

## **Эффективность встроенных методов сортировки Python: `sort()` и `sorted()` против классических алгоритмов**

*Мандронов Антон Андреевич*

*Приамурский государственный университет имени Шолом-Алейхема*

*Студент*

### **Аннотация**

В статье проводится сравнительный анализ производительности различных алгоритмов сортировки, включая QuickSort, MergeSort, InsertionSort и BubbleSort, в сравнении с встроенными методами Python (`sorted()` и `list.sort()`). Экспериментальное исследование выполнено на массивах данных разного размера (от 1000 до 50000 элементов), что позволило оценить временную эффективность каждого алгоритма. Результаты демонстрируют значительное преимущество встроенных методов Python, использующих гибридный алгоритм Timsort, перед классическими алгоритмами сортировки. Особое внимание уделено практическим рекомендациям по выбору оптимального метода сортировки в зависимости от размера данных и требований к производительности.

**Ключевые слова:** алгоритмы сортировки, QuickSort, MergeSort, InsertionSort, BubbleSort, Timsort, производительность, временная сложность, Python, `sorted()`, `list.sort()`, сравнительный анализ.

## **The effectiveness of Python's built-in sorting methods: `sort()` and `sorted()` against classical algorithms**

*Mandronov Anton Andreevich*

*Sholom-Aleichem Priamursky State University*

*Student*

### **Abstract**

The article provides a comparative analysis of the performance of various sorting algorithms, including QuickSort, MergeSort, InsertionSort and BubbleSort, in comparison with Python's built-in methods (`sorted()` and `list.sort()`). The experimental study was performed on data arrays of different sizes (from 1000 to 50,000 elements), which made it possible to estimate the time efficiency of each algorithm. The results demonstrate a significant advantage of Python's built-in methods using the hybrid Timsort algorithm over classical sorting algorithms. Special attention is paid to practical recommendations for choosing the optimal sorting method depending on the size of the data and performance requirements.

**Keywords:** sorting algorithms, QuickSort, MergeSort, InsertionSort, BubbleSort, Timsort, performance, time complexity, Python, `sorted()`, `list.sort()`, comparative analysis.

## **1 Введение**

### **1.1 Актуальность**

Сортировка данных — одна из самых распространённых операций в программировании, влияющая на производительность алгоритмов в таких областях, как обработка больших данных (эффективная сортировка ускоряет поиск и анализ), машинное обучение (предварительная сортировка может улучшить работу алгоритмов), базы данных (индексация и запросы часто опираются на сортировку), веб-разработка (сортировка списков, таблиц и результатов поиска). Python, как один из самых популярных языков программирования, предоставляет встроенные методы сортировки (`sort()` и `sorted()`), но многие разработчики, особенно начинающие, не до конца понимают их эффективность по сравнению с классическими алгоритмами.

### **1.2 Обзор исследований**

В исследовании В.С. Дейч и С.А. Зырянова проводится системный анализ популярных алгоритмов сортировки с акцентом на практические аспекты их применения. Разрабатываются рекомендации по выбору оптимального метода в зависимости от типа данных и требований к производительности [1]. В работе А.Г. Дупленко представлен подробный сравнительный анализ эффективности различных алгоритмов обработки массивов. Исследование включает оценку временной сложности методов и их поведения при работе с наборами данных разного объема [2]. М.В. Ульянов и соавторы рассматривают инновационные комбинированные подходы к сортировке, сочетающие преимущества итерационных и рекурсивных методов. Предложенные решения демонстрируют особую эффективность при обработке сложных структур данных [3]. В публикации Н.И. Дикарева, Б.М. Шабанова и А.С. Шмелёва исследуются оптимизированные алгоритмы для современных векторных процессоров. Основное внимание уделено техникам параллельной обработки данных и методам повышения производительности вычислительных систем [4].

### **1.3 Цель исследования**

Целью исследования является сравнительный анализ временной эффективности классических и встроенных алгоритмов сортировки при обработке случайных данных в среде Python.

