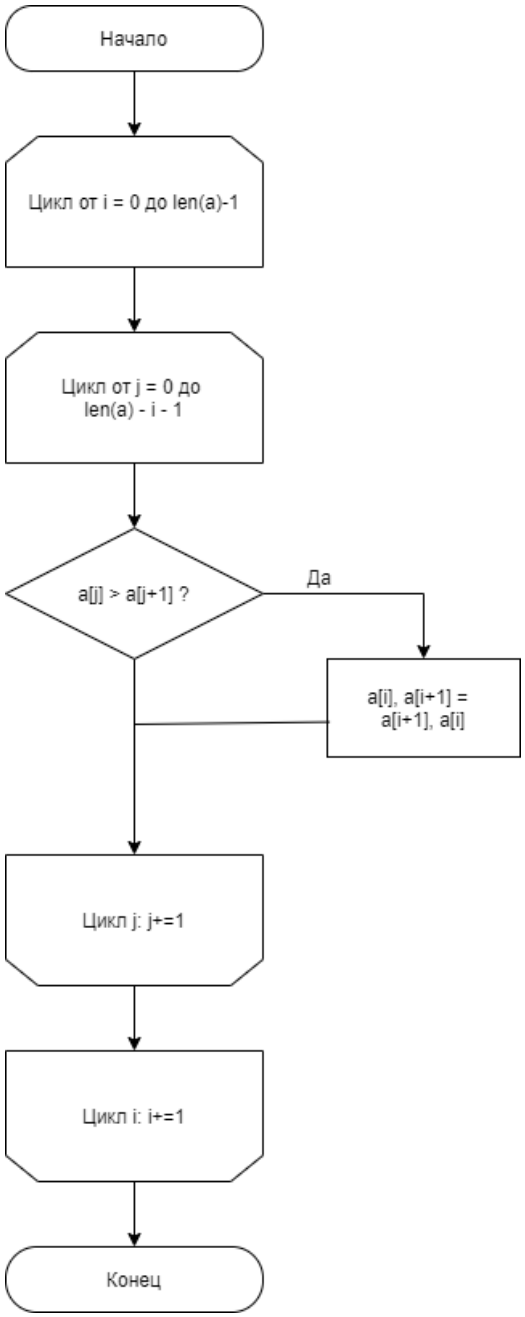

Рис. 1. Схема алгоритма сортировки пузырьком.

На рисунках 2 - 3 изображены схемы алгоритма шейкер-сортировки.

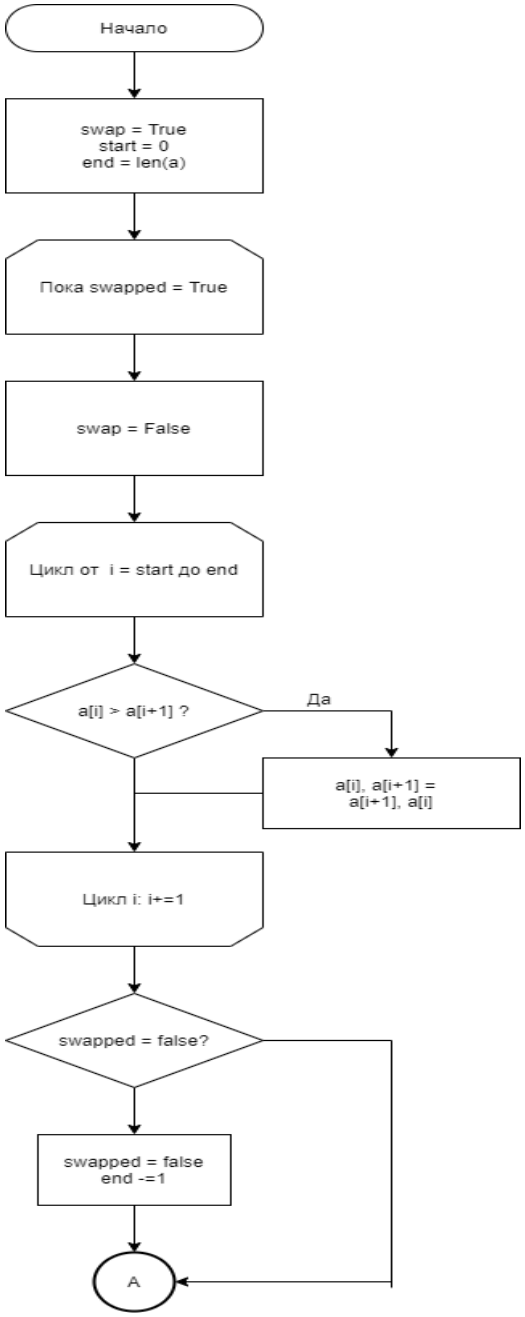

Рис. 2. Схема алгоритма шейкер - сортировки(1).

