EcoLab1

Создание компонента, реализующего алгоритм **«Timsort»**

Выполнил

Семенов Никита

21ПИ-2

Нижний Новгород

2024

Оглавление

[Описание алгоритма 3](#_Toc160356640)

[Псевдокод 4](#_Toc160356641)

[Оценка сложности 5](#_Toc160356642)

[Общий анализ 5](#_Toc160356643)

[Лучший случай 5](#_Toc160356644)

[Средний и худший случай 5](#_Toc160356645)

[Выводы 5](#_Toc160356646)

[Результаты тестирования 6](#_Toc160356647)

[Случайные числа типа **int** 6](#_Toc160356648)

[Случайные числа типа **float** 7](#_Toc160356649)

[Случайные числа типа **double** 8](#_Toc160356650)

[Выводы 8](#_Toc160356651)

# Описание алгоритма

**Timsort** – это алгоритм сортировки, объединяющий в себе идеи сортировок слиянием и вставками. Он был разработан ***Тимом Петерсом*** для использования в языке программирования Python и широко применяется в различных языках программирования и библиотеках.

Среднее время работы **Timsort** также составляет ***O(n\*log(n))***, что делает его асимптотически оптимальным для алгоритма, основанного на сравнении. Хотя в худшем случае его время работы может быть ***O(n\*log(n))***, на практике Timsort проявляет высокую производительность благодаря использованию эффективных оптимизаций.

Принцип работы **Timsort** можно описать как комбинацию двух подходов:

* Сначала массив разбивается на подмассивы минимальной длины, которые затем сортируются с использованием сортировки вставками. Этот этап позволяет эффективно обрабатывать части массива, где элементов немного или массив уже частично отсортирован.
* Затем отсортированные подмассивы объединяются с использованием сортировки слиянием. Этот этап гарантирует стабильность сортировки и эффективно обрабатывает случаи, когда в массиве есть большие участки упорядоченных данных.

Подобно быстрой сортировке, **Timsort** работает по принципу **«разделяй и властвуй»**, что позволяет ему эффективно справляться с различными типами входных данных.

# Псевдокод

void Timsort (array[size], run)

    for (i = 0; i < size; i += run)

        insertionSort(array[size], i, min((i + run - 1), (size - 1)));

    for (i = run; i < size; i = 2 \* i)

        for (left; left < size; left += 2 \* i)

            mid = left + i - 1;

            right = min((left + 2 \* i - 1), (size - 1));

            if (mid < right)

                merge(array[size], left, mid, right);

# Оценка сложности

## Общий анализ

**Timsort** имеет несколько этапов, включающих определение границ блоков данных, сортировку этих блоков и их объединение для получения окончательного отсортированного массива. При этом используется комбинация различных техник сортировки, чтобы достичь оптимальной производительности.

Временная сложность алгоритма **timsort** зависит от нескольких факторов, таких как размер входного массива, его степени уже отсортированности и наличия повторяющихся значений. Худший случай возникает, когда входной массив отсортирован в обратном порядке или содержит большое количество повторяющихся элементов.

## Лучший случай

В лучшем случае, когда входной массив уже отсортирован, временная сложность **timsort** составляет **O(n)**, где **n** — количество элементов в массиве. Это объясняется тем, что алгоритм использует сортировку вставками для блоков данных, которые уже находятся в нужном порядке.

## Средний и худший случай

Средний случай и худший случай временной сложности **timsort** составляют ***O(n\*log(n))***, где n — количество элементов в массиве. Этот результат достигается благодаря сортировке слиянием, которая эффективна на больших и неотсортированных массивах данных.

## Выводы

Алгоритм **timsort** обеспечивает хорошую производительность и стабильность в большинстве случаев. Однако, в некоторых ситуациях, реализация **timsort** может потребовать дополнительной памяти для хранения временных переменных и структур данных. Это следует учитывать при выборе алгоритма сортировки для конкретной задачи.

