Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

СЕМЕСТРОВАЯ РАБОТА

по дисциплине «Алгоритмы и анализ сложности»  
«Экспериментальный анализ различных методов сортировки»

Обучающийся: Хисматуллин Ленар Марселевич гр. 09–332

(ФИО студента) (Группа)

Руководитель: к.ф.-м.н., доцент КСАИТ, А. В. Васильев

Казань – 2025

**Оглавление**

[Введение 3](#_Toc162345006)

[Постановка задачи 4](#_Toc162345007)

[Методика проведения эксперимента 5](#_Toc162345008)

[Полученные результаты 8](#_Toc162345009)

[Заключение 19](#_Toc162345010)

[Приложение 1. Программный код 21](#_Toc162345011)

**Введение**  
Задача сортировки данных — одна из фундаментальных в информатике и программировании. Существует множество алгоритмов, отличающихся сложностью, эффективностью и областью применения. Теоретически большинство алгоритмов уже хорошо исследованы, однако их практическое поведение на разных типах данных и в разных условиях часто оказывается непредсказуемым.

Цель данной работы — провести экспериментальный анализ различных методов сортировки, проверить, насколько их поведение соответствует теоретическим оценкам, и определить, какие алгоритмы оказываются наиболее эффективными в тех или иных ситуациях.

Для реализации и тестирования использован язык программирования Python. Все сортировки реализованы вручную или с использованием встроенных механизмов, а также написаны генераторы тестовых данных, автоматизирован запуск тестов и построение визуализации.

Постановка задачи

1. Реализовать различные методы сортировки и проверить на практике оценки их сложности:

* простые схемы
* сортировка Шелла (с различными длинами промежутков)
* быстрая сортировка
* сортировка слиянием
* сортировка кучей
* поразрядная сортировка
* встроенная в язык программирования сортировка

2. Сравнить их между собой на различных входных данных:

* массивах различных типов
* массивах различной длины
* по-разному порожденных массивах

3. Наглядно представить полученные результаты и сделать выводы

Методика проведения эксперимента

1.Реализация алгоритмов сортировки

Были реализованы 13 алгоритмов сортировки: простые (вставками, пузырьком, выбором, рангам), эффективные (быстрая, слиянием, кучей, Шелла, интроспективная, TimSort (в Python встроенная сортировка (sorted()) использует алгоритм TimSort, реализованный на C++ для эффективности. Я воспроизвёл этот алгоритм на чистом Python, чтобы лучше понять его работу.)) и специальные (поразрядные — поразрядная по младшему разряду (LSD) и поразрядная по старшему разряду (MSD), сортировка по счёту). Все сортировки реализованы на языке Python. Также была реализована сортировка ведрами. Реализация большинства алгоритмов допускает передачу ключевой функции key, что позволяет сортировать любые структуры данных, включая строки, даты и байты. Все сортировки запускались в защищённом блоке try/except, что позволило сохранить информацию о возможных ошибках (например, неподдерживаемые типы данных в некоторых алгоритмах).

2.Подготовка тестовых данных

С помощью генераторов были подготовлены входные данные четырёх типов:

* Однобайтовые числа (0–9)
* Целые числа
* Строки
* Даты

Каждый тип данных был представлен в шести вариантах

* Полностью случайный массив
* Массив, упорядоченный в обратном порядке
* Почти отсортированный массив
* Массив с большим числом повторяющихся элементов
* Массив с неупорядоченным «хвостом»
* Массив с несколькими случайными вставками

Массивы формировались пяти различных размеров:

* 10
* 100
* 1 000
* 10 000
* 100 000 элементов

Таким образом, была обеспечена широта входных данных, охватывающая как простые случаи, так и типичные реальные сценарии. Для каждого типа массива были реализованы отдельные генераторы, находящиеся в utils/generator\_data.py

3.Измерение времени выполнения

Для оценки производительности использовалась функция measure\_time, запускающая сортировку многократно на копиях исходного массива и возвращающая среднее время одного выполнения.

Число повторов выбиралось в зависимости от размера массива:

50 повторов — для массивов длиной до 1 000 элементов

10 повторов — для массивов от 10 000 элементов и выше

Такой подход обеспечивал надёжность измерений при работе с малыми объёмами и оптимизировал время эксперимента при больших. Время фиксировалось с помощью функции time.perf\_counter\_ns() и переводилось в секунды.

4.Сохранение и структура результатов

Результаты каждого теста сохранялись в словарь с ключом, отражающим тип массива и его длину (например, random\_int\_10000), и далее экспортировались в JSON-файл results.json. Для каждого случая сохранялось среднее время выполнения каждого алгоритма либо сообщение об ошибке при возникновении исключений.

5.Визуализация

Для анализа использовались следующие типы графиков:

1. Графики времени выполнения:

Зависимость времени от размера массива (логарифмическая шкала).

Сравнение алгоритмов на одном типе данных.

1. Относительная производительность:

Нормировка времени относительно лучшего алгоритма для каждого теста.

1. Проверка сложности:

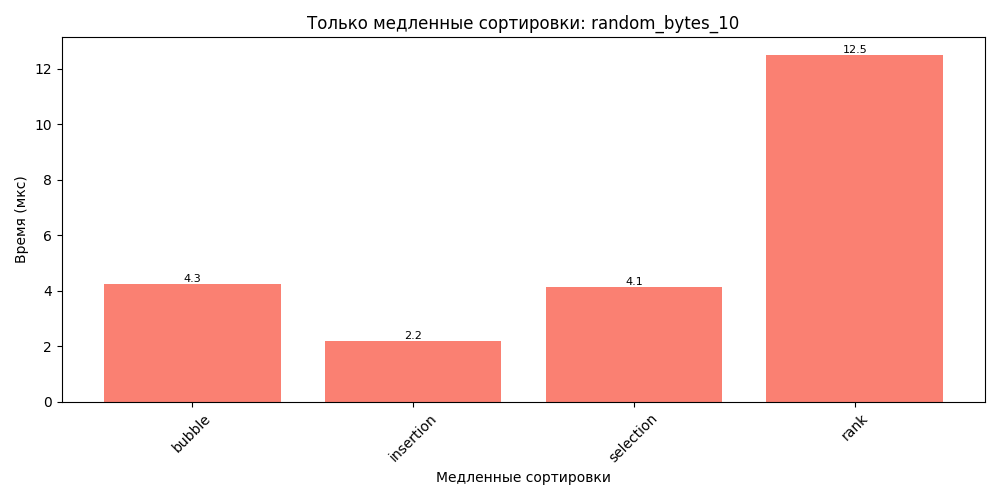
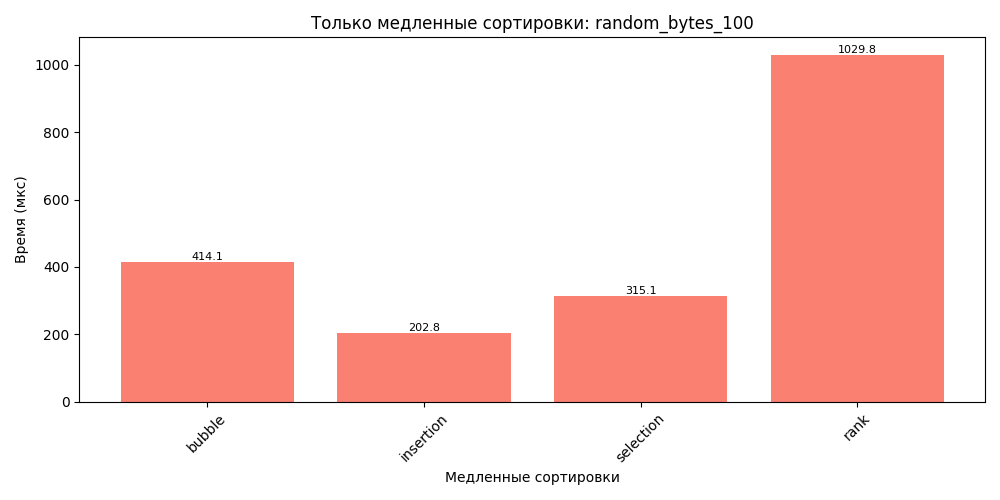
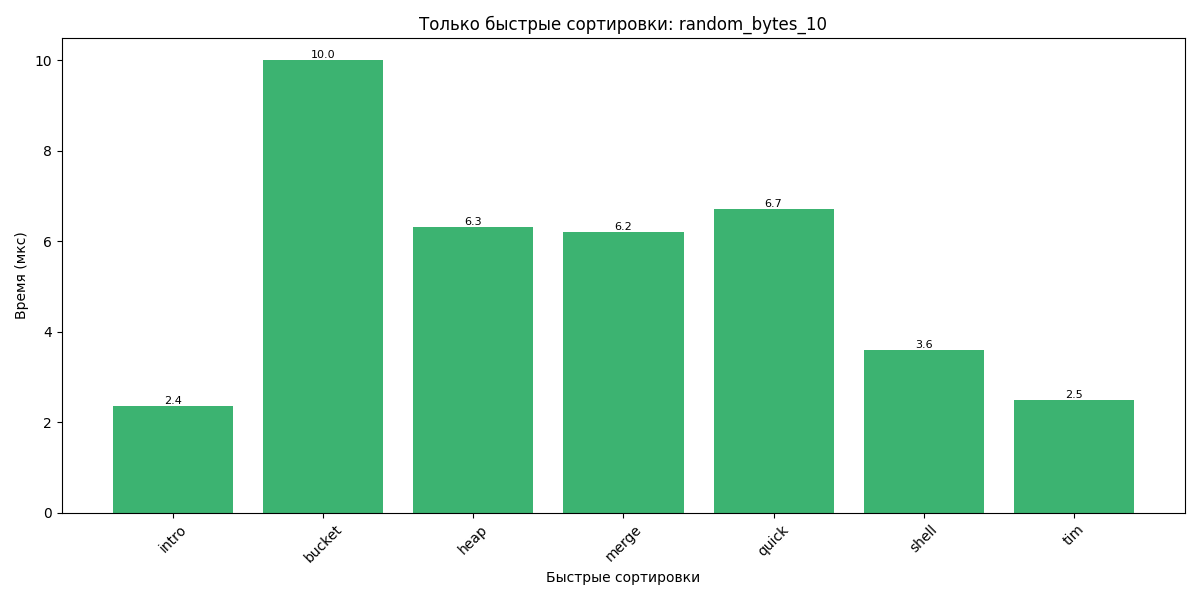
Сравнение реального времени с теоретической оценкой