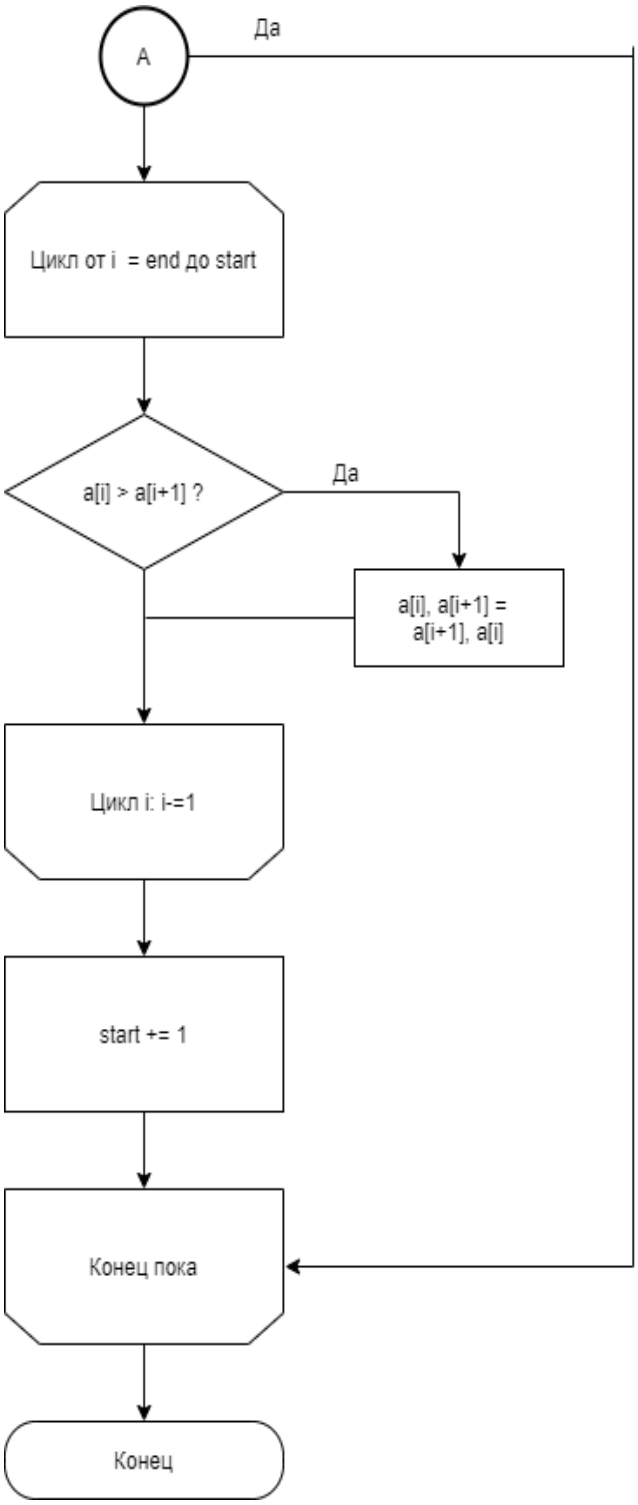

Рис. 3. Схема алгоритма шейкер - сортировки(2).