## **2 Материалы и методы**

Исследование проводилось в среде Google Colab с использованием стандартной среды выполнения Python 3.11.12. Для оценки производительности использовались массивы случайных целых чисел пяти различных размеров: 1000, 5000, 10000, 20000 и 50000 элементов. Были протестированы шесть алгоритмов сортировки: встроенные методы Python `sorted()` и `list.sort()`, а также реализованные вручную QuickSort, MergeSort, InsertionSort и BubbleSort. Все алгоритмы реализованы без дополнительных оптимизаций для обеспечения чистоты сравнения.

### 3 Результаты и обсуждения

Для формирования тестовых массивов использовался модуль `random` стандартной библиотеки Python, которая генерировала равномерно распределённые целочисленные значения в диапазоне от 0 до 100 000. Точные измерения временных характеристик выполнялись при помощи высокоточного модуля `timeit`, специально разработанного для профилирования выполнения кода в Python (рис. 1).

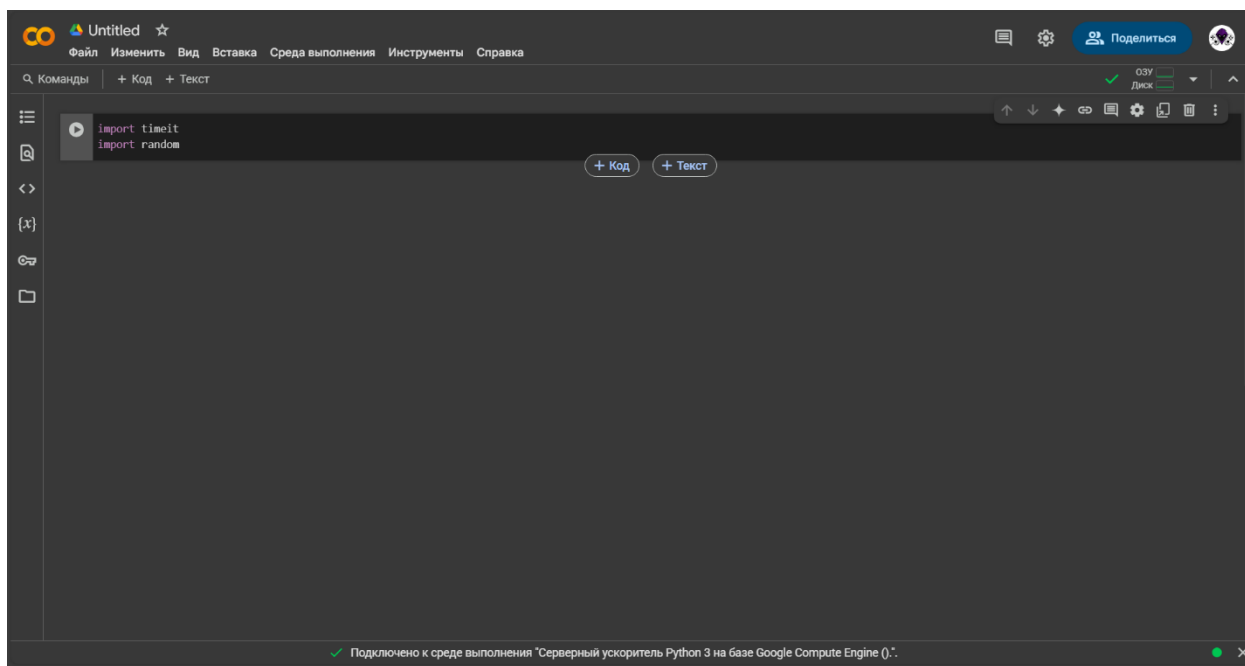


Рис 1. Подключение модулей

Для сравнения были реализованы четыре классических алгоритма сортировки:

1. **BubbleSort (пузырьковая сортировка)** (рис. 2) — самый простой, но и самый медленный из рассматриваемых алгоритмов. Последовательно сравнивает соседние элементы и меняет их местами, если они стоят в неправильном порядке. Процесс повторяется, пока массив не будет полностью отсортирован. Сложность —  $O(n^2)$  во всех случаях, кроме уже отсортированного массива ( $O(n)$ ). В исследовании используется оптимизированная версия, прекращающая работу при отсутствии перестановок за проход.

```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        swapped = False  
        for j in range(0, n-i-1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]  
                swapped = True  
        if not swapped:  
            break  
    return arr
```

Рис 2. Реализация сортировки пузырьком

2. **InsertionSort (сортировка вставками)** (рис. 3) — этот алгоритм строит отсортированный массив по одному элементу, вставляя каждый новый элемент в правильную позицию среди уже упорядоченных. На небольших или почти отсортированных массивах работает достаточно быстро ( $O(n)$  в лучшем случае), но в общем случае имеет квадратичную сложность  $O(n^2)$ , что делает его неэффективным для больших данных.

```
def insertion_sort(arr):  
    for i in range(1, len(arr)):  
        key = arr[i]  
        j = i-1  
        while j >= 0 and key < arr[j]:  
            arr[j+1] = arr[j]  
            j -= 1  
        arr[j+1] = key  
    return arr
```

Рис 3. Реализация сортировки вставками

3. **QuickSort (быстрая сортировка)** (рис. 4) — этот алгоритм работает по принципу "разделяй и властвуй". Он выбирает опорный элемент, разделяет массив на две части (элементы меньше опорного и больше опорного) и рекурсивно сортирует подмассивы. В среднем случае QuickSort имеет сложность  $O(n \cdot \log n)$ , но в худшем (при неудачном выборе опорного элемента) может деградировать до  $O(n^2)$ . Реализация в исследовании использует центральный элемент в качестве опорного, но не оптимизирована для избежания худшего случая.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

Рис 4. Реализация быстрой сортировки

**4. MergeSort (сортировка слиянием)** (рис. 5) — алгоритм также использует стратегию "разделяй и властвуй", рекурсивно разбивая массив на подмассивы до единичных элементов, а затем сливая их в упорядоченном порядке. Главное преимущество MergeSort — гарантированная сложность  $O(n \cdot \log n)$  в любом случае. Однако он требует  $O(n)$  дополнительной памяти для хранения промежуточных массивов при слиянии.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Рис 5. Реализация сортировки слиянием

В языке Python существуют встроенные сортировки — `list.sort()` и `sorted()`. Функция `sorted()` возвращает новый отсортированный список, оставляя исходную последовательность неизменной. Она работает с любыми итерируемыми объектами и поддерживает дополнительные параметры, такие как `key` (значение, по которому будет сортироваться объект) и `reverse` (сортировка в обратном порядке). Метод `list.sort()` выполняет сортировку списка на месте, не создавая копию. Оба метода используют алгоритм Timsort — гибридный метод, сочетающий MergeSort и InsertionSort. Timsort особенно эффективен на частично упорядоченных данных, обеспечивая временную сложность  $O(n \cdot \log n)$  в худшем случае и  $O(n)$  в лучшем.

Для проведения тестов были созданы две функции (рис. 6): одна генерирует тестовые данные, другая проводит измерение времени