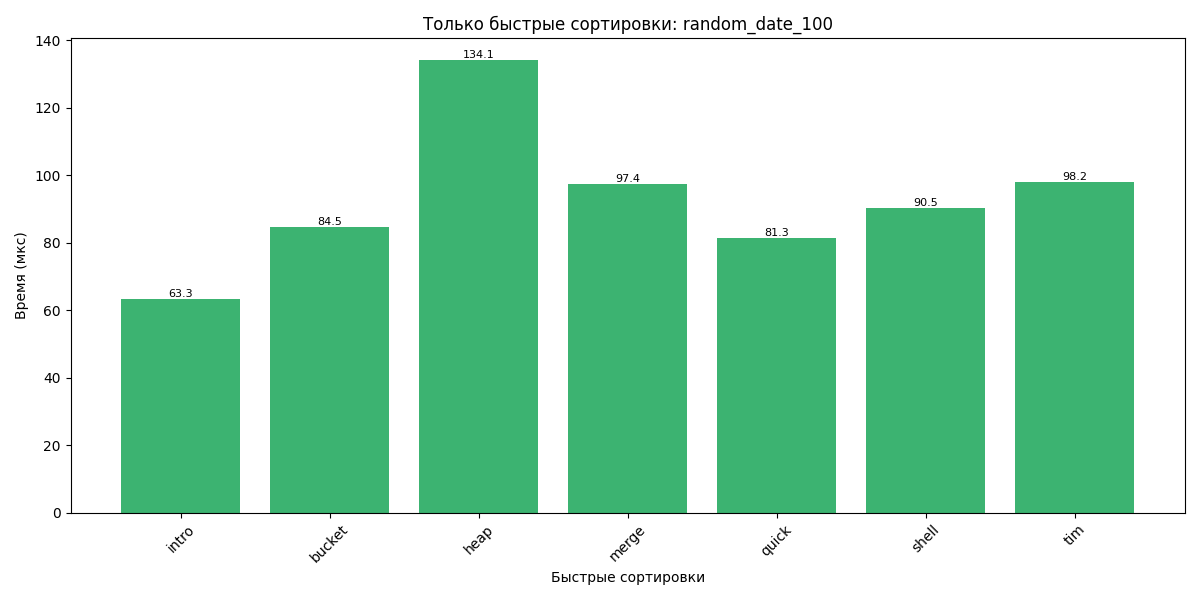
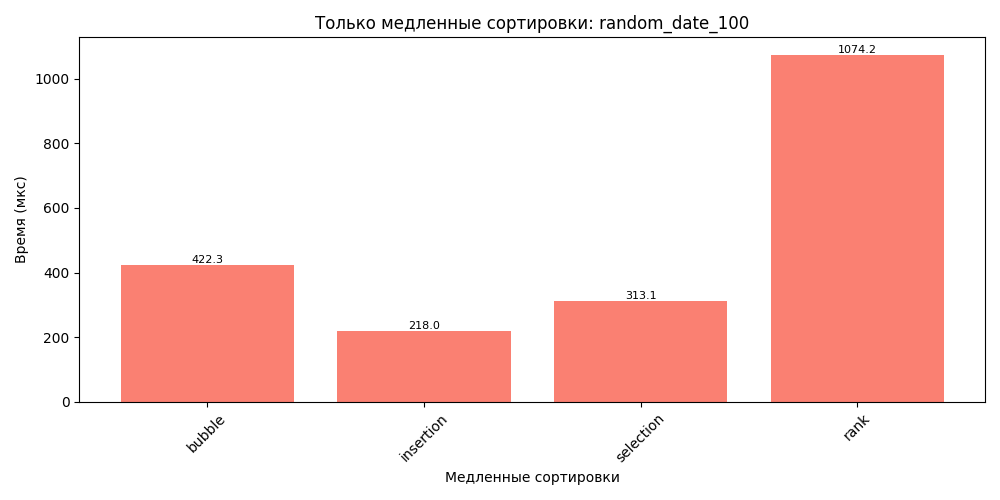
Полученные результаты

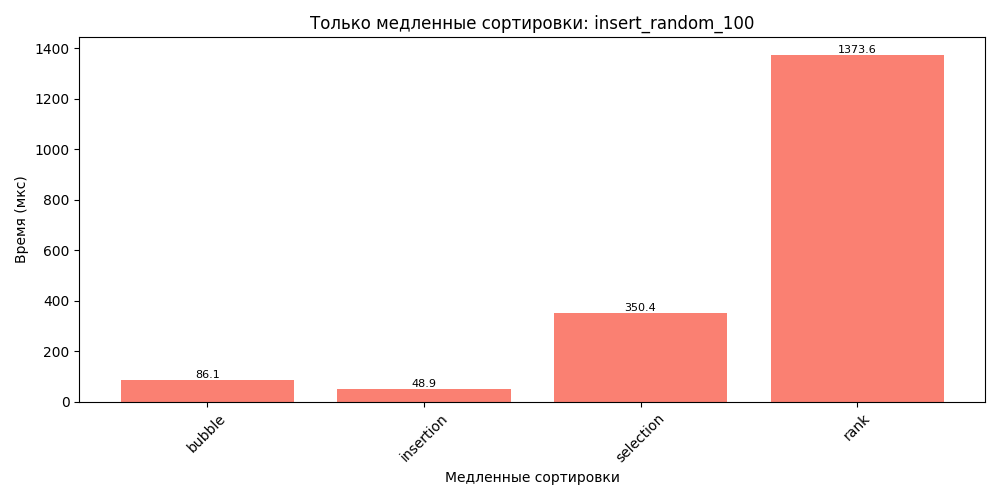
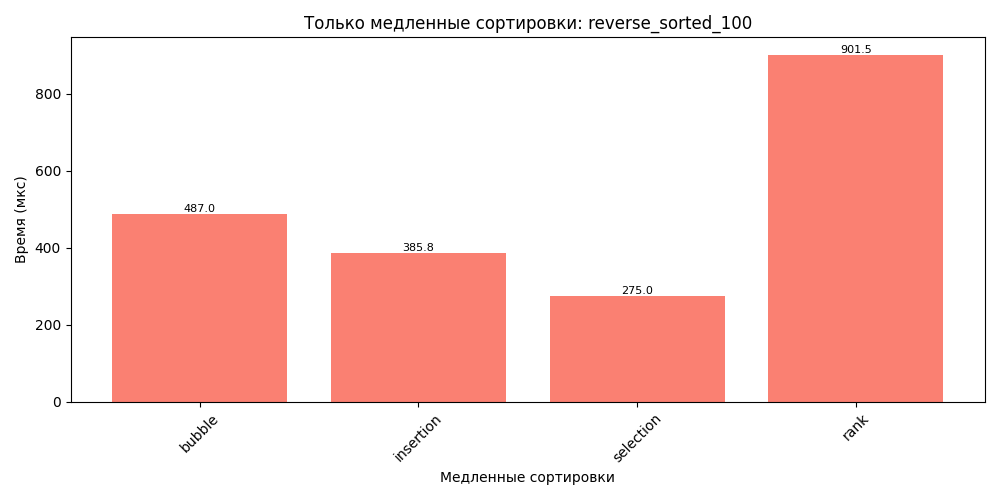
В ходе экспериментального исследования были получены и проанализированы количественные показатели эффективности 13 алгоритмов сортировки при работе с различными типами данных и структурами массивов. Ниже приведены основные выводы и наблюдения, полученные в результате анализа собранных данных и визуализации.