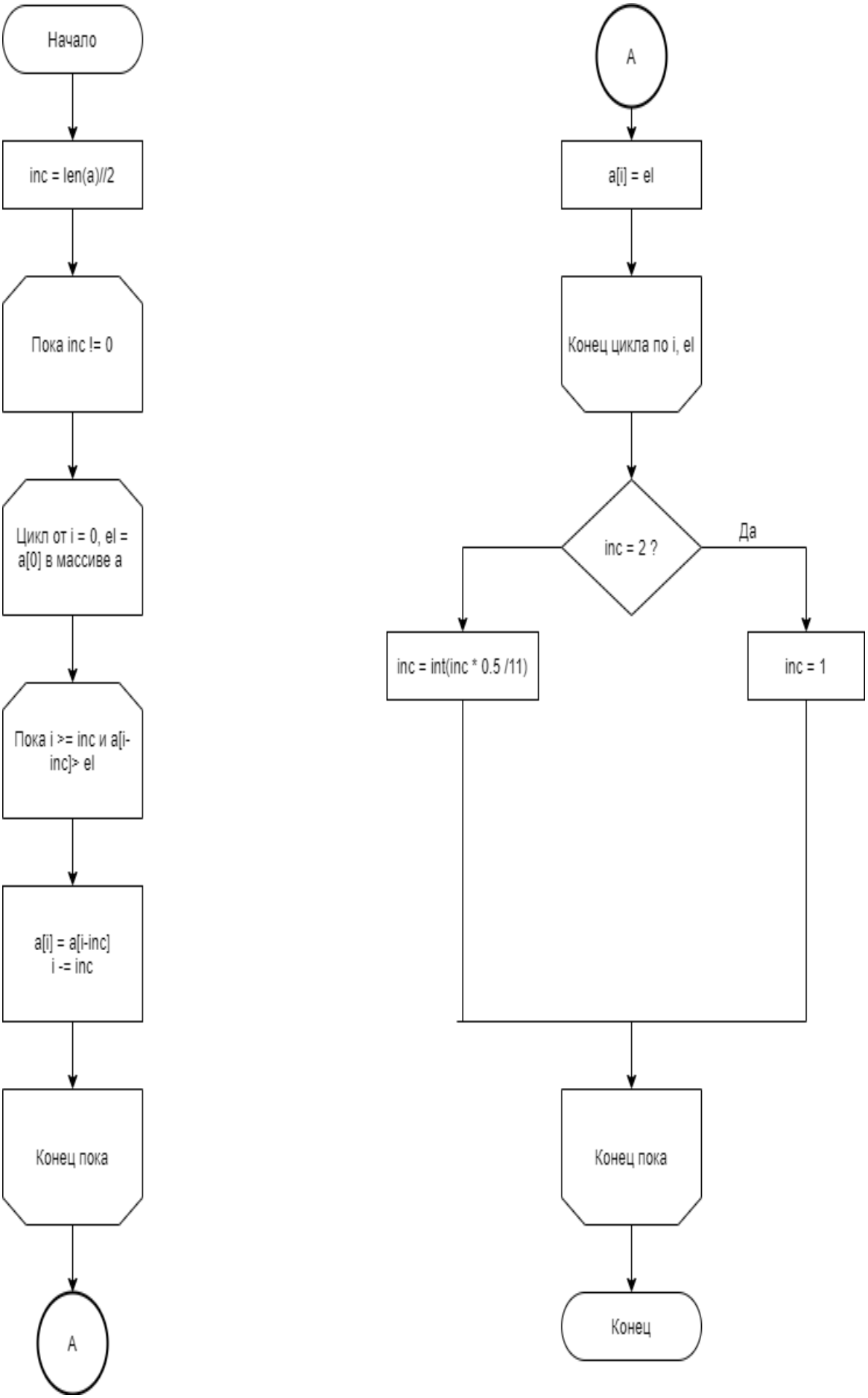

Рис. 4. Схема алгоритма сортировки Шелла (2).

3. Технологическая часть

3.1. Требования к программному обеспечению

Программное обеспечение должно обеспечивать замер процессорного времени выполнения каждого алгоритма. Проводятся замеры для случайно генерируемых массивов размерности до 10000.

3.2. Средства реализации

Python[1] — это высокоуровневый язык программирования, который используется в различных сферах IT, таких как машинное обучение, разработка приложений, web, парсинг и другие. С ним легко работать, что сокращает время разработки. Написанный в удобочитаемом формате, Python делает процесс разработки программного обеспечения быстрым, удобным и максимально упрощенным. По сравнению с другими языками, Python в 5-10 раз быстрее по времени разработки, однако медленный при выполнении программ. Он обеспечивает расширенные возможности управления процессами и объектно-ориентированный дизайн, помогая как в скорости, так и в производительности.

