

Сравнительный анализ алгоритмов сортировки: эффективность и применение

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлено комплексное исследование эффективности шести алгоритмов сортировки (пузырьком, вставками, выбором, быстрой сортировки, слиянием и кучей) на различных типах данных: случайных, частично упорядоченных и обратно упорядоченных. Экспериментальные результаты, полученные на массивах размером от 100 до 10 000 элементов, демонстрируют преимущества алгоритмов с временной сложностью $O(n \cdot \log n)$ (быстрая сортировка, сортировка слиянием и кучей) для больших объёмов данных, а также подчёркивают неэффективность квадратичных алгоритмов (пузырьковая, вставками, выбором) при масштабировании. Особое внимание уделено практическим рекомендациям по выбору алгоритма в зависимости от характеристик входных данных.

Ключевые слова: алгоритмы сортировки, временная сложность, быстрая сортировка, сортировка слиянием, сортировка кучей, сортировка пузырьком, сортировка вставками, сортировка выбором, производительность.

Comparative analysis of sorting algorithms: efficiency and application

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comprehensive study of the efficiency of six sorting algorithms (Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, and Heap Sort) on different types of data: random, partially ordered, and reverse-ordered. Experimental results obtained on arrays ranging from 100 to 10,000 elements demonstrate the advantages of algorithms with $O(n \cdot \log n)$ time complexity (Quick Sort, Merge Sort, and Heap Sort) for large datasets, while highlighting the inefficiency of quadratic algorithms (Bubble Sort, Insertion Sort, Selection Sort) when scaling. Special attention is given to practical recommendations for selecting an algorithm based on input data characteristics.

Keywords: sorting algorithms, time complexity, Quick Sort, Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, performance.

1 Введение

1.1 Актуальность

Алгоритмы сортировки являются фундаментальной частью компьютерных наук и находят применение в самых различных областях: от баз данных и поисковых систем до машинного обучения и анализа больших данных. С ростом объёмов информации и разнообразия вычислительных систем повышается потребность в оптимизации алгоритмов сортировки с учётом их временной и пространственной сложности, а также особенностей аппаратной реализации. Несмотря на существование множества алгоритмов (быстрая сортировка, сортировка слиянием и др.), выбор оптимального метода зависит от специфики данных, требований к скорости и доступных ресурсов. Поэтому сравнительный анализ их эффективности в различных сценариях остаётся важной задачей, позволяющей разработчикам принимать обоснованные решения при проектировании программных систем.

1.2 Обзор исследований

В статье А. Г. Дупленко проведен сравнительный анализ алгоритмов сортировки данных с оценкой их эффективности по времени выполнения и используемой памяти. Результаты исследования показывают зависимость применимости методов от размера и структуры массива [1]. Работа И. В. Заики и И. А. Тюшняковой содержит систематизированный обзор основных методов сортировки с анализом их теоретических основ и практических особенностей. Особый акцент сделан на сравнении сложности алгоритмов и их адаптивности к разным типам данных [2]. В публикации Д. В. Тимшиной и А. О. Максюты рассматривается применение алгоритмов сортировки для оптимизации задач информационной логистики. Предложены практические рекомендации по выбору методов для повышения эффективности логистических систем [3]. В статье М. А. Козлова и С. В. Скворцова рассматриваются современные подходы к параллельной сортировке данных с использованием языка Clojure. Особое внимание уделено вопросам масштабируемости и производительности алгоритмов в многопоточных средах [4].

1.3 Цель исследования

Цель исследования — провести комплексное сравнение эффективности алгоритмов сортировки на числовых данных с тремя ключевыми типами структуры: случайными, частично упорядоченными и полностью обратными последовательностями, чтобы выявить оптимальные сферы применения каждого метода.

2 Материалы и методы

В рамках исследования проведен сравнительный анализ шести алгоритмов сортировки: сортировки пузырьком (Bubble Sort), сортировки вставками (Insertion Sort), сортировки выбором (Selection Sort), быстрой сортировки (Quick Sort), сортировки слиянием (Merge Sort) и сортировки

кучей (Heap Sort). Основное внимание уделено их временной эффективности и применимости в зависимости от характеристик входных данных. Разработка и тестирование алгоритмов проводились на языке Python 3.12 с использованием интегрированной среды разработки PyCharm Community Edition 2024.1.