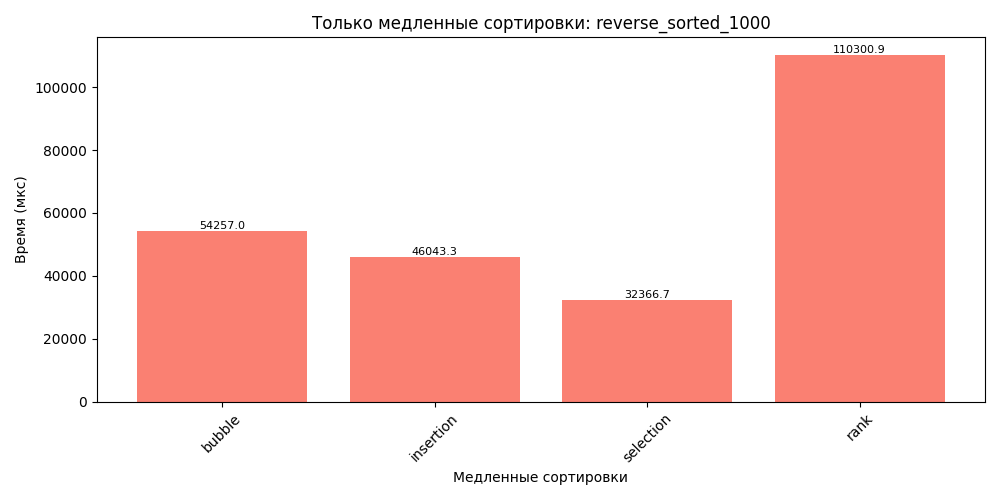
Анализ простых алгоритмов сортировки

Простые (квадратичные) алгоритмы сортировки — такие как сортировка пузырьком, вставками, выбором и по рангам — были протестированы на массивах различных типов и размеров. Их поведение и производительность полностью соответствуют ожидаемой теоретической сложности.

   Как можно заметить на небольшом количестве данных (10 — 1000) все простые алгоритмы не масштабируются: при росте входных данных в 10 раз время увеличивается примерно в 100 раз. \*\*\*Скорость сортировки простых алгоритмов чуть быстрее на малых количествах данных чисел в размере 10, чем у эффективных сортировок, однако в сортировке более больших количествах данных, лучше справляются эффективные сортировки, что показано на графиках ниже.

Так же можно заметить, что сортировка по рангам оказалась наиболее медленной, даже на массивах из 10 элементов уступая остальным простым алгоритмам в несколько раз. На практике он показал наихудшие результаты по времени среди всех рассмотренных сортировок, особенно при увеличении размера массива. Это объясняется его высокой вычислительной сложностью.На частично отсортированных массивах простейшая сортировка вставками и пузырьком оказывается наиболее эффективной среди квадратичных алгоритмов. 



На упорядоченных в обратном порядке массивах простейшая сортировка вставками оказывается наиболее эффективной среди квадратичных алгоритмов.

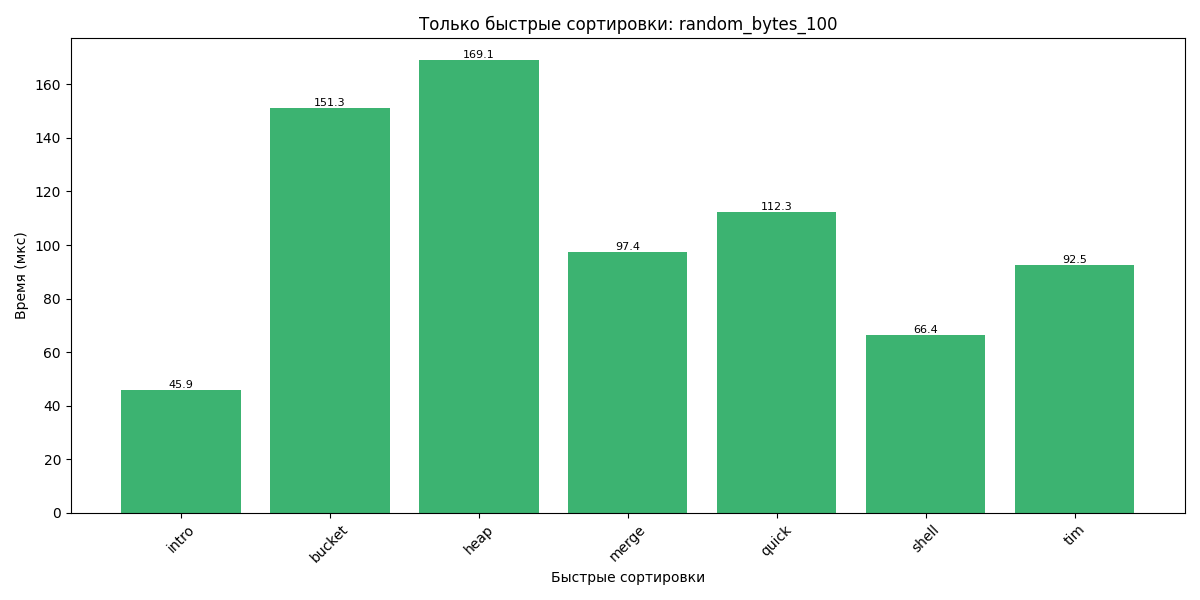
Анализ эффективных алгоритмов сортировки

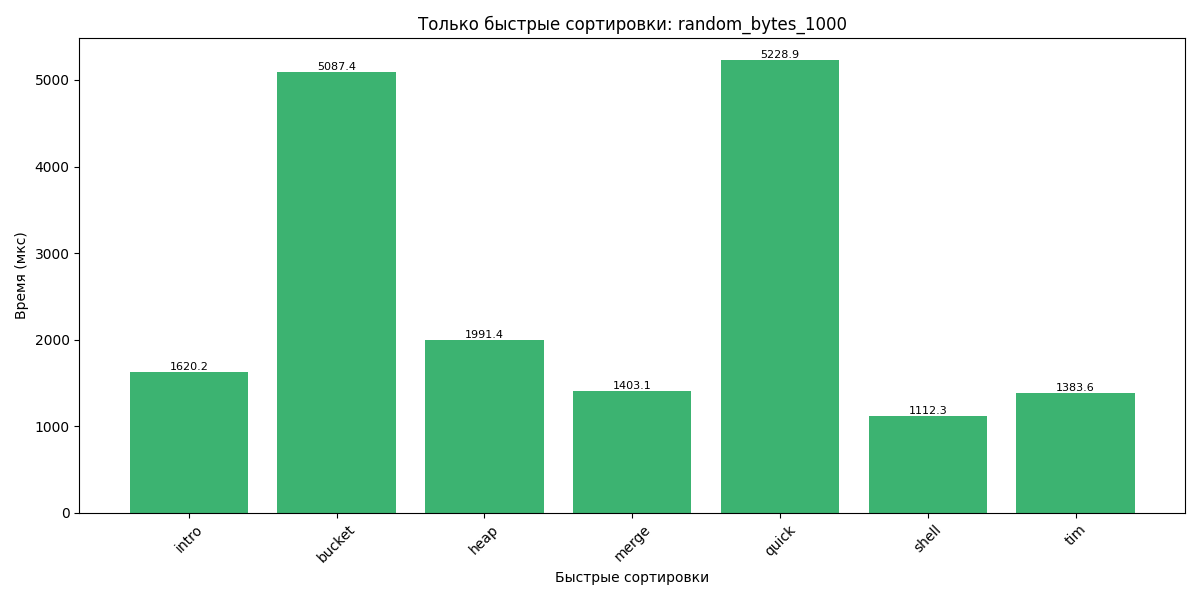
Быстрые алгоритмы сортировки (такие как быстрая, слиянием, пирамидальная, Шелла, интроспективная, timsort, а также поразрядные LSD и MSD) подтвердили свои теоретические оценки сложности в большинстве тестов.

Однако, несмотря на общее подтверждение теоретической сложности, поведение алгоритмов может существенно меняться в зависимости от структуры входных данных.

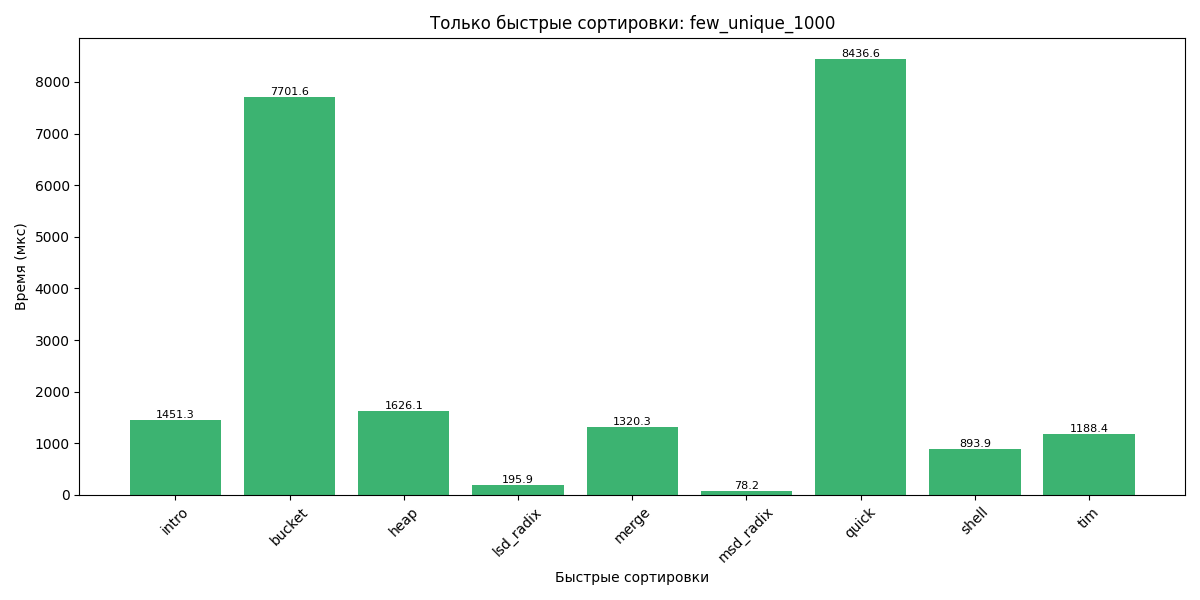
Особенно это проявляется на массивах с нестандартным распределением значений — например, когда массив содержит ограниченное количество уникальных элементов.

