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: Ахметов Камиль Ильшатович гр. 09–232

(ФИО студента) (Группа)

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

Казань – 2024

**Оглавление**

[Введение 3](#_Toc162345006)

[Постановка задачи 4](#_Toc162345007)

[Методика проведения эксперимента 5](#_Toc162345008)

[Полученные результаты 6](#_Toc162345009)

[Заключение 7](#_Toc162345010)

[Приложение 1. Программный код 8](#_Toc162345011)

Введение

Задача сортировки данных является одной из наиболее изученных задач в области компьютерных наук. За все время развития этой науки было создано огромное количество различный алгоритмов сортировки. У каждого из них есть свои преимущества и недостатки, однако нельзя выбрать из всех алгоритмов один, который был бы лучшим при любой ситуации. Какие то алгоритмы работают быстрее на одном типе данных, а на другом медленнее и т.д.

В данной работе я проведу сравнительный анализ наиболее известных и интересных к рассмотрению алгоритмов сортировок и сделаю анализ полученных результатов.

Для выполнения проекта я выбрал язык python на котором напишу все выбранные сортировки и нарисую необходимые графики для дальнейшего анализа.

Постановка задачи

1) Необходимо реализовать следующие сортировки:

* Пузырьковую
* Вставками
* Коктейльную
* Блинную
* Выбором
* Сортировку кучей
* Интроспективную
* Поразрядную (LSD)
* Поразрядную (MSD)
* Слиянием
* Быструю
* Сортировку Шелла
* Встроенную в язык (Tim Sort)

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) Сначала я написал все нужные мне сортировки трех видов:*

* Медленные сортировки
* Быстрые сортировки
* Поразрядные сортировки

Поразрядную сортировку я выполнил в двух видах: LSD и MSD.

Встроенная в язык Python сортировка написана на c++, поэтому я написал этот же алгоритм TimSort на языке Python.

Код всех сортировок в приложении 1.

*2) После этого я написал функцию обертку которую можно использовать для любой сортировки.*

Задачи обертки:

* Проведение серии испытаний с нахождением среднего для достоверности результатов (Например 10 повторов, 50 и тд)
* Замер времени
* Создание копии массива перед его изменением

Код обертки в приложении 1.

*3) Написал функции для автоматической генерации тестовых данных.*

Типы данных для которых написал генерацию:

* Byte (Цифры 0-9)
* Int
* String
* Structure\*

Также написал функции для генерации разного типа массивов на любых типах данных:

* Полностью случайно сгенерированные массивы
* Массивы с неупорядоченным хвостом
* Массивы с несколькими вставленными элементами в упорядоченный
* Массивы с большой долей одинаковых элементов

\*Structure — созданная структура Student у которой есть поля name и average\_score по которой происходит сравнение объектов этого класса

Код генераторов в приложении 1

*4) Написал config в котором указал максимальный размер массива и типы данных на которых можно запускать каждую сортировку:*

* LSD\_Sort и MSD\_Sort — работают только на одном типе данных
* Для квадратичных сортировок поставил лимит на длину массива 5 тысяч элементов, так как на большем массиве время выполнения слишком большое

*5) Main.py:*

1. Сгенерировал все комбинации массивов и сохранил их в Map
2. Отключил сборщик мусора, потому что с ним получаются недостоверные данные
3. Прошелся в цикле по всем комбинациям сгенерированных массивов и сортировок, вызвал каждую сортировку
4. Итоговые данные объекта Map записал в json файл

*6) Генерация графиков:*

1. Считал данные из results.json
2. Прошелся в цикле по всем комбинациям и сгенерировал графики таких типов:
   1. Гистограммы сортировок на всех типах данных, длинах массивов и типах генерации массивов
   2. Гистограммы в которых объединил тип генерации и сортировки для наглядного представления
   3. 3д гистаграммы в которых объединил тип генерации и сортировки для наглядного представления
   4. Если одновременно есть очень низкие и очень высокие столбцы, я делил их на два разных графика
   5. Все графики сохранил в виде фотографий в папке «results»

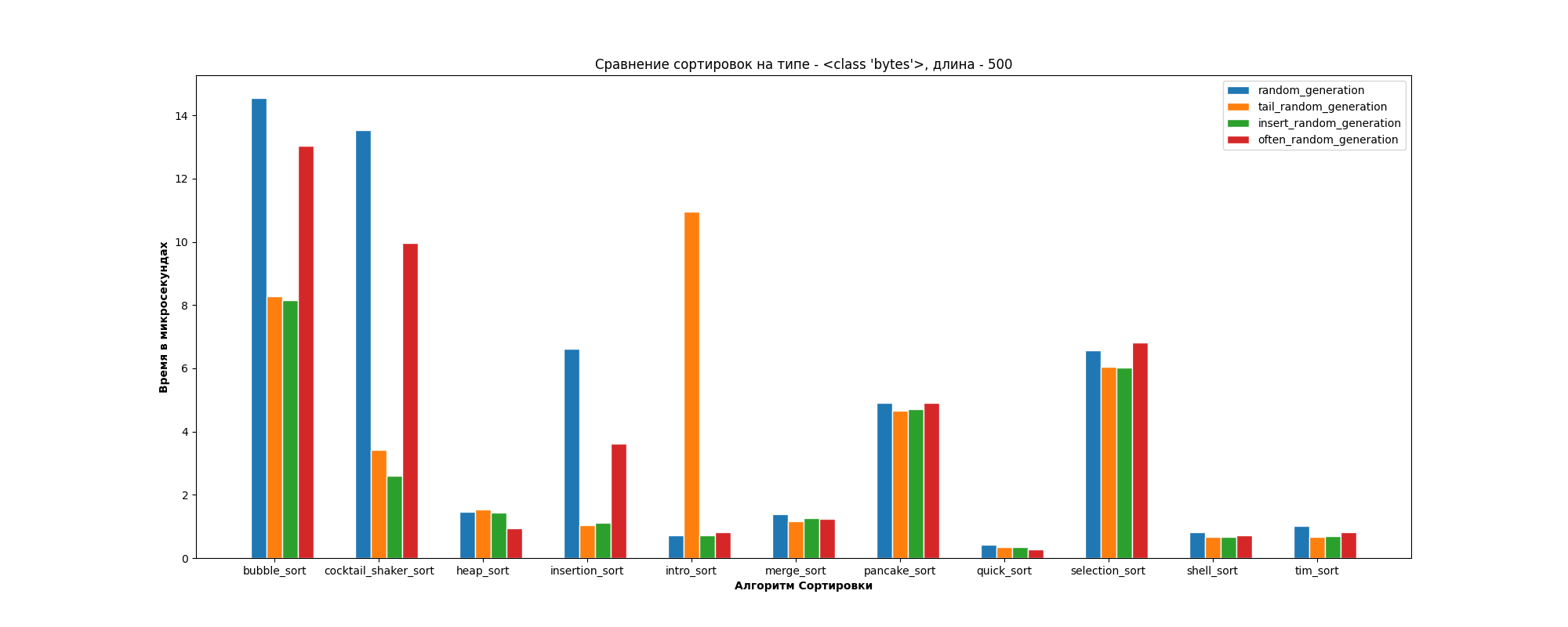
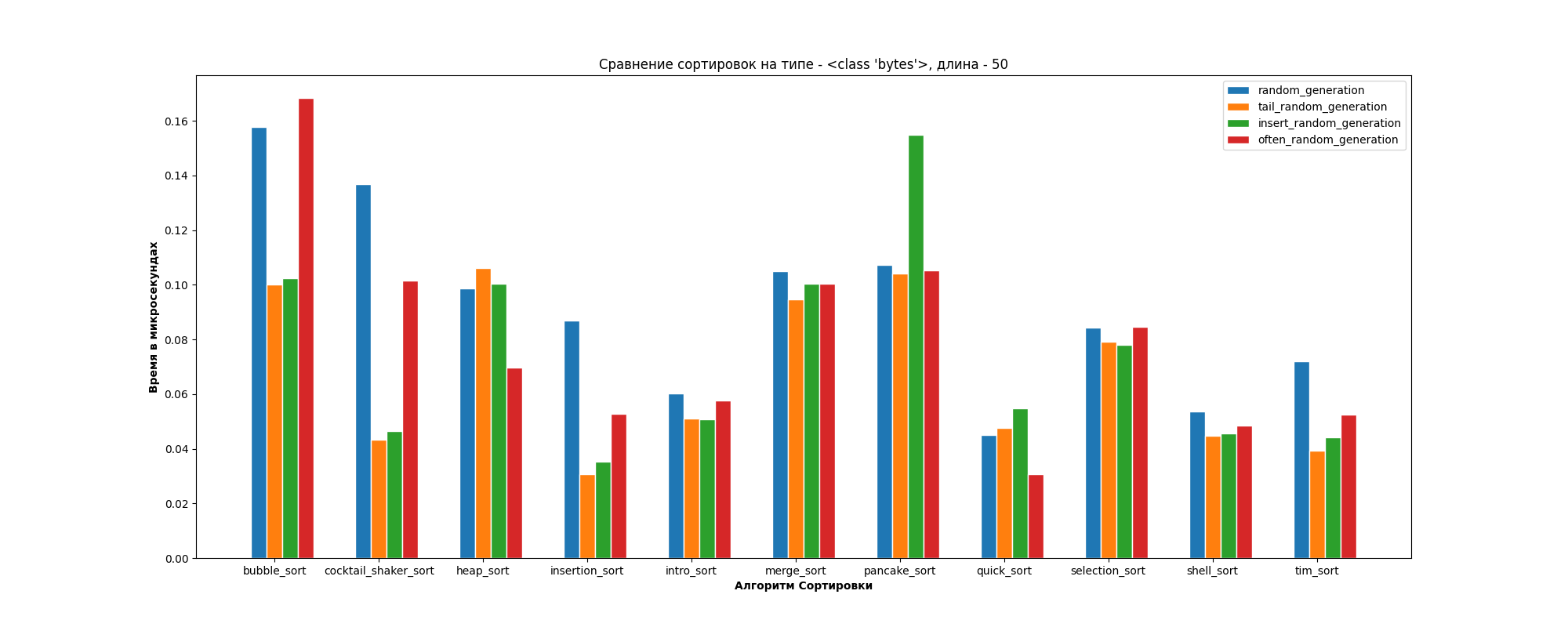
*Замечание:*