3.3. Методы замера времени в программе

3.3.1. Время

Сушествует несколько способов измерения процессорного времени исполнения программы. Помимо стандратного модуля *time* есть библиотека *timeit* [2]. Этот модуль предоставляет простой способ найти время выполнения маленьких битов кода Python. *timeit* запускает фрагмент кода миллионы раз(значение по умолчанию - 1000000), так что получаем наиболее статистически значимое измерение времени выполнения кода.

3.3.2. Улучшение точности замеров времени

Чтобы получить более точные результаты, каждый тест запускается несколько раз, все полученные значения времени(в тиках) суммируются и делятся на количество запусков кода. Таким образом, получаем среднее время выполнения кода.

3.4. Листинг кода

Листинг 1. Сортировка пузырьком

```
    def bubbleSort(data):
      N = len(data)
      for i in range(N-1):
        for j in range(N-i-1):
          if data[j] > data[j+1]:
            data[j], data[j+1] = data[j+1], data[j]
```

Листинг 2. Шейкер-сортировка

```
def cocktailSort(array):
  length = len(array)
  swapped = True
  start_index = 0
  end_index = length - 1

  while (swapped == True):
    swapped = False
    # prohod sleva naprovo
    for i in range(start_index, end_index):
      if (array[i] > array[i + 1]) :
      # obmen elementov
        array[i], array[i + 1] = array[i + 1], array[i]
        swapped = True

    # yesli ne bylo obmenov break
    if (not(swapped)):
      break

    swapped = False
    end_index = end_index - 1

    # prohod sprava nalevo
    for i in range(end_index - 1, start_index - 1, -1):
      if (array[i] > array[i + 1]):
      # obmen
        array[i], array[i + 1] = array[i + 1], array[i]
        swapped = True

    start_index = start_index + 1
```

Листинг 3. Сортировка Шелла

```
  def shellSort(data):
    inc = len(data) // 2
    while inc:
      for i, el in enumerate(data):
        while i >= inc and data[i - inc] > el:
```

```
            data[i] = data[i - inc]
            i -= inc
        data[i] = el
    inc = 1 if inc == 2 else int(inc * 5.0 / 11)
```

4. Экспериментальная часть

Проведём тестирование сравним алгоритмы по времени работы.

4.1. Примеры работ

Тестирование фукнциональности алгоритмов сортировки выполняется методом черного ящика для наглядного показа правильности работы алгоритмов сортировки.

Ниже приведены примеры работ.

Листинг 4. Пример работы алгоритмов на сортированный массив

```
Source array:    [-187, -88, 26, 52, 110, 164]
Expected array:  [-187, -88, 26, 52, 110, 164]
Bubble Sort:     [-187, -88, 26, 52, 110, 164]
Shell Sort:      [-187, -88, 26, 52, 110, 164]
Shaker Sort:     [-187, -88, 26, 52, 110, 164]
```

Листинг 5. Пример работы алгоритмов на случайный массив

```
Source array:      [-136, -18, 146, 116, -64, -91]
Expected array:    [-136, -91, -64, -18, 116, 146]
Bubble Sort:       [-136, -91, -64, -18, 116, 146]
Shell Sort:        [-136, -91, -64, -18, 116, 146]
Shaker Sort:       [-136, -91, -64, -18, 116, 146]
```

Листинг 6. Пример работы алгоритмов на обратно отсортированный массив

```
Source array:     [177, 133, 82, 50, 46, -189]
Expected array:   [-189, 46, 50, 82, 133, 177]
Bubble Sort:      [-189, 46, 50, 82, 133, 177]
Shell Sort:       [-189, 46, 50, 82, 133, 177]
Shaker Sort:      [-189, 46, 50, 82, 133, 177]
```

Листинг 7. Пример работы алгоритмов массив с одинаковыми элементами

```
Source array:   [5, 5, 5, 5, 5, 5]
Expected array: [5, 5, 5, 5, 5, 5]
Bubble Sort:    [5, 5, 5, 5, 5, 5]
Shell Sort:     [5, 5, 5, 5, 5, 5]
Shaker Sort:    [5, 5, 5, 5, 5, 5]
```

Все тесты прошли успешно.