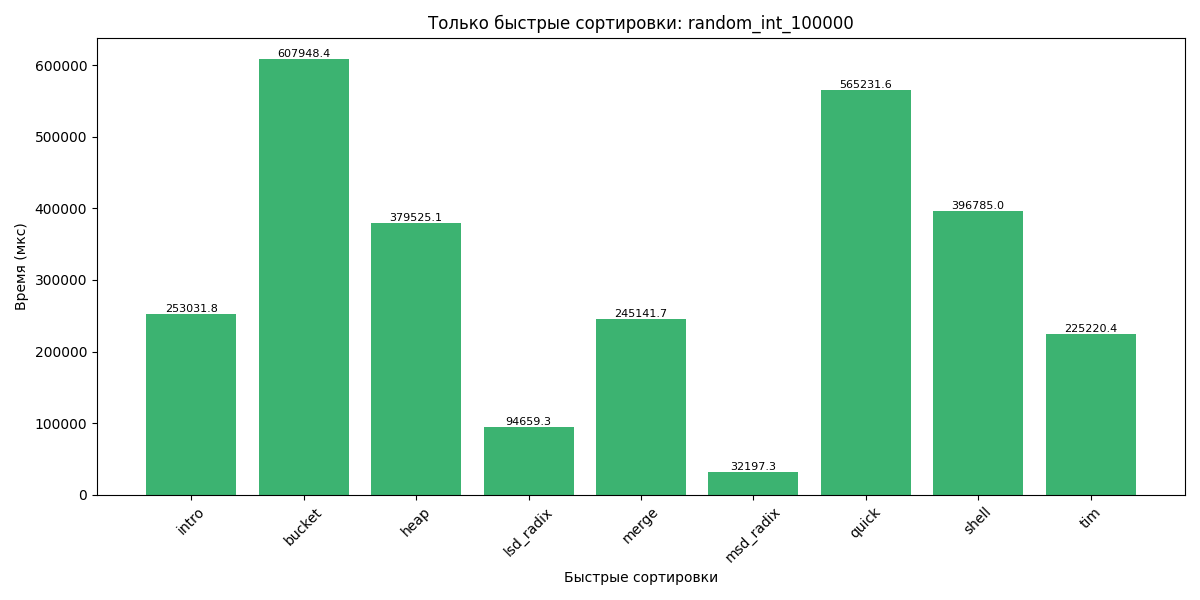
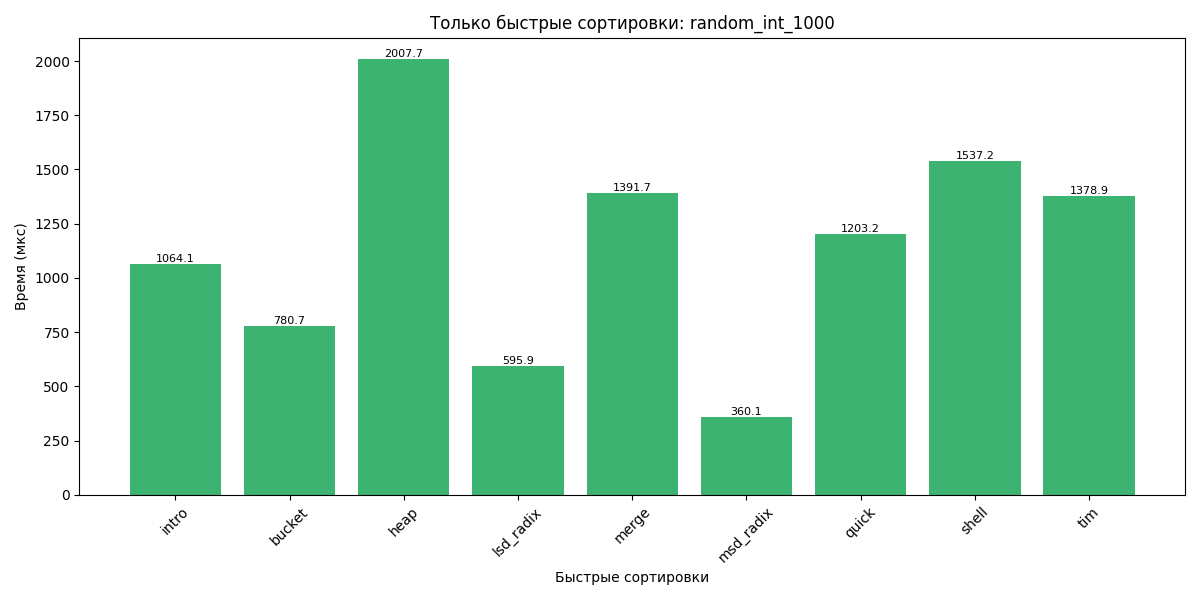
Один из таких случаев рассмотрен на следующих графиках.