Время считал в микро секундах

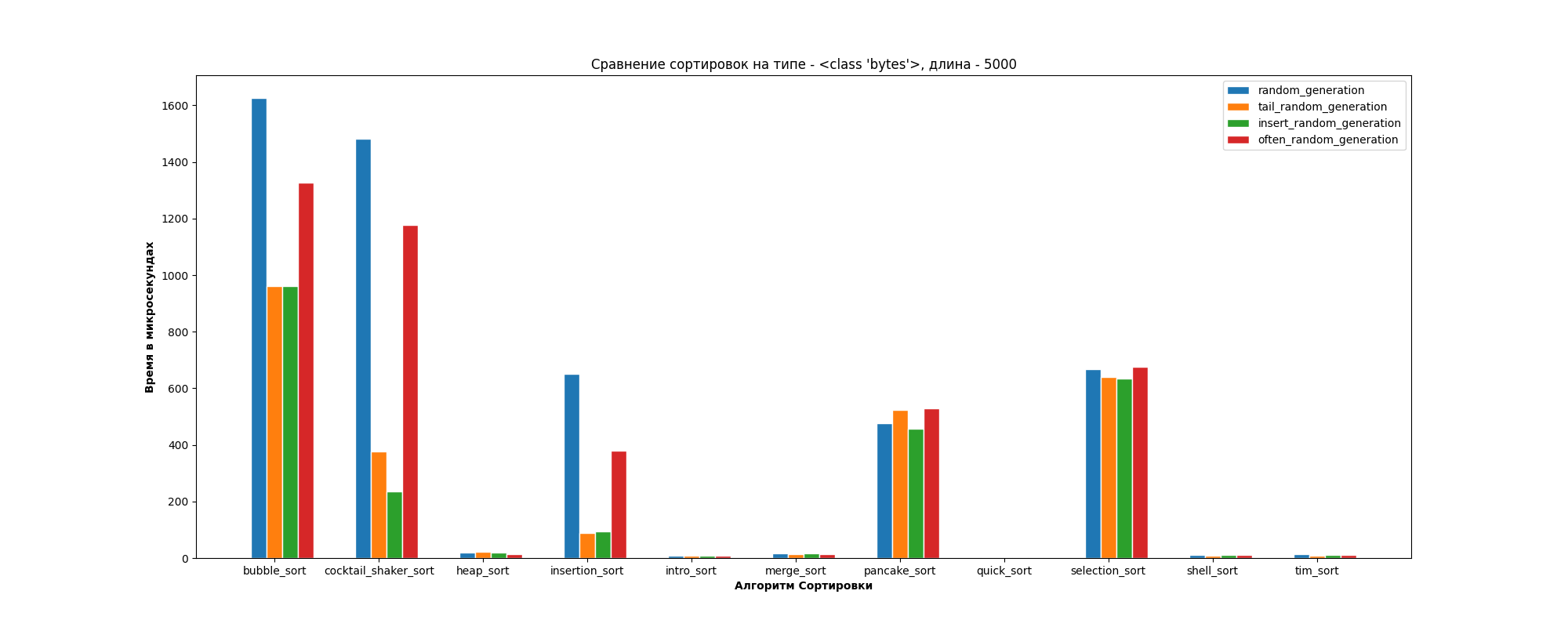
Наиболее удобной визуаализацией оказались 2д гистаграммы

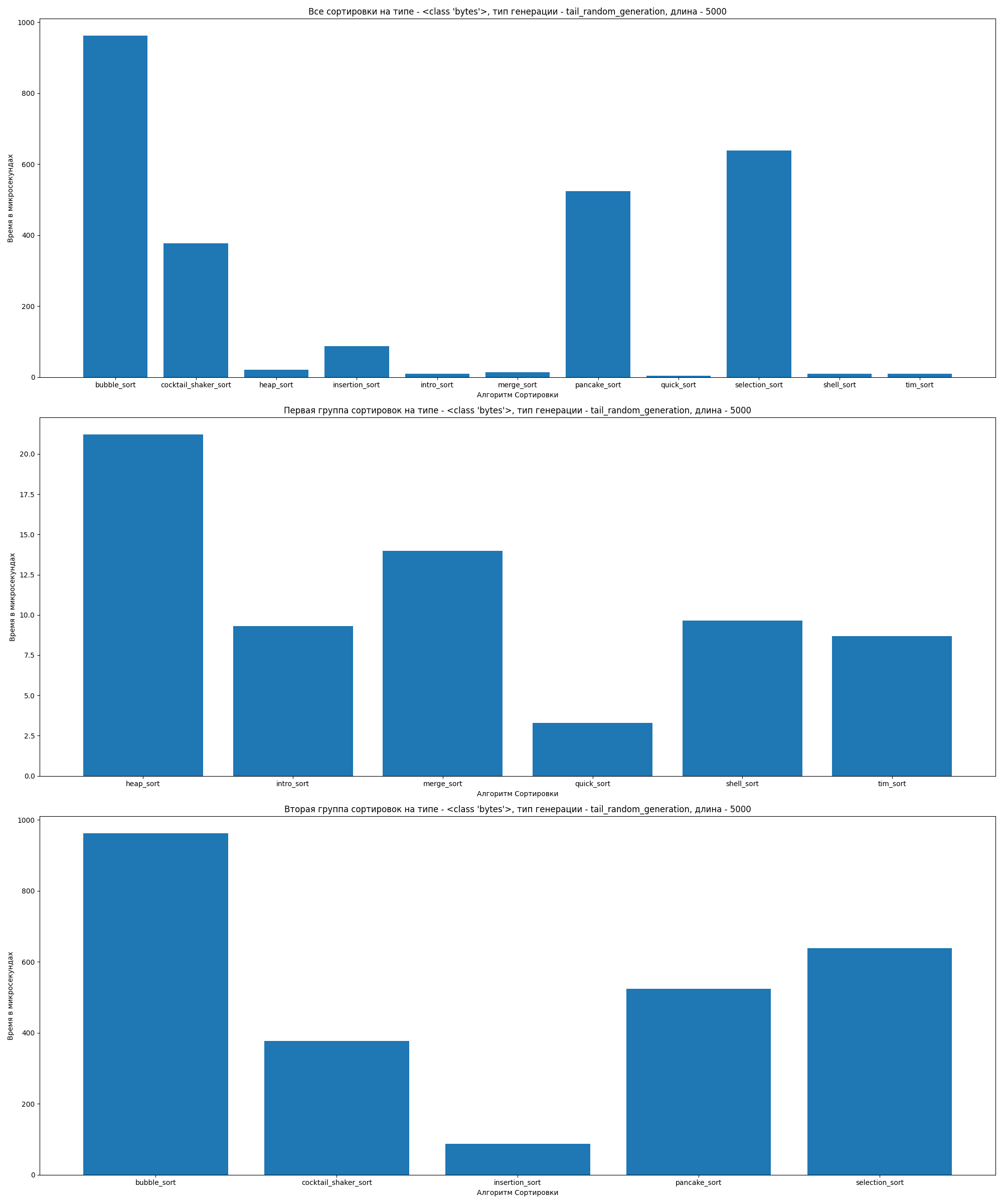
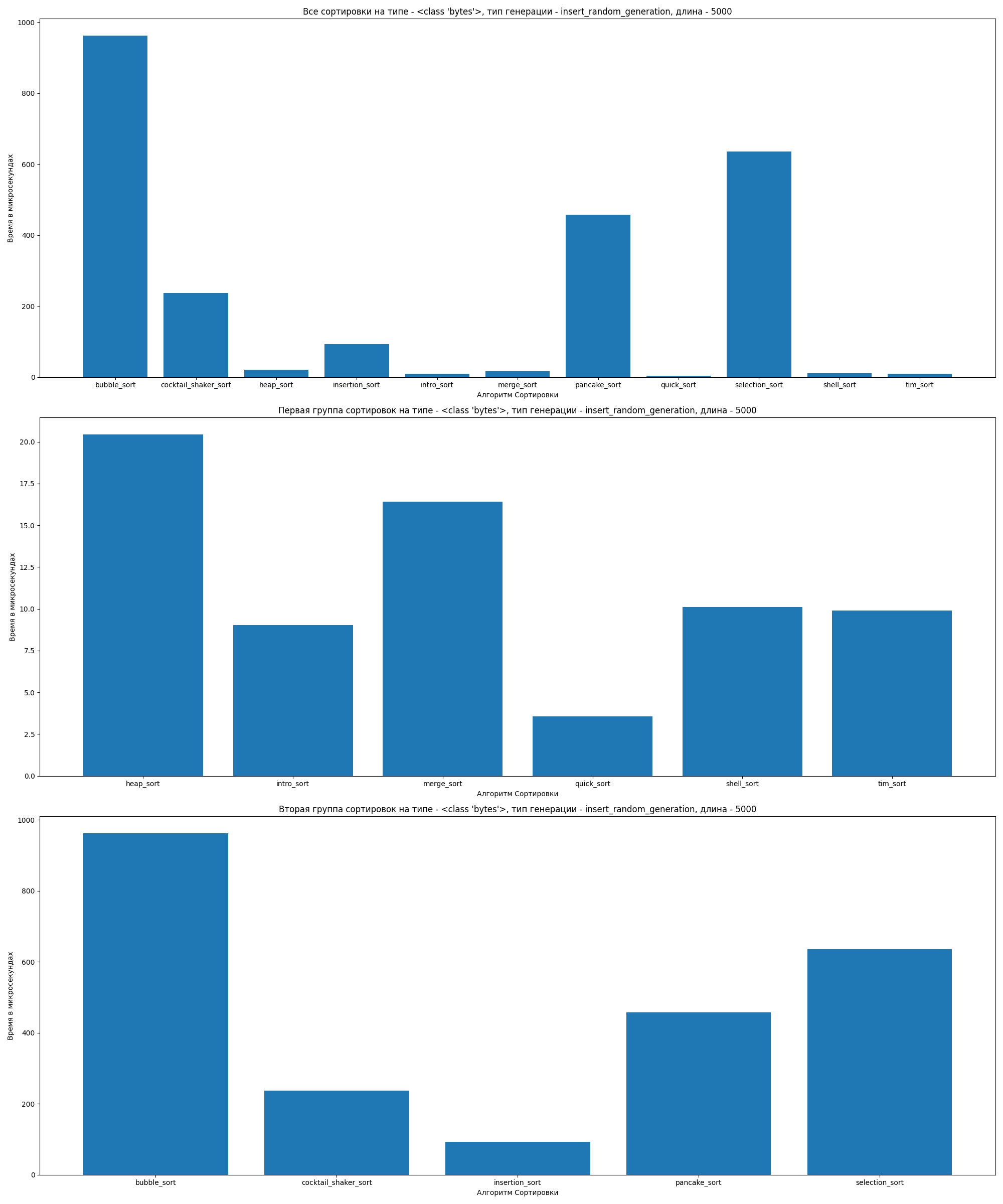
Полученные результаты