4.2. Замеры времени

Ниже в таблице 1 представлены результаты замера времени для отсортированного массива, в таблице 2 - для обратно отсортированного массива и в таблице 3 - для случайно сгенерированного массива. Процессорное время измерено в тиках.

Таблица 1. Результаты замера времени для отсортированного массива.

Алгоритм	**10**	**100**	**300**	**500**	**700**	**1000**
Сортировка пузырьком	1084	77603	433224	1199938	2562328	5016702
Шейкер-сортировка	242	2137	2854	7722	7089	12066
Сортировка Шелла	842	8437	34295	57884	95701	149729

Таблица 2. Результаты замера времени для обратно отсортированного массива.

Алгоритм	**10**	**100**	**300**	**500**	**700**	**1000**
Сортировка пузырьком	2215	56473	477015	1386094	2862636	5539335
Шейкер-сортировка	480	11633	107924	362706	635177	1343340
Сортировка Шелла	907	7352	35593	69416	114963	137485

Таблица 3. Результаты замера времени для случайно сгенерированного массива.

Алгоритм	**10**	**100**	**300**	**500**	**700**	**1000**
Сортировка пузырьком	2145	71631	500447	1310741	2577064	5378399
Шейкер-сортировка	300	7308	86722	262657	460562	942419
Сортировка Шелла	906	8068	40303	121758	128520	142852

4.3. Выводы

Из результатов замер времени алгоритмов сортировки можно сделать следующие выводы:

1) сортировка пузырьком работет медленее, чем сортировка Шелла и Шейкер-сортировка;

2) у алгоритма сортировки Шелла при обратно отсортированном массиве самые лучшие результаты по времени выполнения;

3) Шейкер-сортировка выгодна в тех случаях, когда элементы уже стоят почти в правильном порядке, - но это случается редко на практике.[3]

4.4. Графики

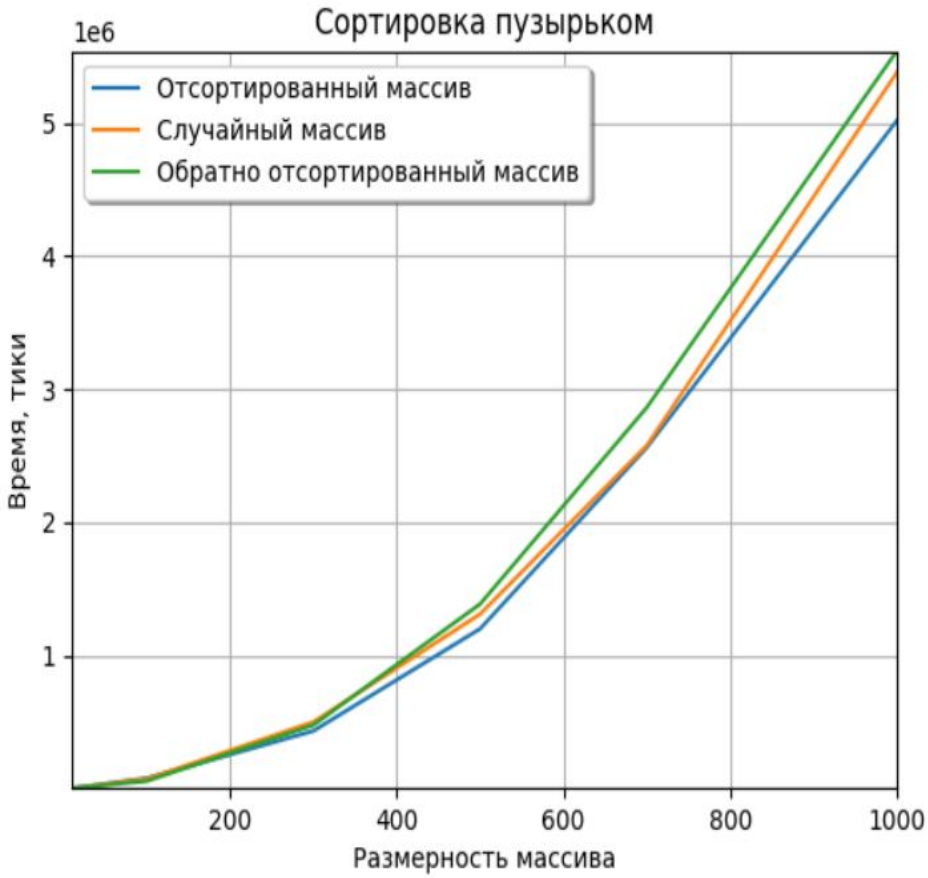

Рис. 5. График зависимости времени алгоритма сортировки пузырька.