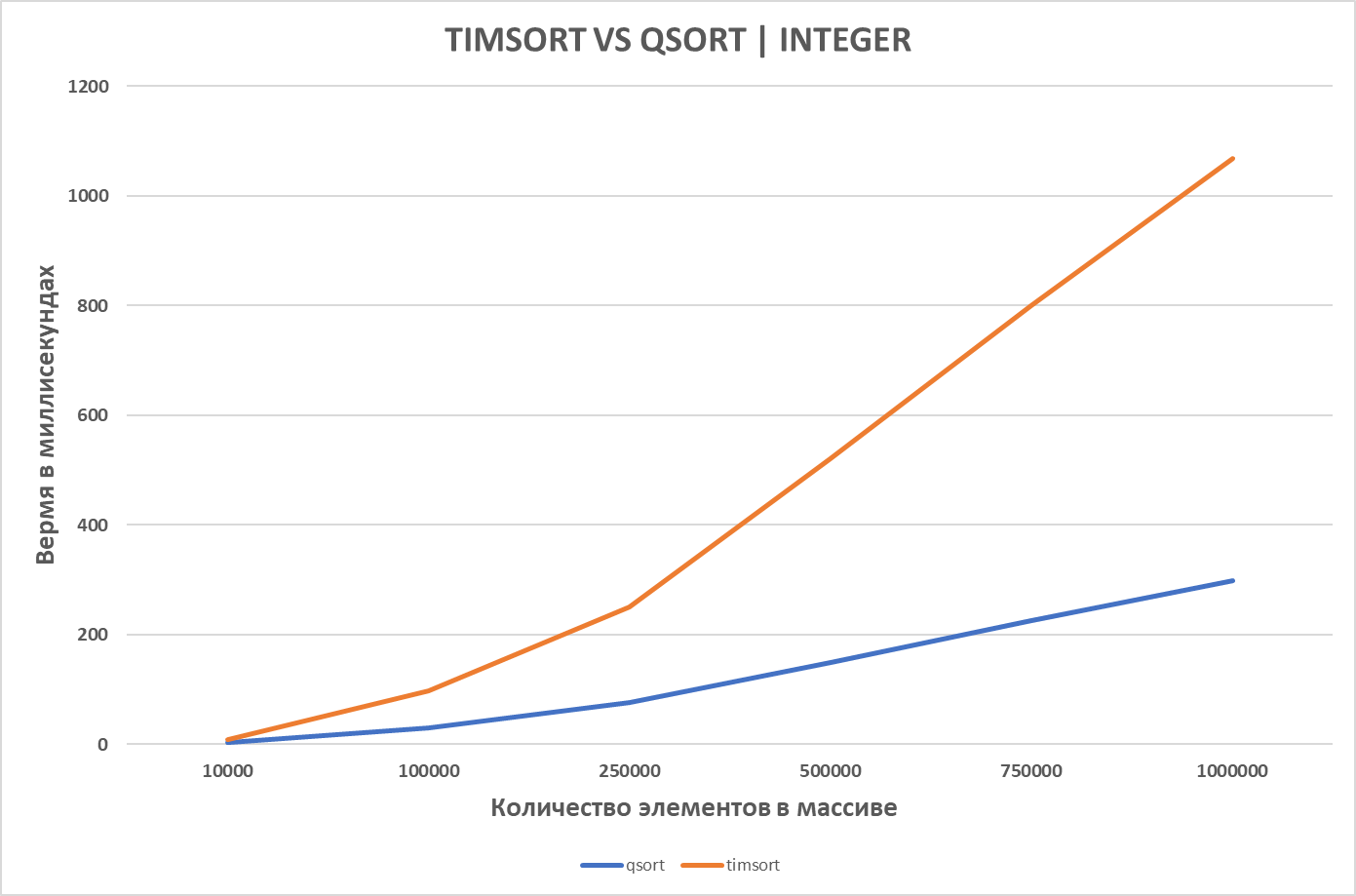
# Результаты тестирования

Было проведено тестирование алгоритма с различными входными данными: случайные числа типа **int**, случайные числа типа **float**, случайные числа типа **double**. Размеры сортируемых массивов варьировались от 10.000 элементов до 1.000.000 элементов.

Результаты тестирования приведены ниже:

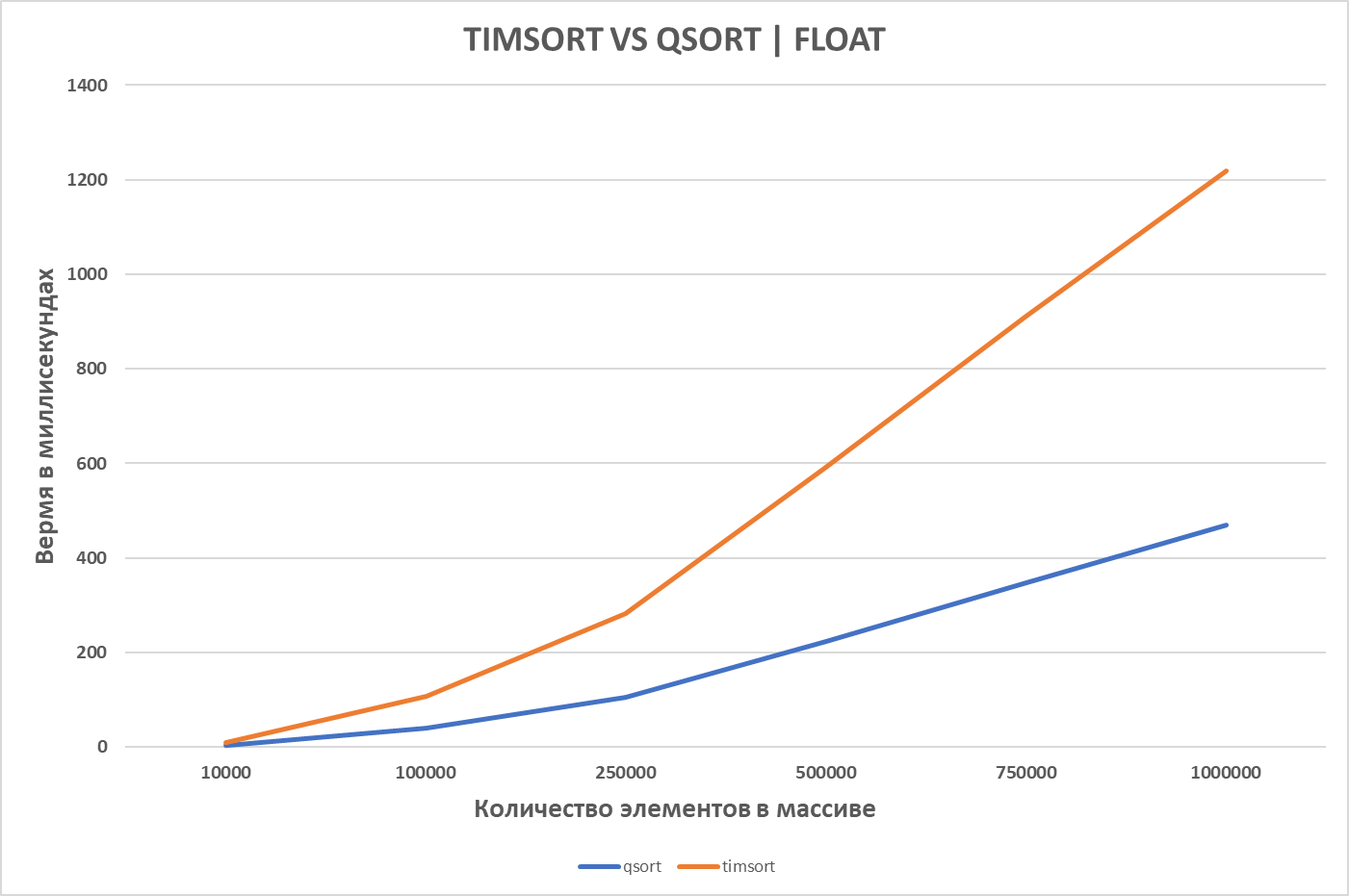
## Случайные числа типа **int**

|  |  |  |
| --- | --- | --- |
| Время работы | | |
| **Алгоритм** | **Количество элементов массива** | **Время, миллисекунды** |
| qsort | 10000 | 3 |
| 100000 | 30 |
| 250000 | 77 |
| 500000 | 150 |
| 750000 | 226 |
| 1000000 | 298 |
| timsort | 10000 | 9 |
| 100000 | 98 |
| 250000 | 250 |
| 500000 | 521 |
| 750000 | 800 |
| 1000000 | 1068 |