Начнем анализ на типе данных Byte



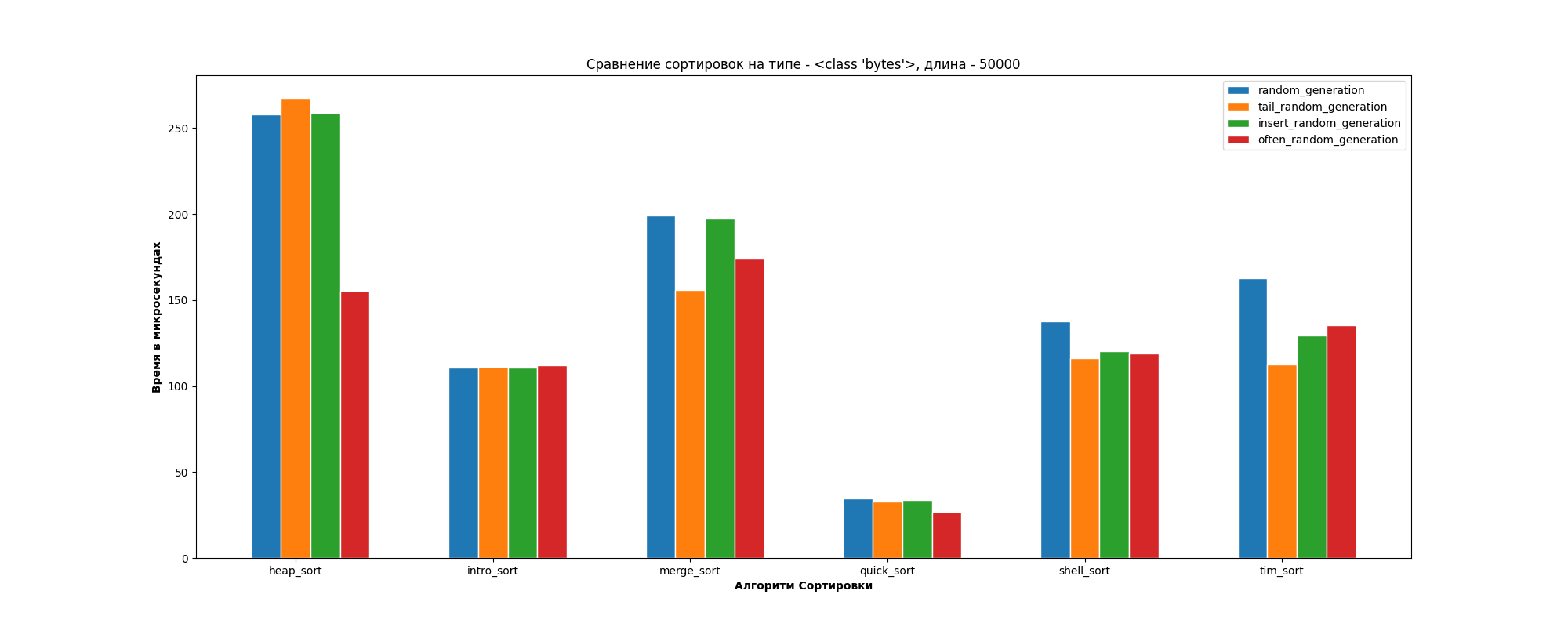
Как можно заметить на небольшом количестве данных (50 — 500) разницы в работе сортировок почти нет, поэтому можно перейти к 5к данных

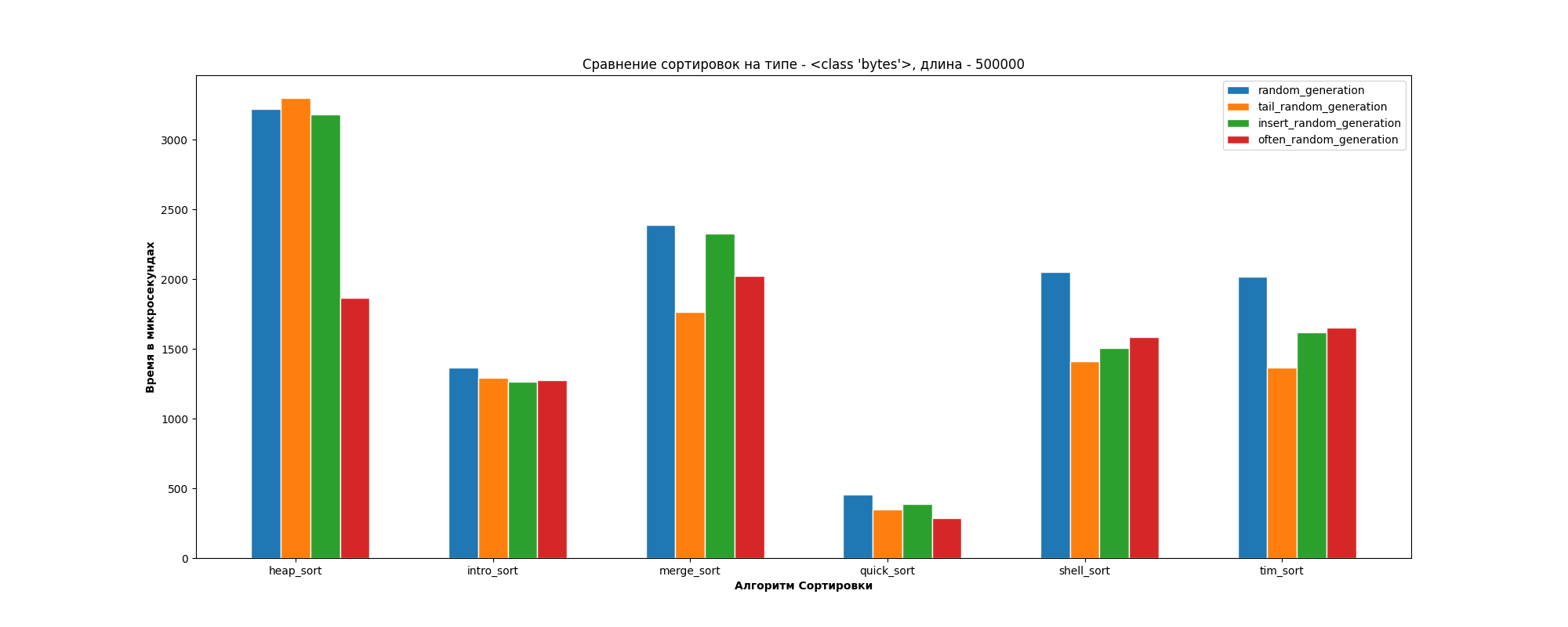
На 5 тысячах данных уже видно различие между квадратичными и эффективными сортировками. Время эффективных почти не видно на этом графике поэтому выведем графики двух групп сортировок отдельно