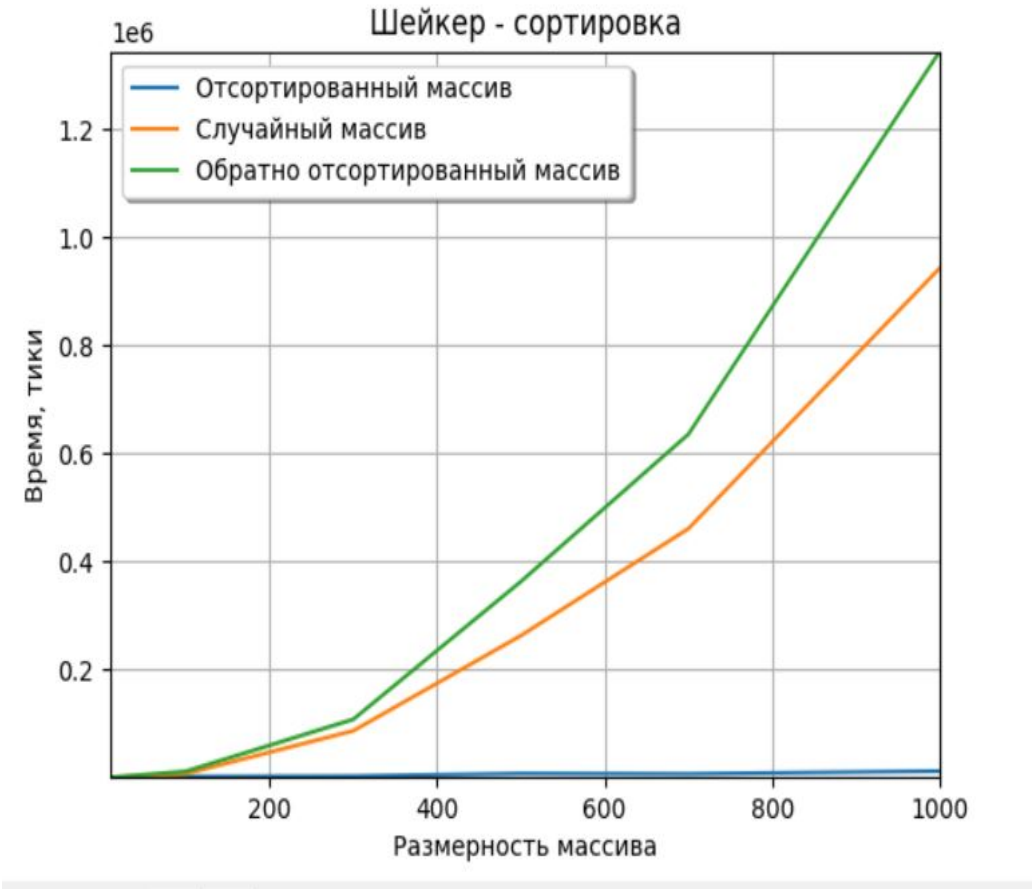

Рис. 6. График зависимости времени алгоритма Шейкер-сортировки.