3 Результаты и обсуждения

В начале кода импортируем следующие библиотеки:

1. **Numpy** (для генерации тестовых данных)
2. **Timeit** (для точного измерения времени)
3. **Matplotlib** (для визуализации зависимости времени выполнения от размеров данных)
4. Функцию **deecopy** из библиотеки **copy** (для точного копирования данных)
5. Функции **heappop**, **heappush** из библиотеки **heapq** (для реализации одного из алгоритмов)

Поскольку **matplotlib** и **numpy** изначально не встроены в интерпретатор Python, необходимо их установить, открыв терминал сочетанием Alt+F12, вписав команду «*pip install numpy matplotlib*» и нажав Enter (рис. 1).

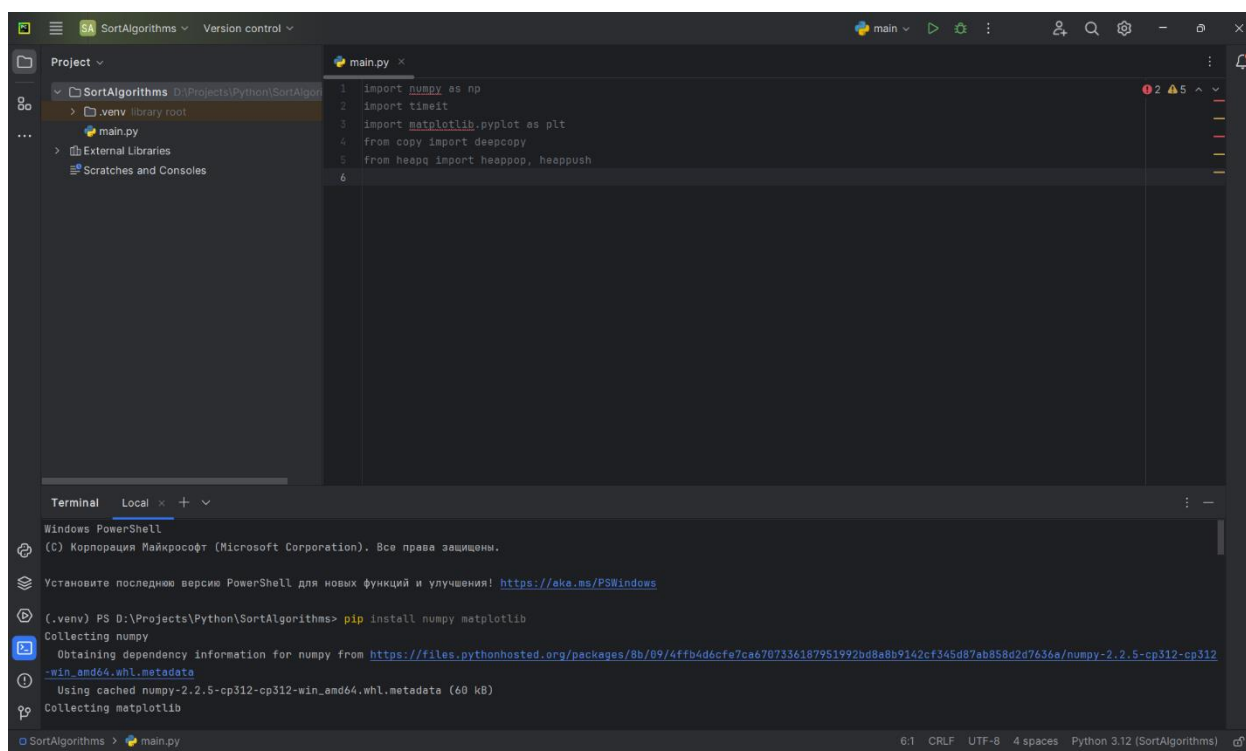


Рис 1. Установка библиотек

Теперь реализуем каждый тип сортировки:

1. Сортировка пузырьком (Bubble Sort) (рис. 2) проходит по массиву несколько раз, сравнивая соседние элементы и меняя их местами, если они стоят в неправильном порядке. Этот процесс повторяется, пока массив не

будет полностью отсортирован. Алгоритм прост в реализации, но неэффективен для больших данных, так как в худшем случае требует $O(n^2)$ операций.

```
7
8 def bubble_sort(arr):
9     n = len(arr)
10    for i in range(n):
11        for j in range(0, n-i-1):
12            if arr[j] > arr[j+1]:
13                arr[j], arr[j+1] = arr[j+1], arr[j]
14
15
```