Видно что среди эффективных сортировок независимо от типа генерации массива в лидеры выбивается quick\_sort. Среди простых лучше всего показывает себя insertion\_sort

Далее рассмотрим сортировки на 50к элементов в массиве



На такой длине я не рассматриваю простые сортировки. Не смотря на тип генерации самой быстрой сортировкой все также является quick\_sort.

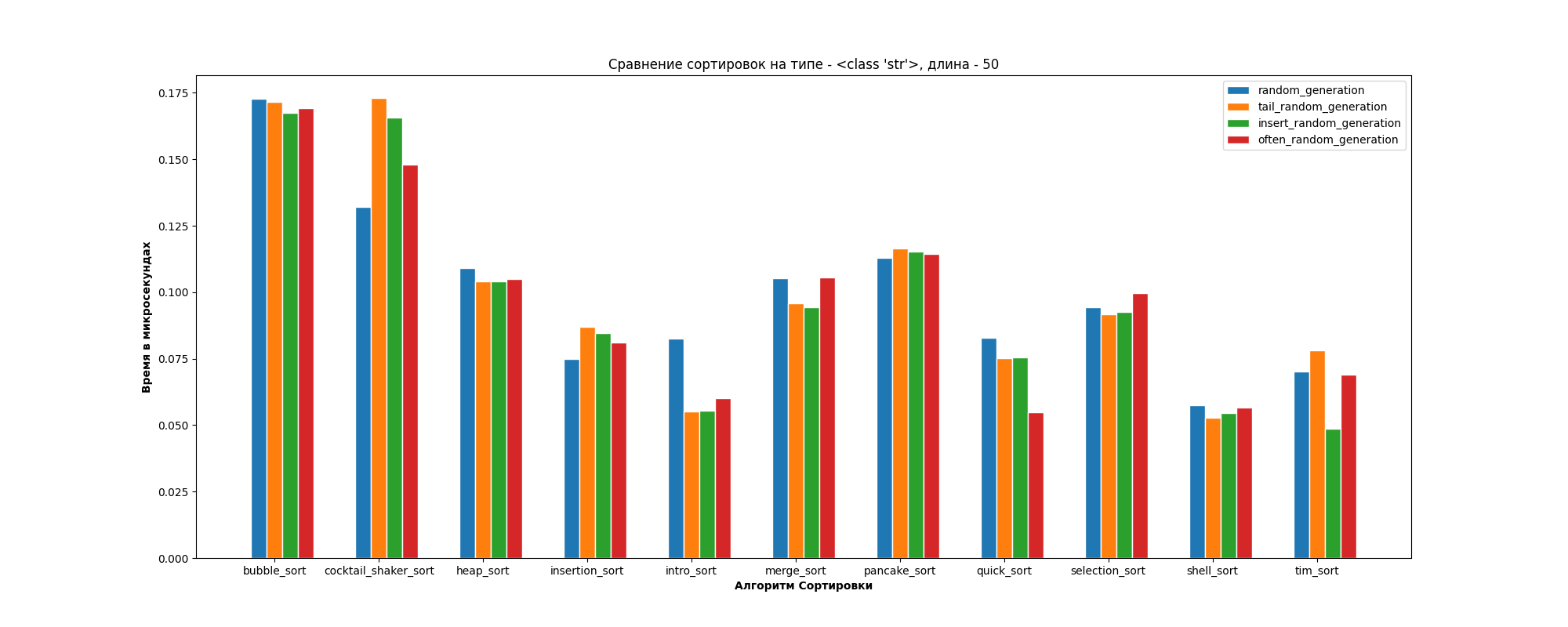
Heap\_sort в свою очередь показывает худшее время, будучи в несколько раз медленнее quick\_sort и остальных

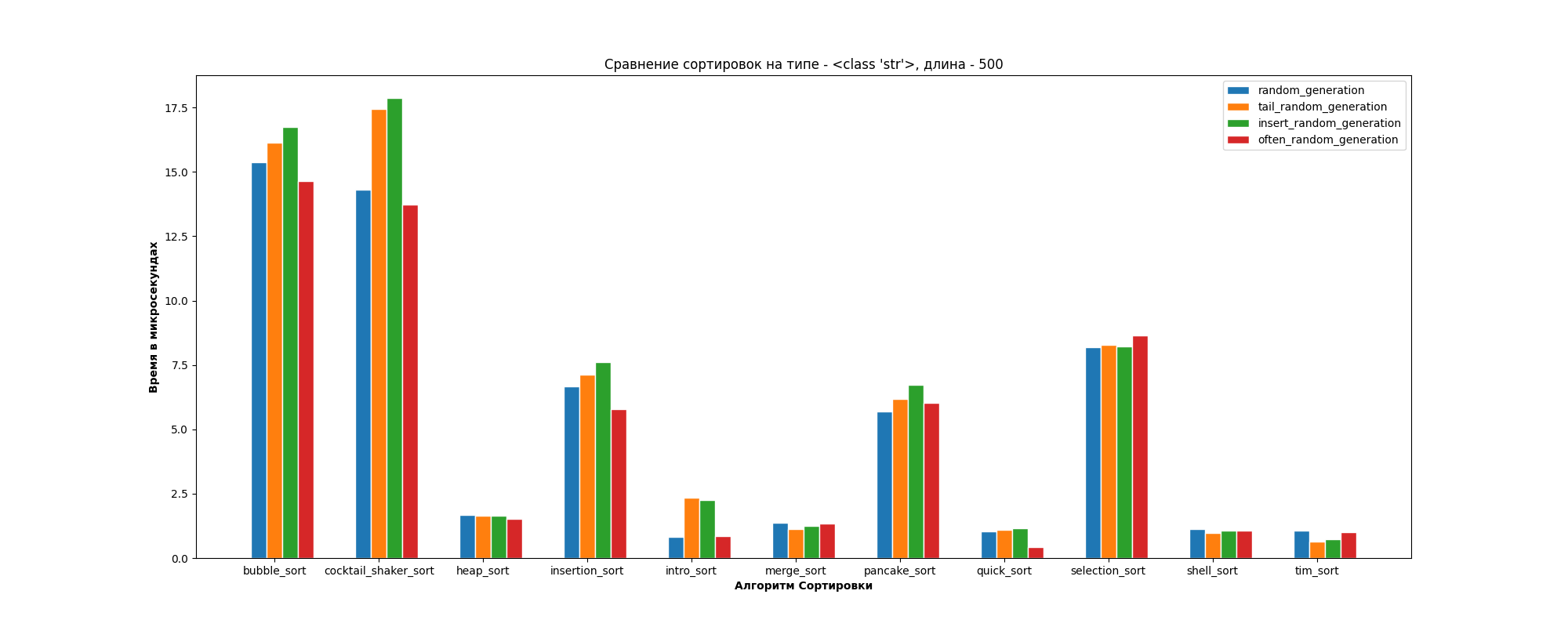
На массивах размером 500к элементов ситуация такая же как и в предыдущем случае. Быстрая сортировка так же побеждает, сортировка кучей отстает.