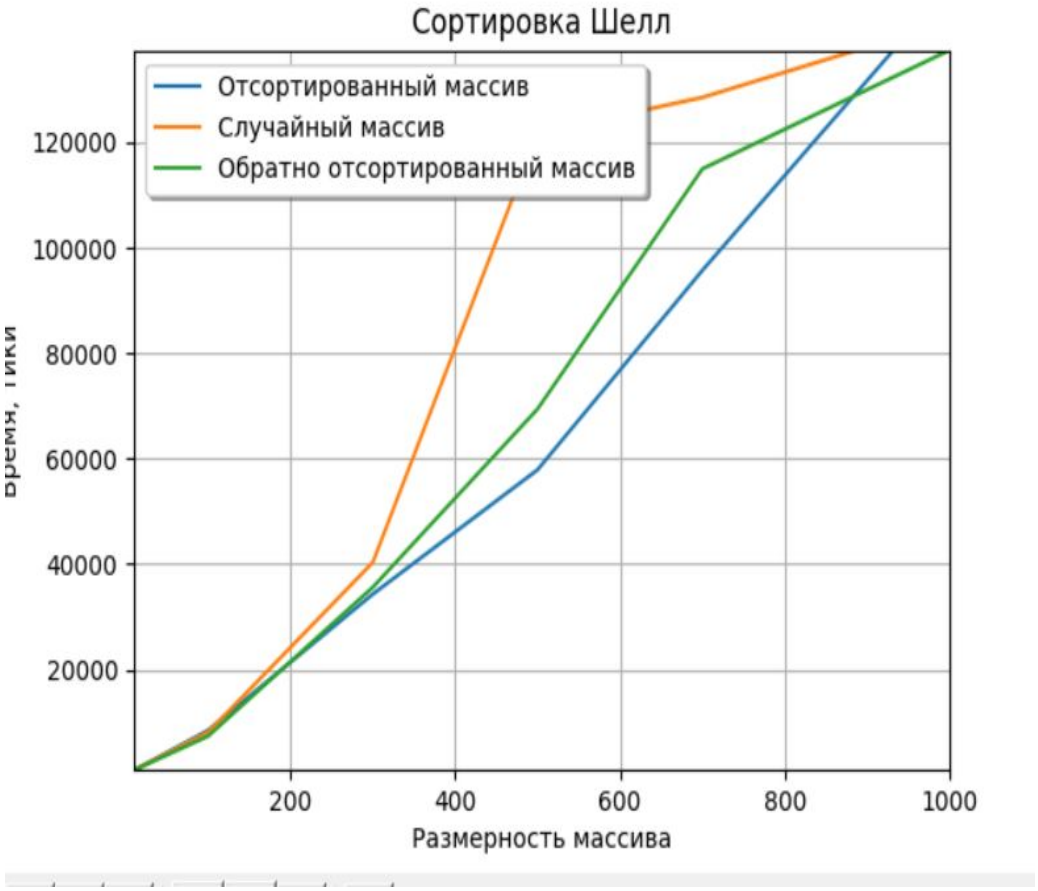

Рис. 7. График зависимости времени алгоритма сортировки Шелла.

5. Заключение

В ходе данной работы было выполнено:

- реализованы три алгоритма сортировки - сортировка пузырьком, Шейкер-сортировка и сортировка Шелла;
- теоретически вычислены эффективности алгоритмов сортировки;
- проведены замеры процессорного времени работы алгоритмов сортировки;

Были сделаны следующие выводы:

1) сортировка пузырьком работет медленее, чем сортировка Шелла и Шейкер-сортировка;
2) у алгоритма сортировки Шелла при обратно отсортированном массиве самые лучшие результаты по времени выполнения;
3) Шейкер-сортировка выгодна в тех случаях, когда элементы уже стоят почти в правильном порядке, - но это случается редко на практике. [3]

Из рассмотренных трех алгоритмов самым лучшим алгоритмом сортировки является сортировка Шелла.
Цели данной лабораторной работы достигнуты.

Список литературы

[1] Python:что нужно знать. [ЭЛ. РЕСУРС]
Режим доступа: https://skillbox.ru/media/code/story_buzunov/
(Дата обращения: 10.11.2020)

[2] Timeit в Python с примерами. [ЭЛ. РЕСУРС]
Режим доступа: http://espressocode.top/timeit-python-examples/
(Дата обращения: 10.11.2020)

[3] Никлаус Вирт Алгоритмы и структуры данных. Новая версия для Обертона/Пер. с англ. Ткачев Ф.В. – М.:ДМК Пресс, 2016–272 с.:ил.
(Дата обращения: 10.11.20)

[4] Шейкерная сортировка [ЭЛ. РЕСУРС]
Режим доступа: https://purecodecpp.com/archives/1895
(Дата обращения: 10.11.20)

[5] Сортировки вставками [ЭЛ.РЕСУРС]
Режим доступа: https://habr.com/ru/post/415935/
(Дата обращения: 10.11.20)

[6] Пузырьковая сортировкаи все-все-все [ЭЛ.РЕСУРС]
Режим доступа: https://habr.com/ru/post/204600/
(Дата обращения: 10.11.20)