Рис 2. Сортировка пузырьком

2. Сортировка вставками (Insertion Sort) (рис. 3) работает по принципу постепенного построения отсортированной части массива. На каждом шаге берётся очередной элемент и вставляется в нужное место среди уже упорядоченных элементов. Этот метод эффективен для небольших или почти отсортированных массивов ($O(n^2)$ в худшем случае, но $O(n)$ в лучшем).

```
15
16 def insertion_sort(arr):
17     for i in range(1, len(arr)):
18         key = arr[i]
19         j = i-1
20         while j >= 0 and key < arr[j]:
21             arr[j+1] = arr[j]
22             j -= 1
23         arr[j+1] = key
24
```

Рис 3. Сортировка вставками

3. Сортировка выбором (Selection Sort) (рис. 4) на каждом проходе находит минимальный (или максимальный) элемент в неотсортированной части массива и перемещает его в конец (или начало) отсортированной части. Как и предыдущие два алгоритма, он имеет квадратичную сложность $O(n^2)$, но требует меньше перестановок, чем пузырьковая сортировка.

```
25
26 def selection_sort(arr):
27     for i in range(len(arr)):
28         min_idx = i
29         for j in range(i+1, len(arr)):
30             if arr[j] < arr[min_idx]:
31                 min_idx = j
32         arr[i], arr[min_idx] = arr[min_idx], arr[i]
33
```

Рис 4. Сортировка выбором

4. Быстрая сортировка (Quick Sort) (рис. 5) использует стратегию «разделяй и властвуй»: выбирается опорный элемент (*pivot*), массив разбивается на две части — элементы меньше *pivot* и больше него, после чего процесс рекурсивно повторяется для каждой части. В среднем случае сложность составляет $O(n \cdot \log n)$, но в худшем (например, при неудачном выборе *pivot*) может деградировать до $O(n^2)$.

```
34
35 2 usages
36 def quick_sort(arr):
37     if len(arr) <= 1:
38         return arr
39     pivot = arr[len(arr)//2]
40     left = [x for x in arr if x < pivot]
41     middle = [x for x in arr if x == pivot]
42     right = [x for x in arr if x > pivot]
43     return quick_sort(left) + middle + quick_sort(right)
```

Рис 5. Быстрая сортировка

5. Сортировка слиянием (Merge Sort) (рис. 6) также применяет принцип «разделяй и властвуй», но более стабильно: массив рекурсивно делится пополам до отдельных элементов, затем упорядоченные части сливаются в один массив. Этот алгоритм всегда работает за $O(n \cdot \log n)$, но требует дополнительной памяти $O(n)$ для хранения промежуточных результатов.

```

44
2 usages
45 def merge_sort(arr):
46     if len(arr) <= 1:
47         return arr
48     mid = len(arr) // 2
49     left = merge_sort(arr[:mid])
50     right = merge_sort(arr[mid:])
51     return merge(left, right)
52
53
1 usage
54 def merge(left, right):
55     result = []
56     i = j = 0
57     while i < len(left) and j < len(right):
58         if left[i] < right[j]:
59             result.append(left[i])
60             i += 1
61         else:
62             result.append(right[j])
63             j += 1
64     result.extend(left[i:])
65     result.extend(right[j:])
66     return result
67

```

Рис 6. Сортировка слиянием

6. Сортировка кучей (Heap Sort) (рис. 7) преобразует массив в двоичную кучу (обычно max-кучу), после чего максимальный элемент (корень) последовательно извлекается и помещается в конец массива. Процесс повторяется для уменьшающейся кучи. Алгоритм гарантирует сложность $O(n \cdot \log n)$ и не требует дополнительной памяти, но менее эффективен на практике, чем Quick Sort, из-за константных множителей.

```

68
69 def heap_sort(arr):
70     h = []
71     for value in arr:
72         heappush(h, value)
73     return [heappop(h) for _ in range(len(h))]
74

```

Рис 7. Сортировка кучей

Теперь создаются две функции: одна для генерации данных (рис. 8), другая для измерения времени выполнения (рис. 9). Генерироваться будут три типа данных:

- Случайные (рандомные): Элементы расположены в произвольном порядке (например, [5, 2, 9, 1, 3]). Наиболее "честный" тест для алгоритмов, так как нет предсказуемых паттернов генерации. Характерны для данных из сенсоров, логов, генерации случайных чисел.

- Частично упорядоченные: Большая часть элементов уже отсортирована, но есть локальные нарушения (например, [1, 2, 4, 3, 5, 6, 8, 7]). Часто встречаются в реальных данных: обновление БД, добавление новых записей в отсортированный список.