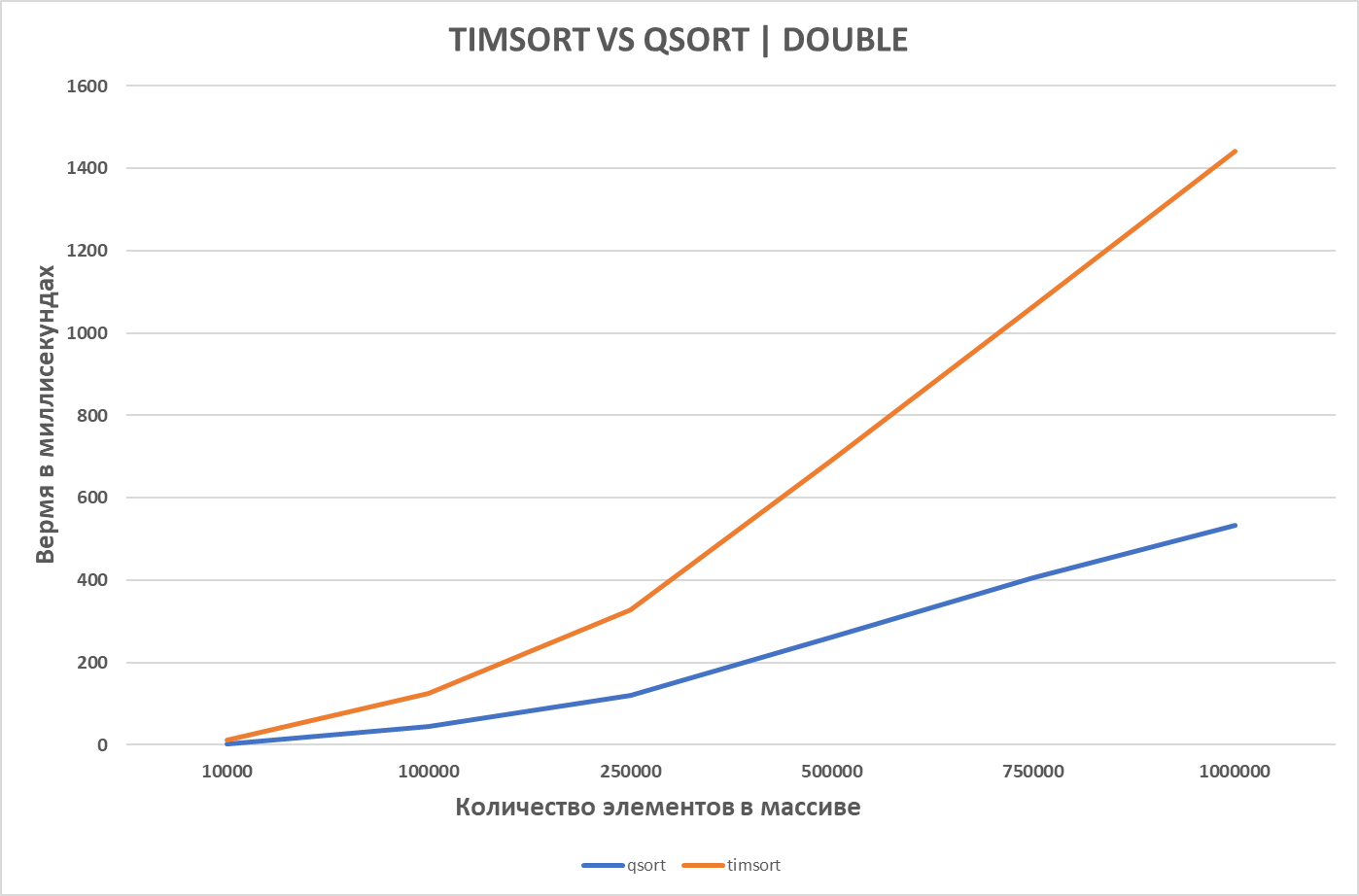
## Случайные числа типа **float**

|  |  |  |
| --- | --- | --- |
| Время работы | | |
| **Алгоритм** | **Количество элементов массива** | **Время, миллисекунды** |
| qsort | 10000 | 3 |
| 100000 | 41 |
| 250000 | 105 |
| 500000 | 224 |
| 750000 | 348 |
| 1000000 | 469 |
| timsort | 10000 | 9 |
| 100000 | 107 |
| 250000 | 282 |
| 500000 | 592 |
| 750000 | 912 |
| 1000000 | 1219 |



## Случайные числа типа **double**

|  |  |  |
| --- | --- | --- |
| Время работы | | |
| Алгоритм | Количество элементов массива | Время, миллисекунды |
| qsort | 10000 | 3 |
| 100000 | 45 |
| 250000 | 121 |
| 500000 | 261 |
| 750000 | 406 |
| 1000000 | 534 |
| timsort | 10000 | 11 |
| 100000 | 126 |
| 250000 | 328 |
| 500000 | 691 |
| 750000 | 1065 |
| 1000000 | 1442 |



## Выводы

Из приведенных выше данных можно сделать вывод, что время работы сортировки напрямую зависит от входных данных. В целом, моя реализация алгоритма Timsort уступает в скорости стандартной функции сортировки qsort из stdlib примерно в 3 – 4 раза, однако все ровно показывает хорошие результаты на всех типах входных данных. Худший результат обе функции показали на входных данных типа double, так как этот тип данных занимает наибольшее количество байтов памяти среди остальных тестируемых типов.