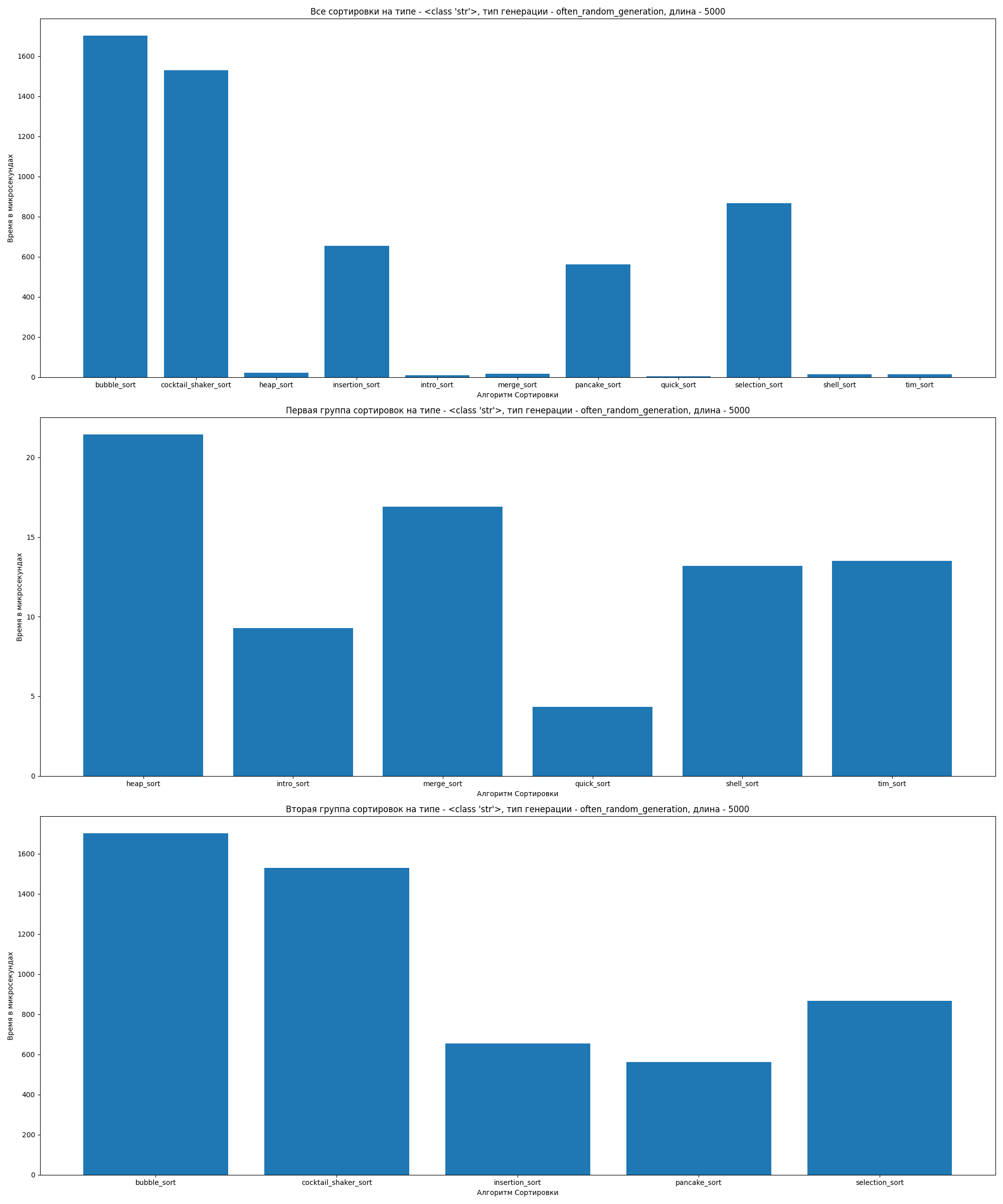
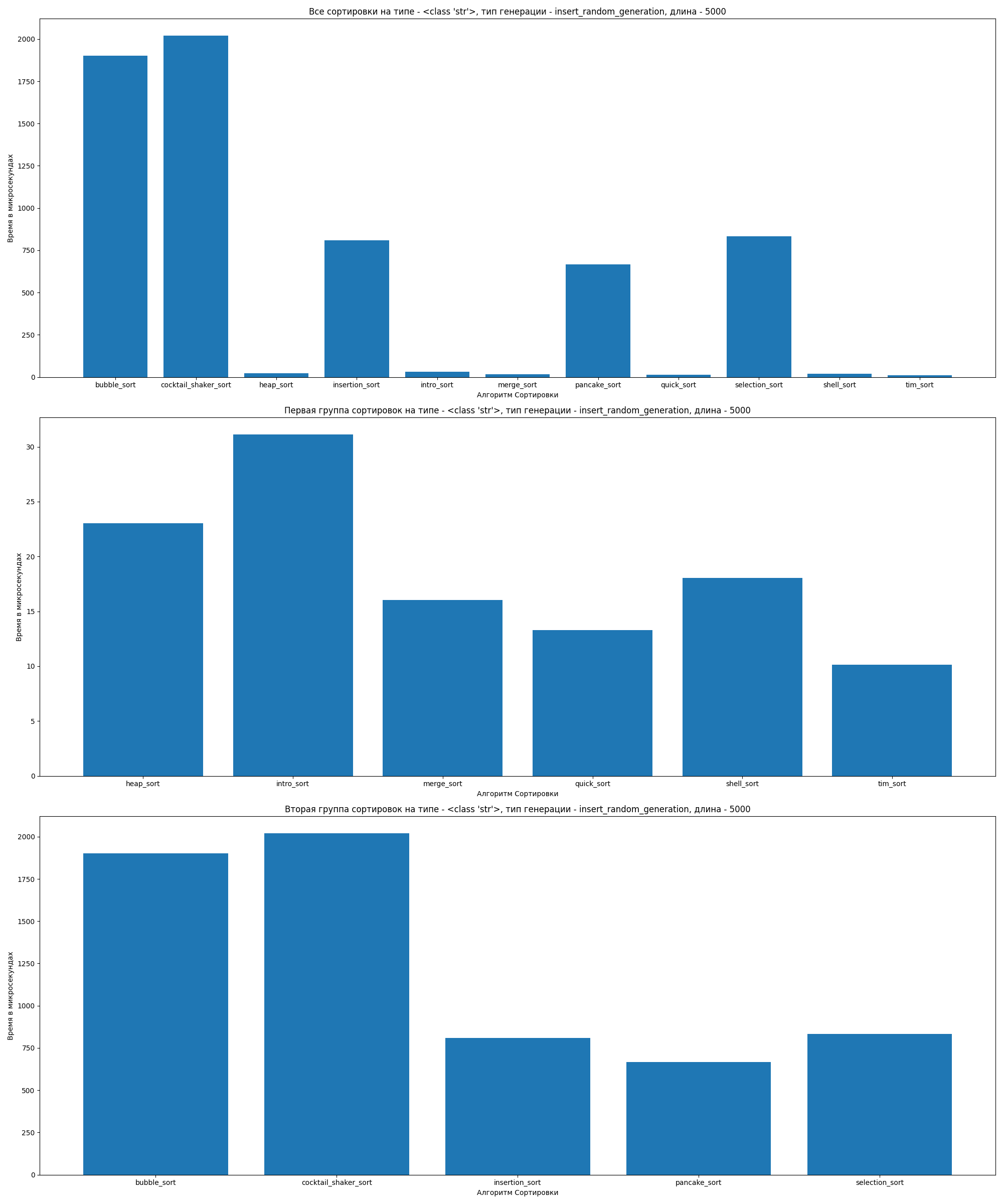
Промежуточный итог на типе данных Byte:

Нет разницы какой тип генерации массива, примерно одинаковое время во всех случаях. Так же можно отметить что quick\_sort всегда побеждает даже встроенную сортировку (tim\_sort). Среди простых сортировок все примерно одинаково но insertion\_sort выбивается вперед.