- Обратно упорядоченные: Элементы расположены в порядке убывания (например, [9, 7, 5, 3, 1]). Встречается реже, но важен для оценки устойчивости алгоритмов.

```
75
76 1 usage
77 def generate_data(n):
78     return {
79         'random': np.random.randint(low=0, high=100000, size=n),
80         'partially_sorted': np.concatenate([
81             np.sort(np.random.randint(low=0, high=100000, size=int(n*0.75))),
82             np.random.randint(low=0, high=100000, size=int(n*0.25))
83         ]),
84         'reverse_sorted': np.sort(np.random.randint(low=0, high=100000, size=n))[::-1]
85     }
```

Рис 8. Генерация данных

```
86
87 def measure_performance(sort_func, data_type, sizes, repeats=5):
88     times = []
89     for n in sizes:
90         data = generate_data(n)[data_type]
91         time_taken = timeit.timeit(
92             lambda: sort_func(deepcopy(data)),
93             number=repeats
94         ) / repeats
95         times.append(time_taken)
96     return times
97
```

Рис 9. Измерение времени выполнения

Для измерения времени каждый алгоритм для каждого типа данных будет запускаться пять раз, поскольку каждый результат может быть разным, и высчитывать среднее арифметическое.

Реализуем тестирование алгоритмов (рис. 10) Проверим скорость выполнения каждого из них на 100, 1000, 5000 и 10000 числах.

```

98
99 sizes = [100, 1000, 5000, 10000]
100 algorithms = {
101     'Bubble': bubble_sort,
102     'Insertion': insertion_sort,
103     'Selection': selection_sort,
104     'Quick': quick_sort,
105     'Merge': merge_sort,
106     'Heap': heap_sort
107 }
108 data_types = ['random', 'partially_sorted', 'reverse_sorted']
109
110 results = {alg: {dtype: [] for dtype in data_types} for alg in algorithms}
111
112 for alg_name, alg_func in algorithms.items():
113     for dtype in data_types:
114         results[alg_name][dtype] = measure_performance(alg_func, dtype, sizes)
115

```

Рис 10. Реализация тестирования алгоритмов

Для визуализации результатов был написан код для построения графиков зависимостей времени выполнения от размеров данных (рис. 11), а также выведены результаты в консоль в виде таблицы, где строками являются названия алгоритмов, столбцами — типы данных, значениями — среднее арифметическое результатов выполнения на данных размером 5000 и 10000 (рис. 12).

```

116
117 plt.figure(figsize=(18, 12))
118 colors = plt.cm.tab10.colors
119
120 for i, dtype in enumerate(data_types):
121     plt.subplot(*args: 2, 2, i+1)
122     for j, (alg_name, _) in enumerate(algorithms.items()):
123         plt.plot(*args: sizes, results[alg_name][dtype], 'o-',
124                 color=colors[j], label=alg_name)
125     plt.xscale('log')
126     plt.yscale('log')
127     plt.xlabel('Размер массива')
128     plt.ylabel('Время (сек)')
129     plt.title(f'Производительность на {dtype} данных')
130     plt.grid(True)
131     plt.legend()
132
133 plt.tight_layout()
134 plt.show()

```

Рис 11. Код для визуализация результатов

```

135
136 print("Среднее время выполнения (секунды):")
137 print(f"{'Алгоритм':<12}", end="")
138 for dtype in data_types:
139     print(f"{dtype[:15]:>20}", end="")
140 print()
141
142 for alg_name in algorithms:
143     print(f"{alg_name:<12}", end="")
144     for dtype in data_types:
145         avg_time = np.mean(results[alg_name][dtype][-2:])
146         print(f"{avg_time:>20.5f}", end="")
147     print()
148

```

Рис 12. Код вывода результатов в виде таблицы

По истечении 18 минут открывается отдельное окно с графиком производительности каждого алгоритма на каждый тип данных (рис. 13) и табличное представление результатов (табл. 1).