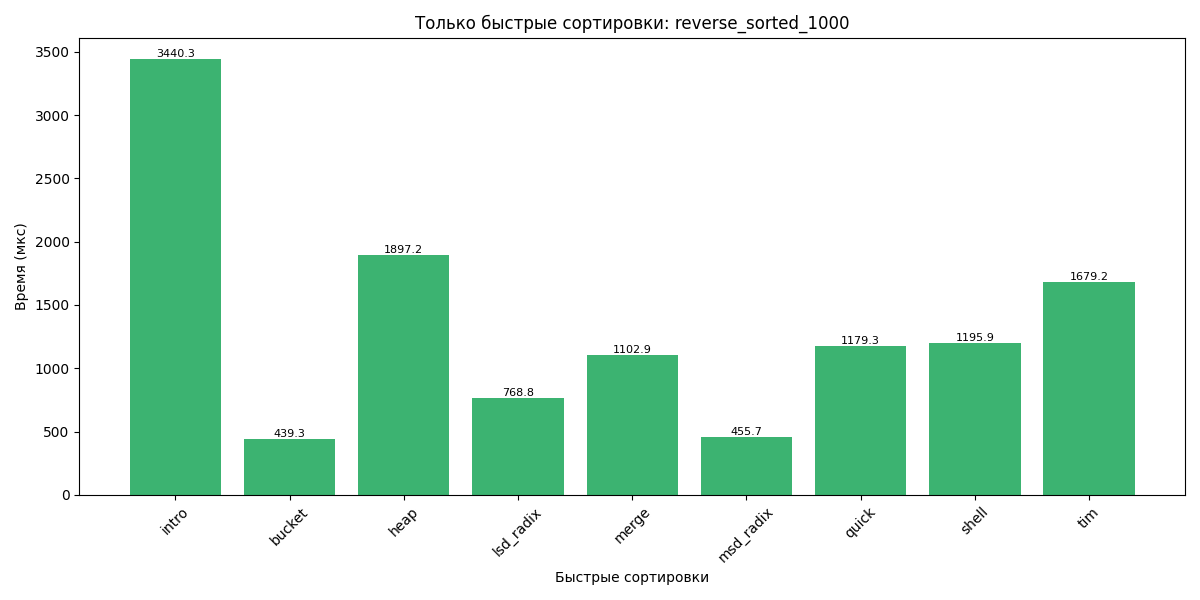


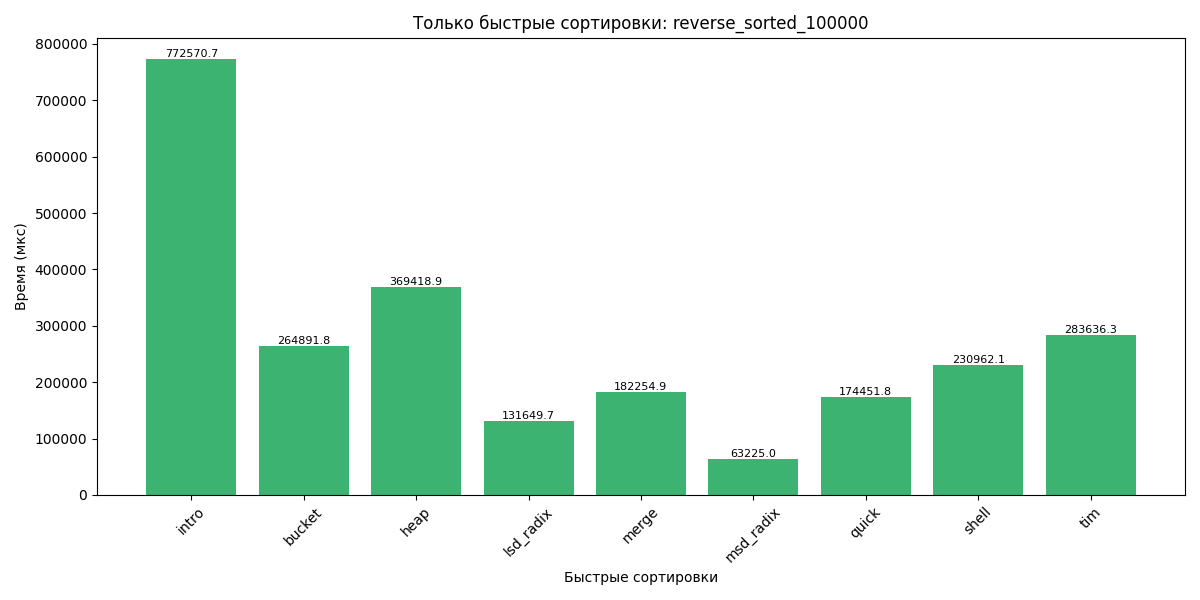
На этих графиках сравниваются только эффективные алгоритмы сортировки, работающие с массивом из 100 и 10 000 элементов, состоящим из однобайтовых значений (0–9). Ожидалось, что такие короткие значения будут обрабатываться быстро, но результат оказался неожиданным для двух алгоритмов. Оказалось, что разбиения, выполняемые quick sort, становятся очень неравномерными. Это приводит к резкому увеличению количества рекурсий. Сортировка bucket sort работает на основе quick sort, что приводит к аналогичной скорости работы. Аналогичная ситуация с сортировкой массива с несколькими уникальными значениями.



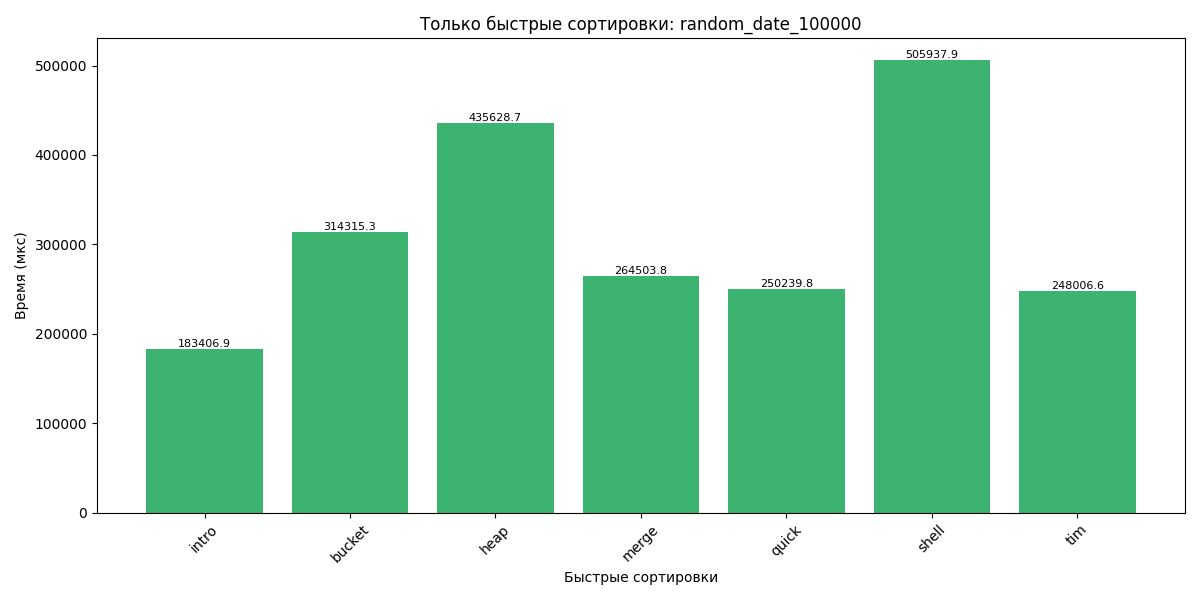
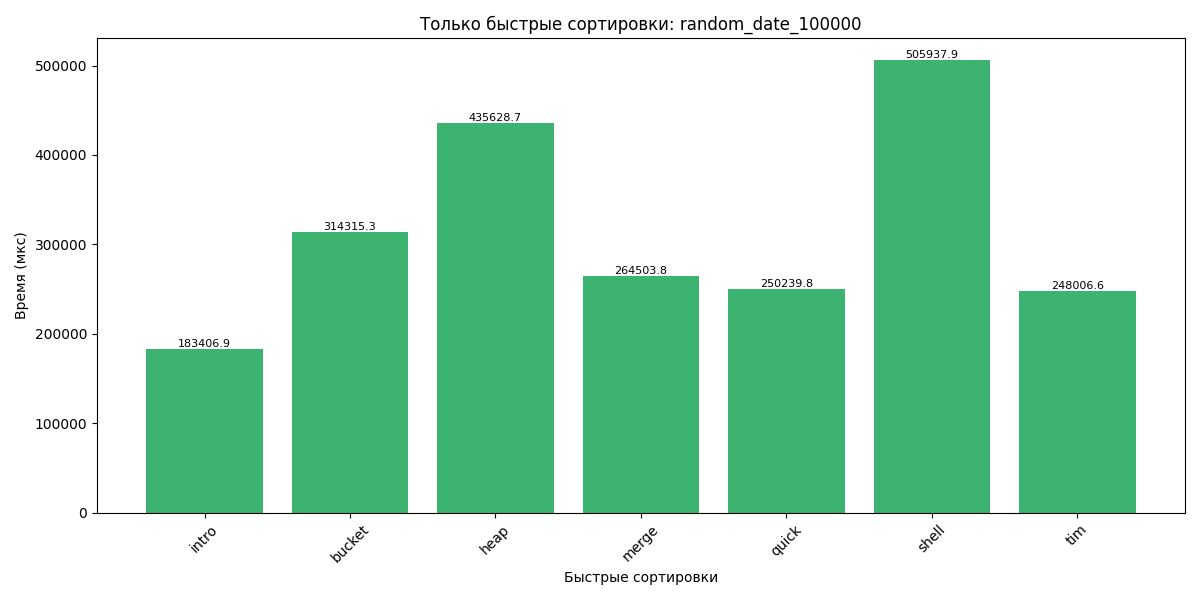
\*Далее рассмотрим поразрядные сортировки.  
Как и ожидалось поразрядные сортировки особенно эффективны на целых числах ограниченной длины, что и наблюдается на данном тесте.

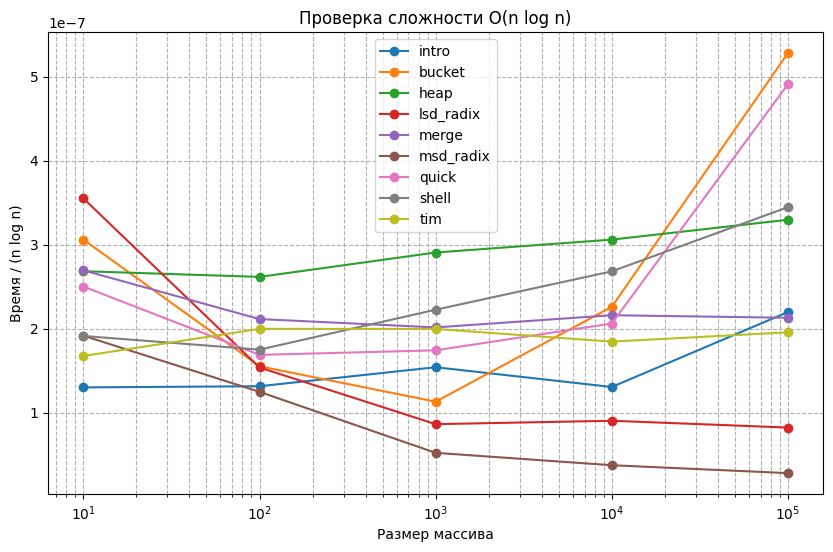
Кроме того, поразрядные сортировки демонстрируют высокую устойчивость к данным в обратном порядке, сохраняя высокую производительность даже на массивах значительного размера. На представленных ниже графиках наглядно отражены результаты их работы.





Также на этих графиках можно заметить, что IntroSort показал худший результат. то объясняется тем, что при плохом разбиении IntroSort переключается на HeapSort, который неэффективен на маленьких массивах.

 В эксперименте сортировок дат и строк все алгоритмы сортировки сравниваются на одном и том же наборе данных что позволяет объективно оценить, что в данных сортировках, лучше всего подходит intro sort, а также неплохими вариантами могут быть merge sort, quick sort, и timsort. А heap sort и shell sort лучше не использовать в данном случае.

  
Проверка сложности демонстрируют поведение, близкое к особенно на больших входных данных.

Заключение

В ходе эксперимента мы протестировали 13 алгоритмов сортировки на массивах разной структуры и с разными типами данных. Результаты в целом подтвердили теоретические оценки сложности, но также показали много интересных практических нюансов.