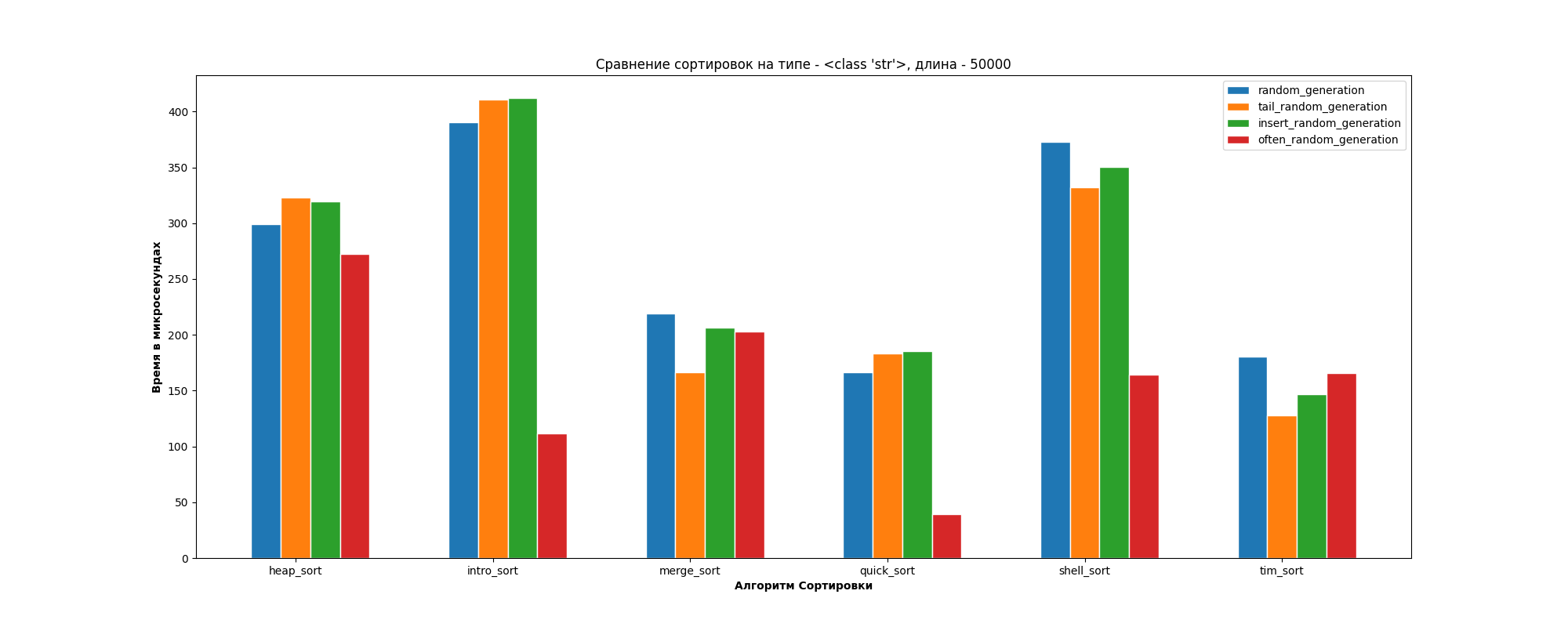
**Следующий тип данных string:**

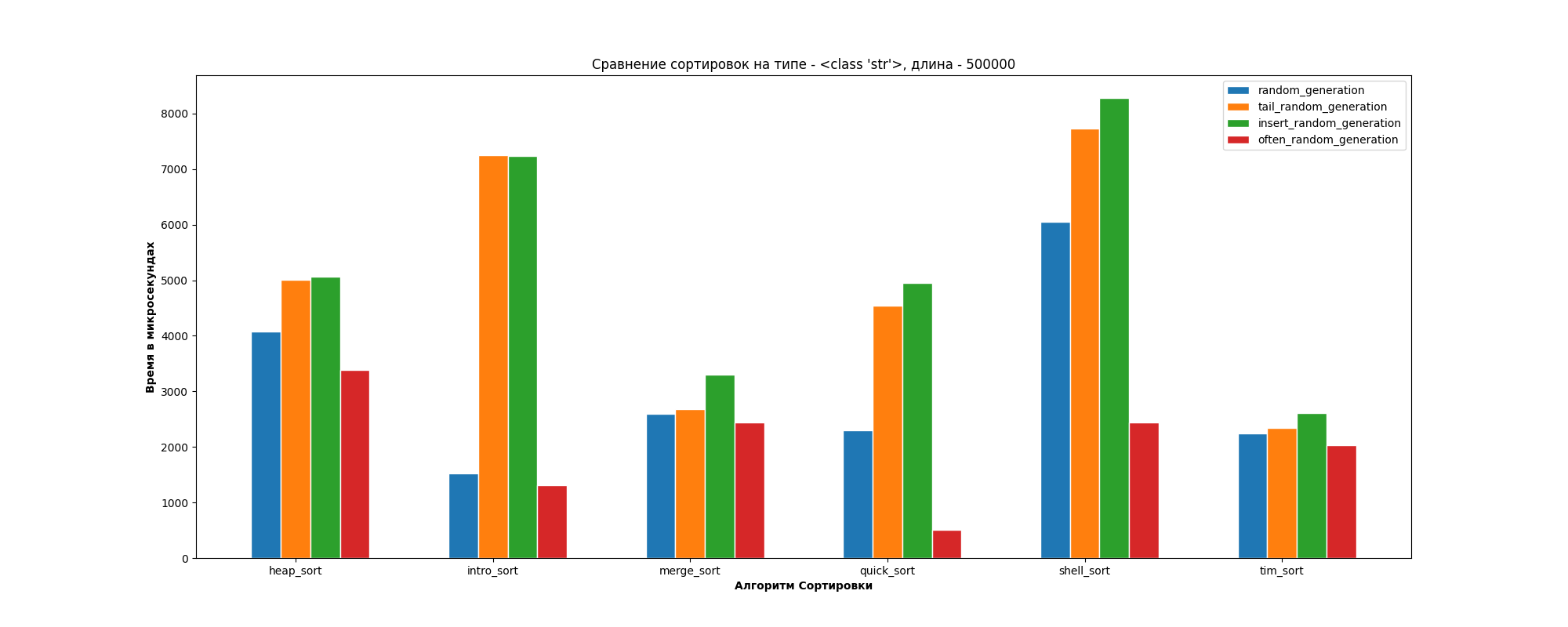
На строках можем заметить что на небольших данных все сортировки показывают один уровень так что переходим к следующему

на 500 данных видно что среди простых сортировок pancake\_sort немного быстрее. А эффективные показывают один уровень.



На 5к данных можно заметить что в зависимости от типа генерации данных алгоритмы показывают разный уровень. Быстрая сортировка лучше всего справляется когда есть много одинаковых элементов в массиве. А tim\_sort в свою очередь показывает лучший результат на массивах со вставленными несколькими элементами в сортированный массив из-за хорошей композиции использующихся алгоритмов алгоритмов

На 50к данных наблюдается тенденция с 5к, все также на often\_random\_arr с большим отрывом побеждает quick\_sort, а в остальных примерно одинаковый уровень у merge\_sort, quick\_sort и tim\_sort. Однако интроспективная работает заметно хуже на всех типах данных кроме often\_random\_arr из-за гибрида алгоритмов которые в нем используются.

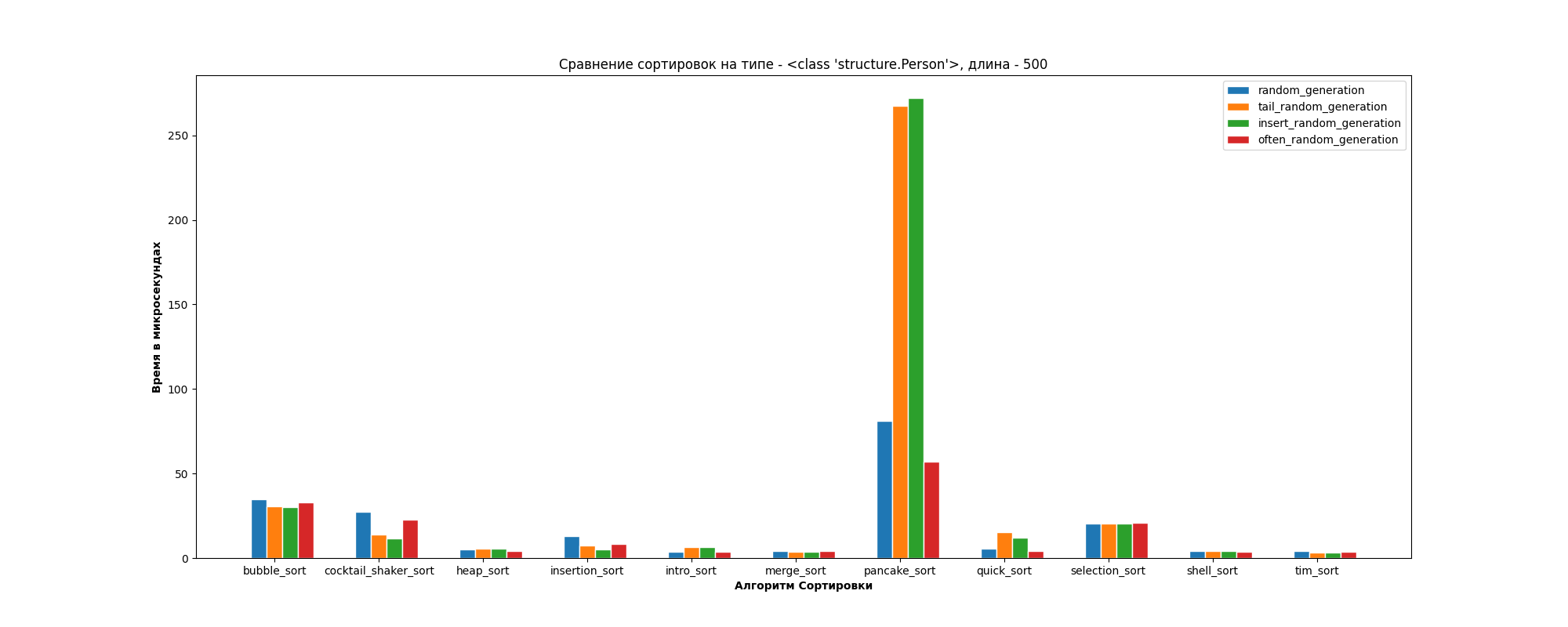
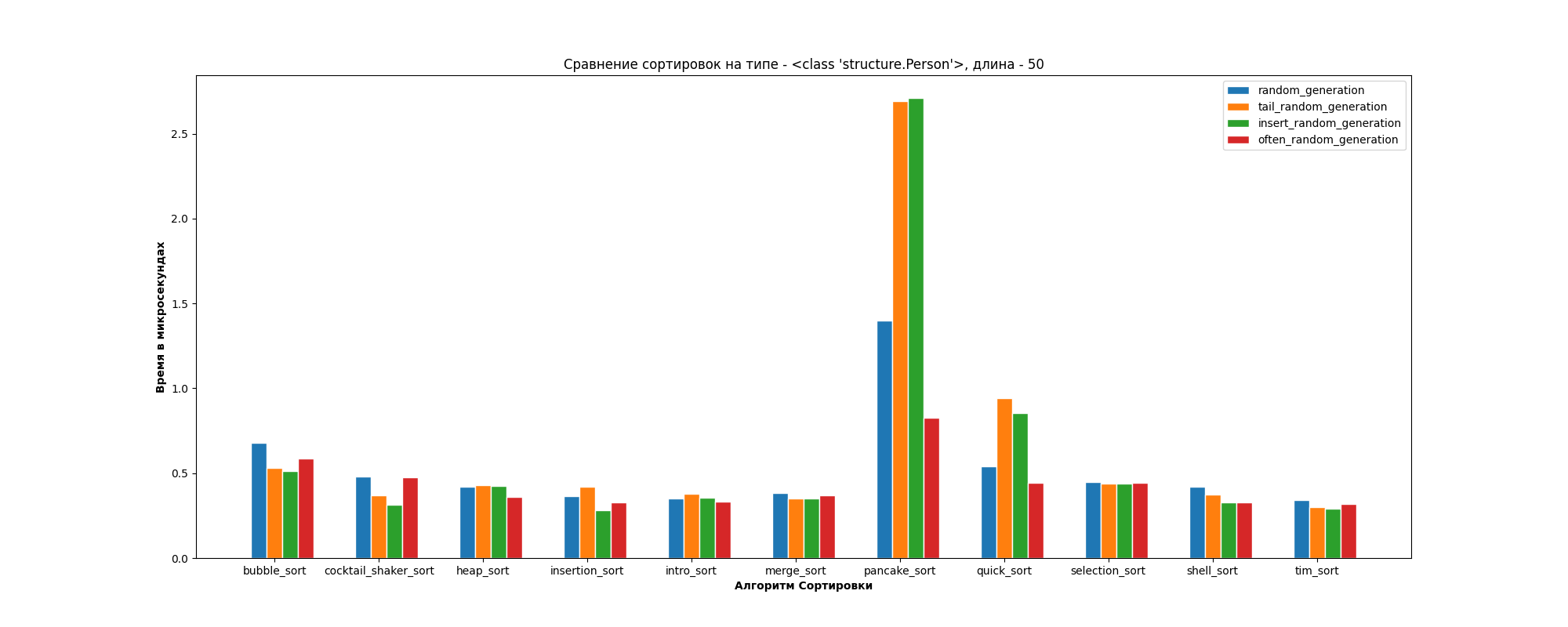
На такой длине массивов заметно что tim\_sort из-заа хорошо подобранных алгоритмов в среднем выбивается вперед. Quick\_sort все также отлично работает только на often\_random\_arr. Также бросается в глаза что интроспективная сортировка построена таким образом, чтобы хорошо работать с полностью рандомными массивами, но у нее проблемы когда только небольшая часть массива не сортирована.