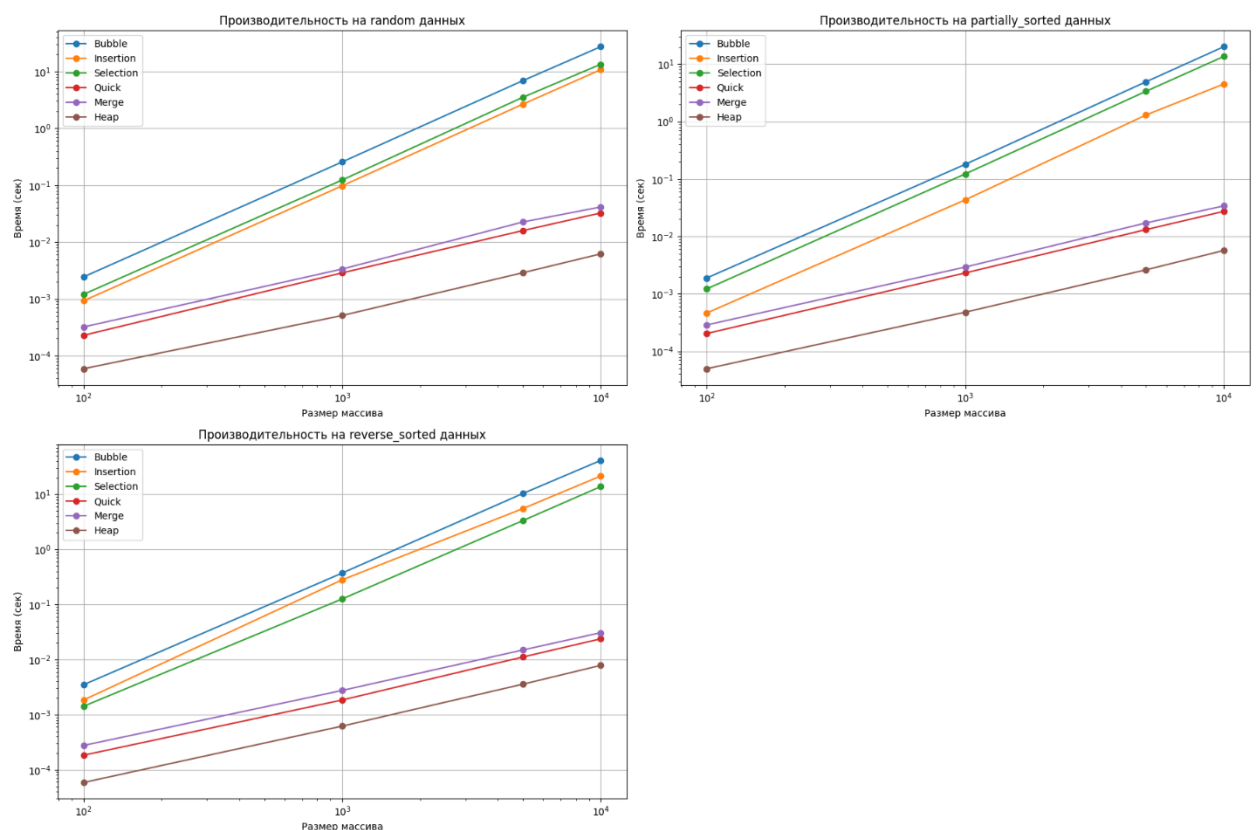


Рис 13. Графики производительностей алгоритмов

Таблица 1 – Время выполнения алгоритмов в секундах

Алгоритм	random	partially_sorted	reverse_sorted
Bubble	18.48362	12.24123	25.87814
Insertion	6.86949	3.09796	13.79666
Selection	8.61471	8.53728	8.68444
Quick	0.02336	0.02075	0.01966
Merge	0.02966	0.02494	0.02428
Heap	0.00600	0.00422	0.00562

В результате проведённого исследования было выявлено, что алгоритмы с временной сложностью $O(n \cdot \log n)$ (Quick Sort, Merge Sort, Heap Sort) продемонстрировали существенное преимущество при работе с большими массивами данных (5000 и 10000 элементов). Квадратичные алгоритмы (Bubble Sort, Insertion Sort, Selection Sort) показали приемлемую производительность только на небольших наборах данных (до 1000 элементов). Сортировка пузырьком является самым медленным из рассмотренных алгоритмов, поэтому на практике её применять не стоит. Хотя сортировки вставками и выборами работают не так быстро, их вполне хватает для решения задач с небольшими объёмами данных.

Библиографический список

1. Дупленко А. Г. Сравнительный анализ алгоритмов сортировки данных в массивах // Молодой ученый. 2013. №. 8. С. 50-53.
2. Заика И. В., Тюшнякова И. А. Обзор методов сортировки // Научный взгляд в будущее. 2016. Т. 2. №. 1. С. 206-211.
3. Тимшина Д. В., Максютя А. О. Алгоритмы сортировки для автоматизации решения задачи информационной логистики // Вестник академии знаний. 2014. №. 3 (10). С. 29-38.
4. Козлов М. А., Скворцов С. В. Алгоритмы параллельной сортировки данных и их реализация на языке Clojure // Вестник Рязанского государственного радиотехнического университета. 2013. №. 46-1. С. 92-96.