Простые алгоритмы, такие как сортировка пузырьком, вставками, выбором и по рангам, быстро теряют эффективность уже на массивах средней длины. Их квадратичная сложность —

— хорошо заметна: если размер массива увеличивается в 10 раз, время выполнения вырастает примерно в 100 раз. Однако у них всё же есть своя ниша: на очень коротких или почти отсортированных массивах (особенно в случае сортировки вставками или пузырьком) они могут обойти даже продвинутые алгоритмы — за счёт минимальных накладных расходов.

Поразрядные сортировки (LSD и MSD) проявили себя лучше всех на массивах с целыми числами и байтами в ограниченном диапазоне. Они стабильно дают самое быстрое время сортировки, даже на больших объёмах данных, и остаются устойчивыми даже при сложных начальных условиях, например, когда массив отсортирован в обратном порядке.

Универсальные эффективные алгоритмы (quick sort, merge sort, heap sort, tim sort, intro sort, shell sort) в большинстве случаев подтвердили свою асимптотику но при этом вели себя по-разному в зависимости от структуры данных.

Quick sort отлично справляется со случайными данными, но может сильно тормозить, если плохо выбрать опорный элемент — особенно на массивах с небольшим количеством уникальных значений.

Intro sort при неудачном разбиении переключается на heap sort, который, как показали тесты, оказался в некоторых случая одним из самых медленных.

Timsort продемонстрировал стабильность и высокую универсальность, он даёт хорошие результаты вне зависимости от типа данных.

Shell sort неожиданно показал высокое время выполнения в некоторых случаях, так что его стоит использовать с осторожностью и только при понимании специфики данных.

Наконец, при работе со строками, датами и пользовательскими структурами (например, классами с переопределённым сравнением), лучше всего себя показали quick sort, intro sort и timsort. Они сохраняют высокую производительность даже на больших объёмах и при сложных условиях сравнения.

Приложение 1. Программный код

Bubble sort:

def bubble\_sort(arr, key=lambda x: x):  
 n = len(arr)  
 for i in range(n):  
 swapped = False  
 for j in range(0, n - i - 1):  
 if key(arr[j]) > key(arr[j + 1]):  
 arr[j], arr[j + 1] = arr[j + 1], arr[j]  
 swapped = True  
 if not swapped:  
 break  
 return arr

Bucket sort:  
from .quick\_sort import quick\_sort  
from datetime import datetime  
def bucket\_sort(arr, bucket\_size=None, key=lambda x: x):  
 if not arr:  
 return []  
 def to\_numeric(val):  
 result = key(val)  
 if isinstance(result, bytes):  
 return int.from\_bytes(result, 'little')  
 elif isinstance(result, datetime):  
 return result.timestamp()  
 return result  
 numeric\_arr = [(item, to\_numeric(item)) for item in arr]  
 min\_k = min(numeric\_arr, key=lambda x: x[1])[1]  
 max\_k = max(numeric\_arr, key=lambda x: x[1])[1]  
 if min\_k == max\_k:  
 return [item for item, \_ in numeric\_arr]  
 if bucket\_size is None:  
 bucket\_size = (max\_k - min\_k) / len(arr)  
 bucket\_count = min(int((max\_k - min\_k) / bucket\_size) + 1, 1000)  
 buckets = [[] for \_ in range(bucket\_count)]  
 for item, num in numeric\_arr:  
 index = int((num - min\_k) / bucket\_size)  
 index = min(index, bucket\_count - 1)  
 buckets[index].append(item)  
 result = []  
 for bucket in buckets:  
 quick\_sort(bucket, key=key)  
 result.extend(bucket)  
 return result

Heap\_sort:

def heapify(arr, n, i, key):  
 largest = i  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
 if left < n and key(arr[left]) > key(arr[largest]):  
 largest = left  
 if right < n and key(arr[right]) > key(arr[largest]):  
 largest = right  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i]  
 heapify(arr, n, largest, key)  
def heap\_sort(arr, key=lambda x: x):  
 n = len(arr)  
 for i in range(n // 2 - 1, -1, -1):  
 heapify(arr, n, i, key)  
 for i in range(n - 1, 0, -1):  
 arr[i], arr[0] = arr[0], arr[i]  
 heapify(arr, i, 0, key)  
 return arr

Insertion sort:  
def insertion\_sort(arr, key=lambda x: x):  
 for i in range(1, len(arr)):  
 current = arr[i]  
 j = i - 1  
 while j >= 0 and key(current) < key(arr[j]):  
 arr[j + 1] = arr[j]  
 j -= 1  
 arr[j + 1] = current  
 return arr

Intro sort:  
def intro\_sort(arr, key=lambda x: x):  
 max\_depth = 2 \* (len(arr).bit\_length())  
 \_intro\_sort(arr, 0, len(arr) - 1, max\_depth, key)  
def \_intro\_sort(arr, start, end, max\_depth, key):  
 if end - start < 16:  
 \_insertion\_sort(arr, start, end, key)  
 elif max\_depth == 0:  
 \_heap\_sort(arr, start, end, key)  
 else:  
 pivot = partition(arr, start, end, key)  
 \_intro\_sort(arr, start, pivot - 1, max\_depth - 1, key)  
 \_intro\_sort(arr, pivot + 1, end, max\_depth - 1, key)  
def \_insertion\_sort(arr, start, end, key):  
 for i in range(start + 1, end + 1):  
 current = arr[i]  
 j = i - 1  
 while j >= start and key(arr[j]) > key(current):  
 arr[j + 1] = arr[j]  
 j -= 1  
 arr[j + 1] = current  
def \_heap\_sort(arr, start, end, key):  
 def heapify(n, i):  
 largest = i  
 left = 2 \* i + 1  
 right = 2 \* i + 2  
 if left < n and key(heap[left]) > key(heap[largest]):  
 largest = left  
 if right < n and key(heap[right]) > key(heap[largest]):  
 largest = right  
 if largest != i:  
 heap[i], heap[largest] = heap[largest], heap[i]  
 heapify(n, largest)  
 heap = arr[start:end + 1]  
 n = len(heap)  
 for i in range(n // 2 - 1, -1, -1):  
 heapify(n, i)  
 for i in range(n - 1, 0, -1):  
 heap[0], heap[i] = heap[i], heap[0]  
 heapify(i, 0)  
 arr[start:end + 1] = heap  
def partition(arr, low, high, key):  
 pivot = arr[high]  
 i = low  
 for j in range(low, high):  
 if key(arr[j]) < key(pivot):  
 arr[i], arr[j] = arr[j], arr[i]  
 i += 1  
 arr[i], arr[high] = arr[high], arr[i]  
 return i

LSD\_radix:

def counting\_sort(arr, exp):  
 n = len(arr)  
 output = [0] \* n  
 count = [0] \* 10  
 for i in range(n):  
 index = (arr[i] // exp) % 10  
 count[index] += 1  
 for i in range(1, 10):  
 count[i] += count[i - 1]  
 i = n - 1  
 while i >= 0:  
 index = (arr[i] // exp) % 10  
 output[count[index] - 1] = arr[i]  
 count[index] -= 1  
 i -= 1  
 for i in range(n):  
 arr[i] = output[i]  
def radix\_sort\_lsd(arr):  
 if not arr:  
 return arr  
 max\_num = max(arr)  
 exp = 1  
 while max\_num // exp > 0:  
 counting\_sort(arr, exp)  
 exp \*= 10  
 return arr

Merge sort:

def merge\_sort(arr, key=lambda x: x):  
 if len(arr) > 1:  
 mid = len(arr) // 2  
 left = arr[:mid]  
 right = arr[mid:]  
 merge\_sort(left, key=key)  
 merge\_sort(right, key=key)  
 i = j = k = 0  
 while i < len(left) and j < len(right):  
 if key(left[i]) < key(right[j]):  
 arr[k] = left[i]  
 i += 1  
 else:  
 arr[k] = right[j]  
 j += 1  
 k += 1  
 while i < len(left):  
 arr[k] = left[i]  
 i += 1  
 k += 1  
 while j < len(right):  
 arr[k] = right[j]  
 j += 1  
 k += 1  
 return arr

MSD radix sort:  
def msd\_radix\_sort(arr, digit=None):  
 if not arr:  
 return arr  
 if digit is None:  
 max\_num = max(arr)  
 digit = len(str(max\_num))  
 if digit <= 0 or len(arr) <= 1:  
 return arr  
 buckets = [[] for \_ in range(10)]  
 for num in arr:  
 d = (num // (10 \*\* (digit - 1))) % 10  
 buckets[d].append(num)  
 result = []  
 for bucket in buckets:  
 if bucket:  
 result += msd\_radix\_sort(bucket, digit - 1)  
 return result