выполнения каждого алгоритма сортировки с данными разных размеров: 1000, 5000, 10000, 20000 и 50000 элементов.

```
def generate_random_data(size):
    return [random.randint(0, 100000) for _ in range(size)]

def measure_sorting_time():
    sizes = [1000, 5000, 10000, 20000, 50000]
    algorithms = {
        'sorted()': lambda data: sorted(data.copy()),
        'list.sort()': lambda data: data.copy().sort(),
        'QuickSort': quicksort,
        'MergeSort': merge_sort,
        'InsertionSort': insertion_sort,
        'BubbleSort': bubble_sort
    }

    print(f"{'Алгоритм':<12} | {'Размер':<8} | {'Время (мс)':<12}")
    print("-" * 40)

    for size in sizes:
        data = generate_random_data(size)
        for name, algorithm in algorithms.items():
            test_data = data.copy()

            timer = timeit.Timer(
                lambda: algorithm(test_data.copy()) if name != 'list.sort()'
                else lambda: test_data.copy().sort()
            )

            time_taken = timer.timeit(number=10) * 1000 / 10

            print(f"{name:<12} | {size:<8} | {time_taken:>10.2f}")

    print("Тестирование времени выполнения алгоритмов сортировки...\n")
    measure_sorting_time()
    print("\nТестирование завершено.")
```

Рис 6. Генерация данных и измерение времени сортировок

После запуска кода сочетанием клавиш Ctrl+Enter получаем таблицу времени выполнения каждой сортировки при определённом размере массива (табл. 1).

Таблица 1. Результаты выполнения алгоритмов

| Алгоритм      | Размер | Время(мс) |
|---------------|--------|-----------|
| sorted()      | 1000   | 0.18      |
| list.sort()   | 1000   | 0.00      |
| QuickSort     | 1000   | 3.23      |
| MergeSort     | 1000   | 3.09      |
| InsertionSort | 1000   | 41.36     |
| BubbleSort    | 1000   | 69.35     |
| sorted()      | 5000   | 0.93      |
| list.sort()   | 5000   | 0.00      |
| QuickSort     | 5000   | 11.70     |
| MergeSort     | 5000   | 12.28     |
| InsertionSort | 5000   | 605.13    |
| BubbleSort    | 5000   | 1457.98   |
| sorted()      | 10000  | 2.02      |
| list.sort()   | 10000  | 0.00      |
| QuickSort     | 10000  | 22.17     |
| MergeSort     | 10000  | 25.57     |

|               |       |           |
|---------------|-------|-----------|
| InsertionSort | 10000 | 2618.34   |
| BubbleSort    | 10000 | 6053.73   |
| sorted()      | 20000 | 4.12      |
| list.sort()   | 20000 | 0.00      |
| QuickSort     | 20000 | 45.75     |
| MergeSort     | 20000 | 55.42     |
| InsertionSort | 20000 | 10118.62  |
| BubbleSort    | 20000 | 24515.32  |
| sorted()      | 50000 | 12.47     |
| list.sort()   | 50000 | 0.00      |
| QuickSort     | 50000 | 124.67    |
| MergeSort     | 50000 | 265.86    |
| InsertionSort | 50000 | 68790.32  |
| BubbleSort    | 50000 | 167301.09 |

#### 4 Выводы

Результаты показали, что встроенные алгоритмы Python (sorted() и list.sort()) значительно превосходят по скорости классические алгоритмы, такие как QuickSort, MergeSort, InsertionSort и BubbleSort. Хотя функция sorted() работает медленнее метода list.sort() из-за создания копии данных, она работает значительно быстрее классических алгоритмов, что особенно хорошо видно на больших данных. В реальных задачах стоит всегда использовать sorted() или list.sort(), а классические алгоритмы изучать для понимания основ, но не для практического применения в Python.

#### Библиографический список

1. Дейч В. С., Зырянова С. А. Алгоритмы сортировки и их применение //Образование. Транспорт. Инновации. Строительство. 2022. С. 557-560.
2. Дупленко А. Г. Сравнительный анализ алгоритмов сортировки данных в массивах //Молодой ученый. 2013. №. 8. С. 50-53.
3. Ульянов М. В. и др. Комбинированные итерационнорекурсивные алгоритмы сортировки //Автоматизация и современные технологии. 2008. №. 3. С. 17-26.
4. Дикарев Н. И., Шабанов Б. М., Шмелёв А. С. Быстрые алгоритмы сортировки для векторного потокового процессора //Суперкомпьютерные технологии (СКТ-2018). 2018. С. 87-91.