Quick sort:

import random  
import sys  
sys.setrecursionlimit(2000)  
def partition(arr, low, high, key):  
 pivot\_index = random.randint(low, high)  
 arr[high], arr[pivot\_index] = arr[pivot\_index], arr[high]  
 pivot = key(arr[high])  
 i = low - 1  
 for j in range(low, high):  
 if key(arr[j]) <= pivot:  
 i += 1  
 arr[i], arr[j] = arr[j], arr[i]  
 arr[i + 1], arr[high] = arr[high], arr[i + 1]  
 return i + 1  
def quick\_sort(arr, low=0, high=None, key=lambda x: x):  
 if high is None:  
 high = len(arr) - 1  
 while low < high:  
 pi = partition(arr, low, high, key)  
 if pi - low < high - pi:  
 quick\_sort(arr, low, pi - 1, key=key)  
 low = pi + 1  
 else:  
 quick\_sort(arr, pi + 1, high, key=key)  
 high = pi - 1  
 return arr

Rank sort:

def rank\_sort(arr, key=lambda x: x):  
 n = len(arr)  
 ranks = [0] \* n  
 for i in range(n):  
 for j in range(n):  
 if key(arr[j]) < key(arr[i]) or (key(arr[j]) == key(arr[i]) and j < i):  
 ranks[i] += 1  
 sorted\_arr = [0] \* n  
 for i in range(n):  
 sorted\_arr[ranks[i]] = arr[i]  
 for i in range(n):  
 arr[i] = sorted\_arr[i]  
 return arr

Selection sort:

def selection\_sort(arr, key=lambda x: x):  
 for i in range(len(arr)):  
 min\_idx = i  
 for j in range(i + 1, len(arr)):  
 if key(arr[j]) < key(arr[min\_idx]):  
 min\_idx = j  
 arr[i], arr[min\_idx] = arr[min\_idx], arr[i]  
 return arr

Shell sort:

def shell\_sort(arr, key=lambda x: x, gaps=None):  
 n = len(arr)  
 if gaps is None:  
 gaps = []  
 gap = n // 2  
 while gap > 0:  
 gaps.append(gap)  
 gap //= 2  
 for gap in gaps:  
 for i in range(gap, n):  
 temp = arr[i]  
 j = i  
 while j >= gap and key(arr[j - gap]) > key(temp):  
 arr[j] = arr[j - gap]  
 j -= gap  
 arr[j] = temp  
 return arr

Tim sort:  
from .insertion\_sort import insertion\_sort  
def merge(arr, start, mid, end, key=lambda x: x):  
 if mid == end:  
 return  
 left = arr[start:mid+1]  
 right = arr[mid+1:end+1]  
 i = j = 0  
 for k in range(start, end+1):  
 if i < len(left) and (j >= len(right) or key(left[i]) <= key(right[j])):  
 arr[k] = left[i]  
 i += 1  
 else:  
 arr[k] = right[j]  
 j += 1  
def tim\_sort(arr, key=lambda x: x):  
 n = len(arr)  
 RUN = 30  
 for start in range(0, n, RUN):  
 end = min(start + RUN, n)  
 sorted\_block = insertion\_sort(arr[start:end], key=key)  
 arr[start:end] = sorted\_block  
 size = RUN  
 while size < n:  
 for start in range(0, n, size \* 2):  
 mid = min(start + size - 1, n - 1)  
 end = min(start + 2 \* size - 1, n - 1)  
 if mid < end:  
 merge(arr, start, mid, end, key=key)  
 size \*= 2  
 return arr

Generator\_data:

import random  
import string  
from datetime import datetime, timedelta  
def generate\_random\_int\_array(length: int, min\_value: int = 0, max\_value: int = 1000):  
 return [random.randint(min\_value, max\_value) for \_ in range(length)]  
def generate\_random\_string\_array(length: int, min\_len: int = 5, max\_len: int = 10):  
 return [''.join(random.choices(string.ascii\_letters, k=random.randint(min\_len, max\_len))) for \_ in range(length)]  
def generate\_random\_bytes\_array(length):  
 return [random.randint(0, 9).to\_bytes(1, 'little') for \_ in range(length)]  
def generate\_random\_date\_array(length: int, start: datetime, end: datetime):  
 delta = (end - start).days  
 return [start + timedelta(days=random.randint(0, delta)) for \_ in range(length)]  
def generate\_reverse\_sorted\_array(length: int):  
 return list(range(length, 0, -1))  
def generate\_almost\_sorted\_array(length: int, num\_unsorted: int = 10):  
 arr = list(range(length))  
 for \_ in range(num\_unsorted):  
 i = random.randint(0, length - 1)  
 j = random.randint(0, length - 1)  
 arr[i], arr[j] = arr[j], arr[i]  
 return arr  
def generate\_few\_unique\_array(length: int, num\_unique: int = 5):  
 return [random.randint(1, num\_unique) for \_ in range(length)]  
def generate\_tail\_random\_array(length, random\_ratio=0.1):  
 base = list(range(length))  
 k = int(length \* random\_ratio)  
 base[-k:] = generate\_random\_int\_array(k)  
 return base  
def generate\_insert\_random\_array(length, insert\_count=10):  
 base = list(range(length))  
 for \_ in range(insert\_count):  
 base.insert(random.randint(0, len(base)), random.randint(0, 1000))  
 return base

Time warapper:

import time  
import copy  
def measure\_time(sort\_func, arr, repeat\_count=None, min\_total\_time=0.1):  
 total\_time = 0  
 repeats = 0  
 if repeat\_count is not None:  
 for \_ in range(repeat\_count):  
 test\_arr = copy.deepcopy(arr)  
 start = time.perf\_counter\_ns()  
 sort\_func(test\_arr)  
 total\_time += (time.perf\_counter\_ns() - start) / 1e9  
 return total\_time / repeat\_count  
 while total\_time < min\_total\_time:  
 test\_arr = copy.deepcopy(arr)  
 start = time.perf\_counter\_ns()  
 sort\_func(test\_arr)  
 total\_time += (time.perf\_counter\_ns() - start) / 1e9  
 repeats += 1  
 return total\_time / repeats

Main:

import json  
from utils.generator\_data import \*  
from utils.timer\_wrapper import measure\_time  
from algorithms.bubble\_sort import bubble\_sort  
from algorithms.bucket\_sort import bucket\_sort  
from algorithms.heap\_sort import heap\_sort  
from algorithms.insertion\_sort import insertion\_sort  
from algorithms.intro\_sort import intro\_sort  
from algorithms.least\_significant\_digit\_radix\_sort import radix\_sort\_lsd  
from algorithms.merge\_sort import merge\_sort  
from algorithms.most\_significant\_digit\_radix\_sort import msd\_radix\_sort  
from algorithms.quick\_sort import quick\_sort  
from algorithms.rank\_sort import rank\_sort  
from algorithms.selection\_sort import selection\_sort  
from algorithms.shell\_sort import shell\_sort  
from algorithms.tim\_sort import tim\_sort  
from datetime import datetime  
SLOW\_SORTS = {  
 "bubble": bubble\_sort,  
 "insertion": insertion\_sort,  
 "selection": selection\_sort,  
 "rank": rank\_sort,  
}  
FAST\_SORTS = {  
 "intro": intro\_sort,  
 "bucket": bucket\_sort,  
 "heap": heap\_sort,  
 "lsd\_radix": radix\_sort\_lsd,  
 "merge": merge\_sort,  
 "msd\_radix": msd\_radix\_sort,  
 "quick": quick\_sort,  
 "shell": shell\_sort,  
 "tim": tim\_sort,  
}  
array\_generators = {  
 "random\_int": generate\_random\_int\_array,  
 "reverse\_sorted": generate\_reverse\_sorted\_array,  
 "almost\_sorted": generate\_almost\_sorted\_array,  
 "few\_unique": generate\_few\_unique\_array,  
 "random\_string": generate\_random\_string\_array,  
 "random\_bytes": generate\_random\_bytes\_array,  
 "random\_date": lambda l: generate\_random\_date\_array(l, datetime(2000, 1, 1), datetime(2025, 1, 1)),  
 "tail\_random": lambda l: generate\_tail\_random\_array(l),  
 "insert\_random": lambda l: generate\_insert\_random\_array(l),  
}  
results = {}  
for name, gen\_func in array\_generators.items():  
 print(f"\nТест: {name}")  
 lengths = [10, 100, 1000, 10000, 100000]  
 for length in lengths:  
 print(f"\nТест: {name} | длина: {length}")  
 if name == "random\_bytes":  
 arr = gen\_func(length)  
 key\_func = lambda x: x  
 else:  
 arr = gen\_func(length)  
 key\_func = None  
 results[f"{name}\_{length}"] = {}  
 current\_sorts = FAST\_SORTS if length > 1000 else {\*\*FAST\_SORTS, \*\*SLOW\_SORTS}  
 for sort\_name, sort\_func in current\_sorts.items():  
 try:  
 if key\_func:  
 avg\_time = measure\_time(lambda a: sort\_func(a, key=key\_func), arr.copy())  
 else:  
 repeat\_count = 50 if length <= 1000 else 1  
 avg\_time = measure\_time(sort\_func, arr.copy(), repeat\_count=repeat\_count)  
 time\_str = f"{avg\_time:.9f}".rstrip('0').rstrip('.') if avg\_time < 1 else f"{avg\_time:.3f}"  
 print(f"{sort\_name:15} | {time\_str} сек")  
 results[f"{name}\_{length}"][sort\_name] = avg\_time  
 except Exception as e:  
 print(f"{sort\_name:15} | ОШИБКА: {e}")  
 results[f"{name}\_{length}"][sort\_name] = f"error: {str(e)}"  
with open("results.json", "w") as f:  
 json.dump(results, f, indent=4, default=str)

Plot result:

import json  
import os  
from math import log  
os.makedirs("charts", exist\_ok=True)  
with open("results.json", "r") as f:  
 results = json.load(f)  
import os  
import matplotlib.pyplot as plt  
from collections import defaultdict  
def plot\_time\_vs\_size():  
 size\_data = defaultdict(lambda: defaultdict(list))  
 for test\_name, sort\_results in results.items():  
 if "\_" not in test\_name:  
 continue  
 data\_type, size = test\_name.rsplit("\_", 1)  
 size = int(size)  
 for sort\_name, time in sort\_results.items():  
 if isinstance(time, (float, int)):  
 time\_microseconds = time \* 1\_000\_000  
 size\_data[sort\_name][data\_type].append((size, time\_microseconds))  
 for sort\_name, data\_types in size\_data.items():  
 plt.figure(figsize=(10, 6))  
 for data\_type, points in data\_types.items():  
 sizes, times = zip(\*sorted(points))  
 plt.plot(sizes, times, "o-", label=data\_type)  
  
 plt.xlabel("Размер массива")  
 plt.ylabel("Время (мкс)")  
 plt.title(f"Производительность {sort\_name}")  
 plt.xscale("log")  
 plt.yscale("log")  
 plt.legend()  
 plt.grid(True, which="both", ls="--")  
 os.makedirs("charts", exist\_ok=True)  
 filename = os.path.join("charts", f"time\_vs\_size\_{sort\_name}.png")  
 plt.savefig(filename, bbox\_inches="tight")  
 plt.close()  
def plot\_relative\_performance():  
 FAST\_SORTS = {  
 "intro", "bucket", "heap", "lsd\_radix",  
 "merge", "msd\_radix", "quick", "shell", "tim"  
 }  
 for data\_type in ["random\_int", "almost\_sorted", "reverse\_sorted", "few\_unique"]:  
 plt.figure(figsize=(12, 6))  
 size\_data = defaultdict(list)  
 perf\_data = defaultdict(list)  
 for test\_name, sort\_results in results.items():  
 if not test\_name.startswith(data\_type):  
 continue  
 size = int(test\_name.split("\_")[-1])  
 for sort\_name, time in sort\_results.items():  
 if sort\_name in FAST\_SORTS and isinstance(time, (float, int)):  
 size\_data[sort\_name].append(size)  
 perf\_data[sort\_name].append(time)  
 if not size\_data:  
 continue  
 common\_sizes = None  
 for sort\_name, sizes in size\_data.items():  
 if common\_sizes is None:  
 common\_sizes = set(sizes)  
 else:  
 common\_sizes.intersection\_update(sizes)  
 if not common\_sizes:  
 continue  
 common\_sizes = sorted(common\_sizes)  
 filtered\_perf\_data = defaultdict(list)  
 for sort\_name in perf\_data:  
 for size, time in zip(size\_data[sort\_name], perf\_data[sort\_name]):  
 if size in common\_sizes:  
 filtered\_perf\_data[sort\_name].append(time)  
 min\_times = []  
 for i in range(len(common\_sizes)):  
 min\_time = min(times[i] for times in filtered\_perf\_data.values())  
 min\_times.append(min\_time)  
 normalized\_data = {}  
 for sort\_name, times in filtered\_perf\_data.items():  
 normalized\_data[sort\_name] = [t / min\_t for t, min\_t in zip(times, min\_times)]  
 for sort\_name, rel\_times in normalized\_data.items():  
 plt.plot(common\_sizes, rel\_times, "o-", label=sort\_name)  
 plt.xlabel("Размер массива")  
 plt.ylabel("Относительное время")  
 plt.title(f"Относительная производительность (только быстрые сортировки: {data\_type})")  
 plt.xscale("log")  
 plt.legend()  
 plt.grid(True)  
 filename = os.path.join("charts", f"relative\_perf\_fast\_{data\_type}.png")  
 plt.savefig(filename, bbox\_inches="tight")  
 plt.close()  
def check\_complexity():  
 fast\_sorts = ["intro", "bucket", "heap", "lsd\_radix", "merge", "msd\_radix", "quick", "shell", "tim"]  
 data\_type = "random\_int"  
 plt.figure(figsize=(10, 6))  
  
 for sort\_name in fast\_sorts:  
 sizes = []  
 times = []  
 for test\_name, sort\_results in results.items():  
 if not test\_name.startswith(data\_type):  
 continue  
 size = int(test\_name.split("\_")[-1])  
 if sort\_name in sort\_results and isinstance(sort\_results[sort\_name], (float, int)):  
 sizes.append(size)  
 times.append(sort\_results[sort\_name])  
 if not sizes:  
 continue  
 n\_log\_n = [n \* log(n) for n in sizes]  
 normalized\_times = [t / nlogn for t, nlogn in zip(times, n\_log\_n)]  
 plt.plot(sizes, normalized\_times, "o-", label=sort\_name)  
 plt.xlabel("Размер массива")  
 plt.ylabel("Время / (n log n)")  
 plt.title("Проверка сложности O(n log n)")  
 plt.xscale("log")  
 plt.legend()  
 plt.grid(True, which="both", ls="--")  
 filename = os.path.join("charts", "complexity\_check.png")  
 plt.savefig(filename, bbox\_inches="tight")  
 plt.close()  
def plot\_basic\_charts():  
 for test\_name, sort\_results in results.items():  
 labels = []  
 times = []  
 errors = []  
 for sort\_name, value in sort\_results.items():  
 if isinstance(value, (float, int)):  
 labels.append(sort\_name)  
 times.append(value \* 1\_000\_000)  
 else:  
 errors.append(sort\_name)  
  
 plt.figure(figsize=(12, 6))  
 bars = plt.bar(labels, times, color="skyblue")  
 for bar, time in zip(bars, times):  
 height = bar.get\_height()  
 plt.text(bar.get\_x() + bar.get\_width() / 2, height,  
 f"{height:.1f}", ha="center", va="bottom", fontsize=8)  
 plt.xlabel("Алгоритм сортировки")  
 plt.ylabel("Время (мкс)")   
 plt.title(f"Сравнение сортировок: {test\_name}")  
 plt.xticks(rotation=45)  
 plt.tight\_layout()  
 filename = os.path.join("charts", f"chart\_{test\_name}.png")  
 plt.savefig(filename)  
 plt.close()  
def plot\_slow\_sorts\_only():  
 SLOW\_SORTS = {"bubble", "insertion", "selection", "rank"}  
 for test\_name, sort\_results in results.items():  
 labels = []  
 times = []  
 for sort\_name, value in sort\_results.items():  
 if sort\_name in SLOW\_SORTS and isinstance(value, (float, int)):  
 labels.append(sort\_name)  
 times.append(value \* 1\_000\_000)  
 if not labels:  
 continue  
 plt.figure(figsize=(10, 5))  
 bars = plt.bar(labels, times, color="salmon")  
 for bar, time in zip(bars, times):  
 height = bar.get\_height()  
 plt.text(bar.get\_x() + bar.get\_width() / 2, height,  
 f"{height:.1f}", ha="center", va="bottom", fontsize=8)  
 plt.xlabel("Медленные сортировки")  
 plt.ylabel("Время (мкс)")  
 plt.title(f"Только медленные сортировки: {test\_name}")  
 plt.xticks(rotation=45)  
 plt.tight\_layout()  
 filename = os.path.join("charts", f"slow\_chart\_{test\_name}.png")  
 plt.savefig(filename)  
 plt.close()  
def plot\_fast\_sorts\_only():  
 FAST\_SORTS = {  
 "intro", "bucket", "heap", "lsd\_radix",  
 "merge", "msd\_radix", "quick", "shell", "tim"  
 }  
 for test\_name, sort\_results in results.items():  
 labels = []  
 times = []  
 for sort\_name, value in sort\_results.items():  
 if sort\_name in FAST\_SORTS and isinstance(value, (float, int)):  
 labels.append(sort\_name)  
 times.append(value \* 1\_000\_000)  
 if not labels:  
 continue  
 plt.figure(figsize=(12, 6))  
 bars = plt.bar(labels, times, color="mediumseagreen")  
 for bar, time in zip(bars, times):  
 height = bar.get\_height()  
 plt.text(bar.get\_x() + bar.get\_width() / 2, height,  
 f"{height:.1f}", ha="center", va="bottom", fontsize=8)  
 plt.xlabel("Быстрые сортировки")  
 plt.ylabel("Время (мкс)")  
 plt.title(f"Только быстрые сортировки: {test\_name}")  
 plt.xticks(rotation=45)  
 plt.tight\_layout()  
 filename = os.path.join("charts", f"fast\_chart\_{test\_name}.png")  
 plt.savefig(filename)  
 plt.close()  
if \_\_name\_\_ == "\_\_main\_\_":  
 plot\_basic\_charts()  
 plot\_time\_vs\_size()  
 plot\_relative\_performance()  
 check\_complexity()  
 plot\_slow\_sorts\_only()  
 plot\_fast\_sorts\_only()