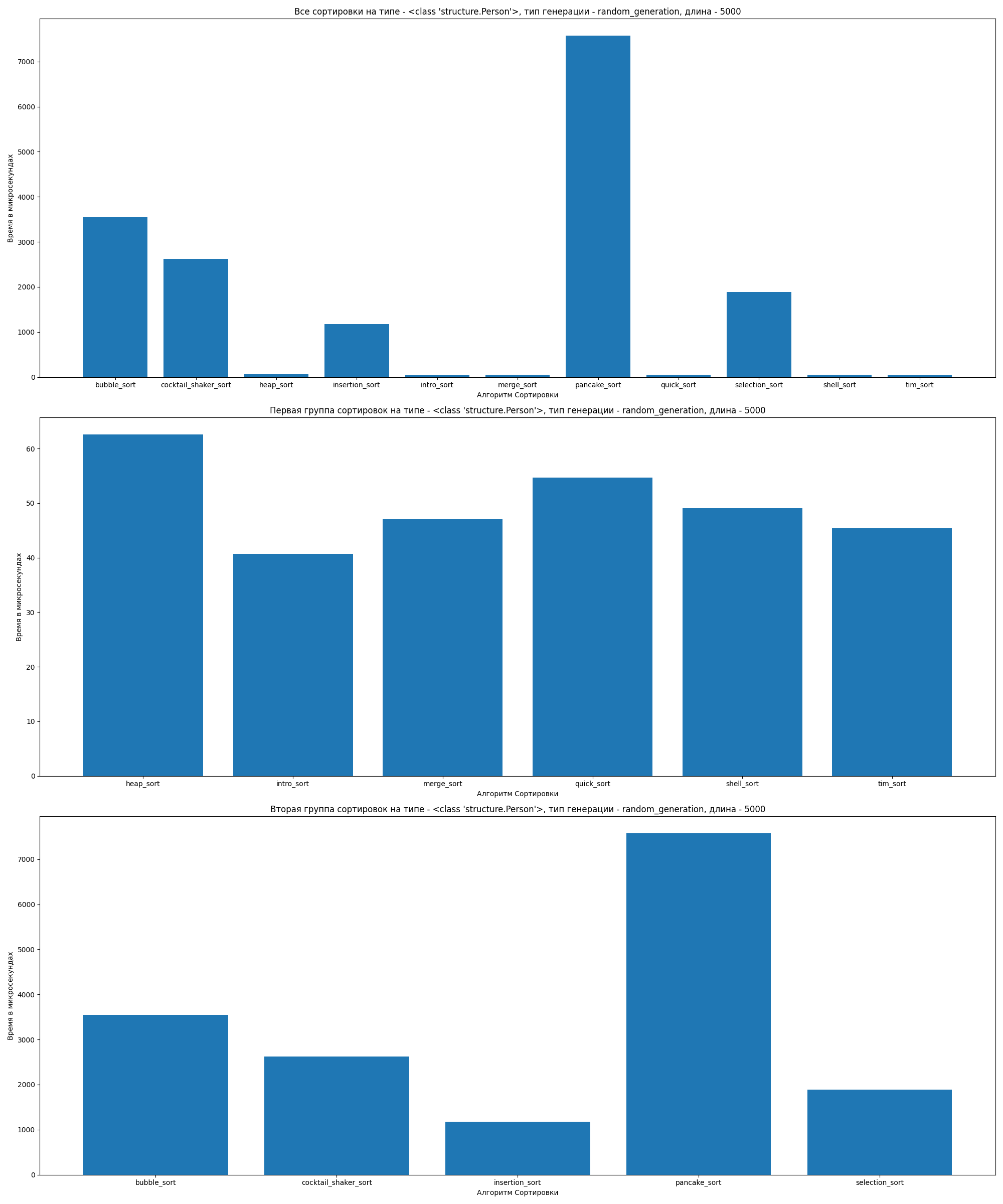
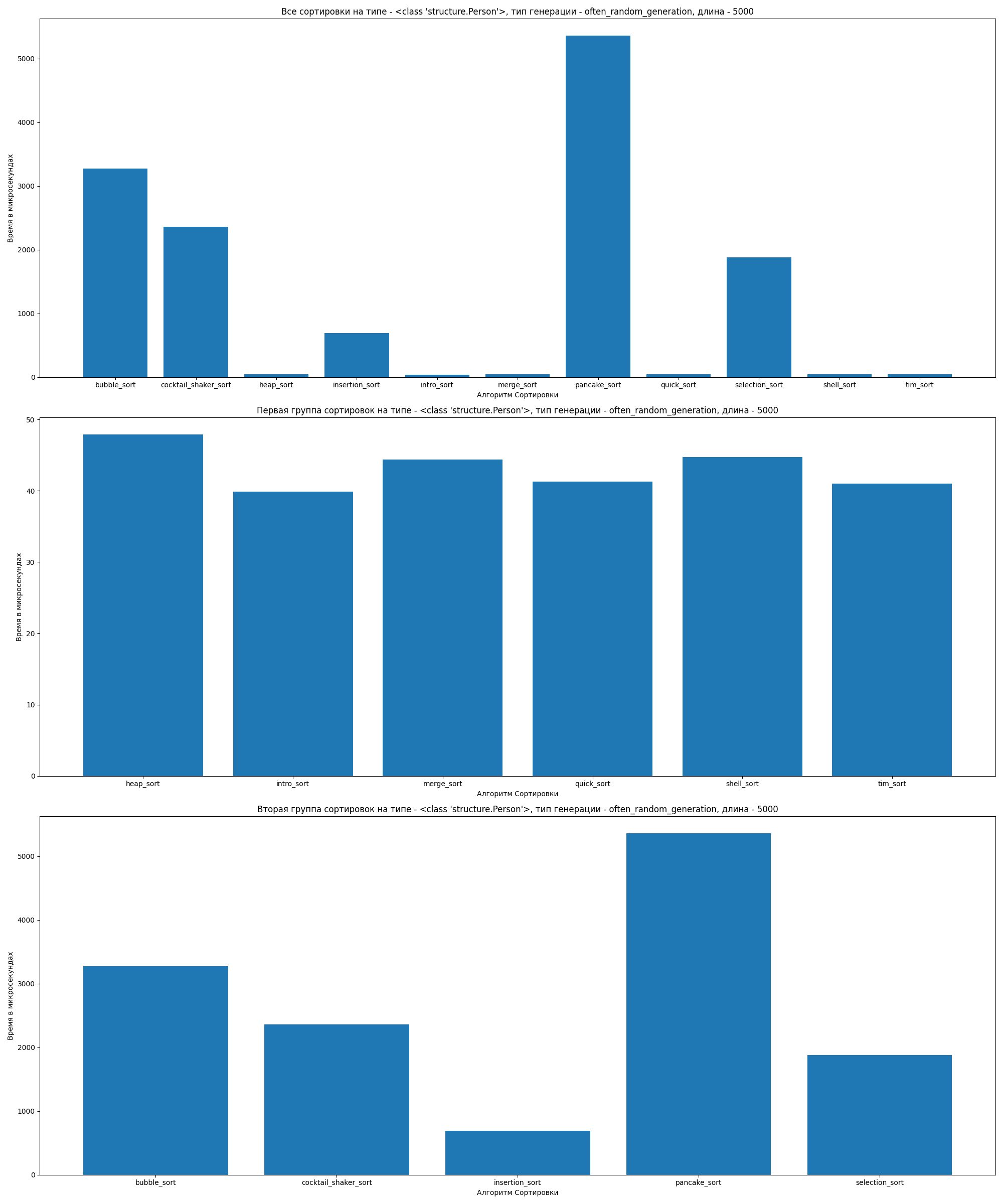
Итог по типу данных string:

Если вам нужно отсортировать массив строк то наиболее правильным вариантом будет взять встроенную сортировку, quick\_sort, либо merge\_sort.

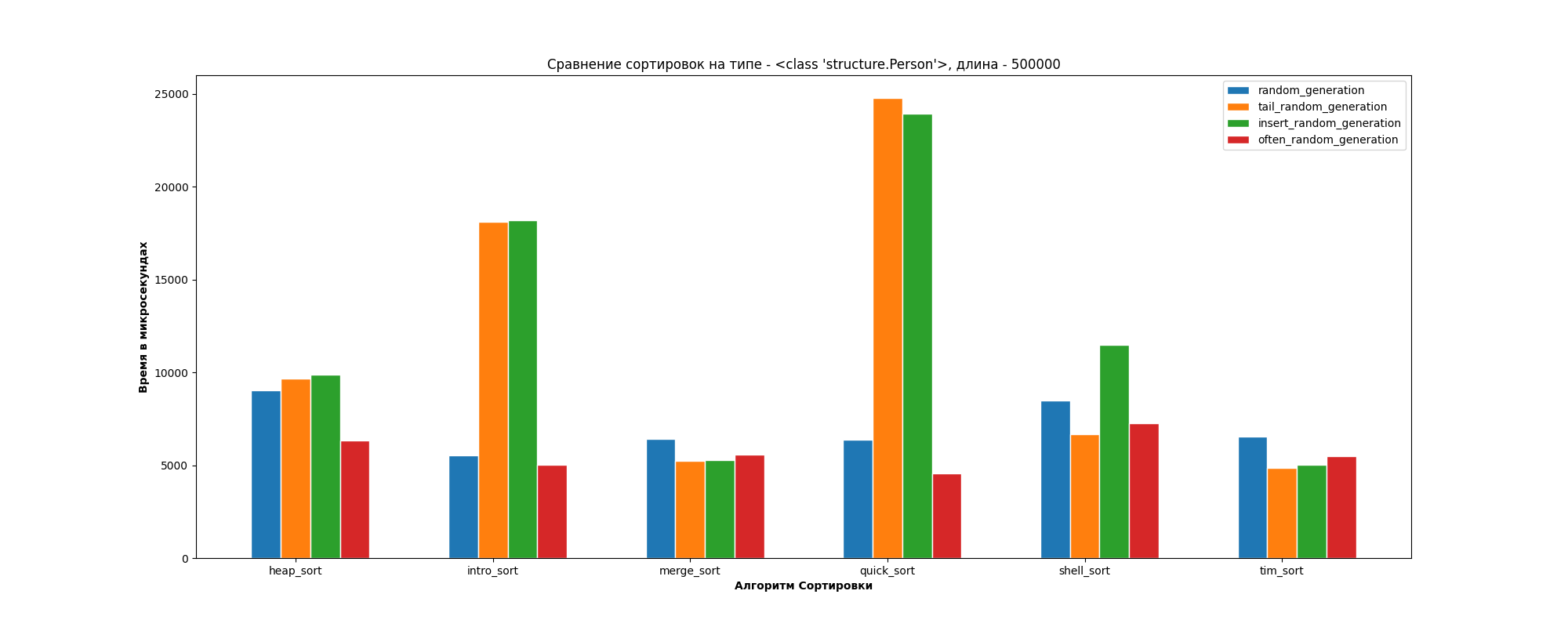
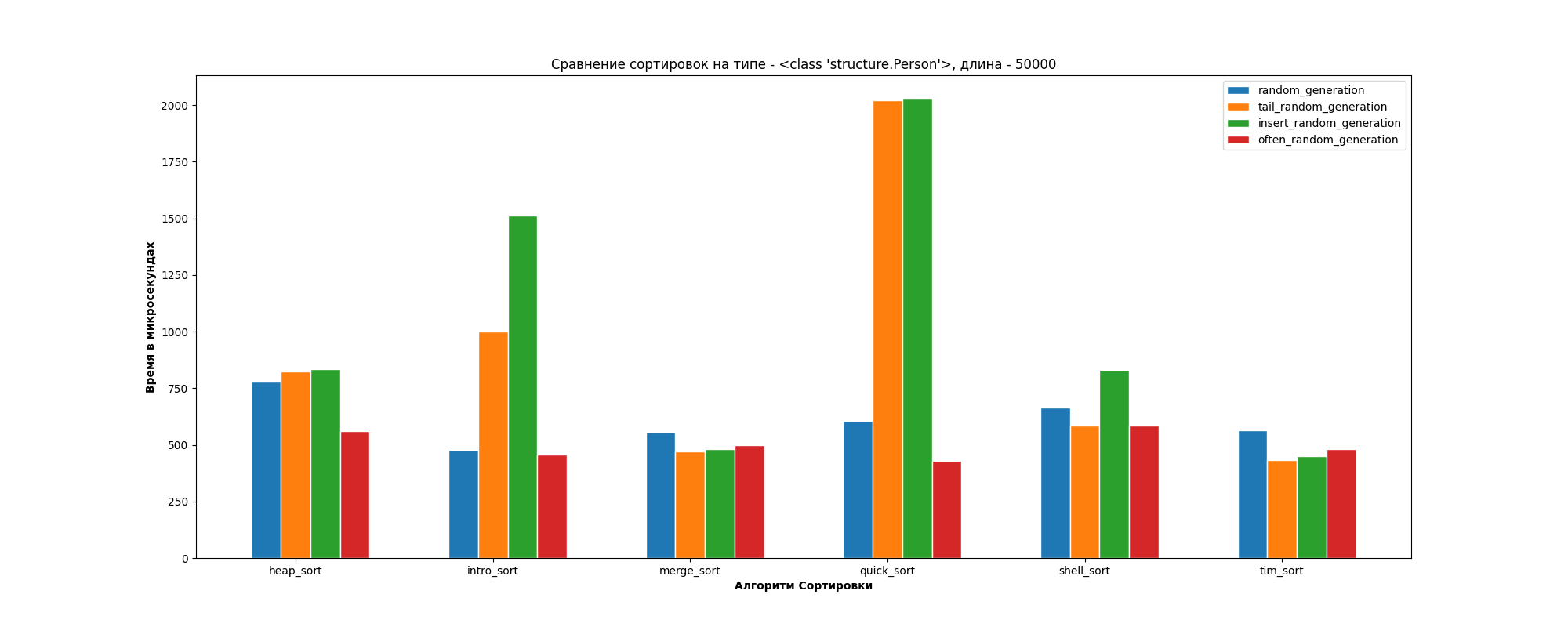
Многие алгоритмы работают хорошо только с массивами где много похожих элементов, тут лидирует quick\_sort и intro\_sort

**Тип данных struct:**

Рассмотрим 500 и 50 вместе. Видно что на обоих длинах массивов все сортировки кроме блинной показывают хороший результат.

****

На структурах никакая сортировка из эффективных не выбивается вперед. А в простых также блинная показывает худший вариант.



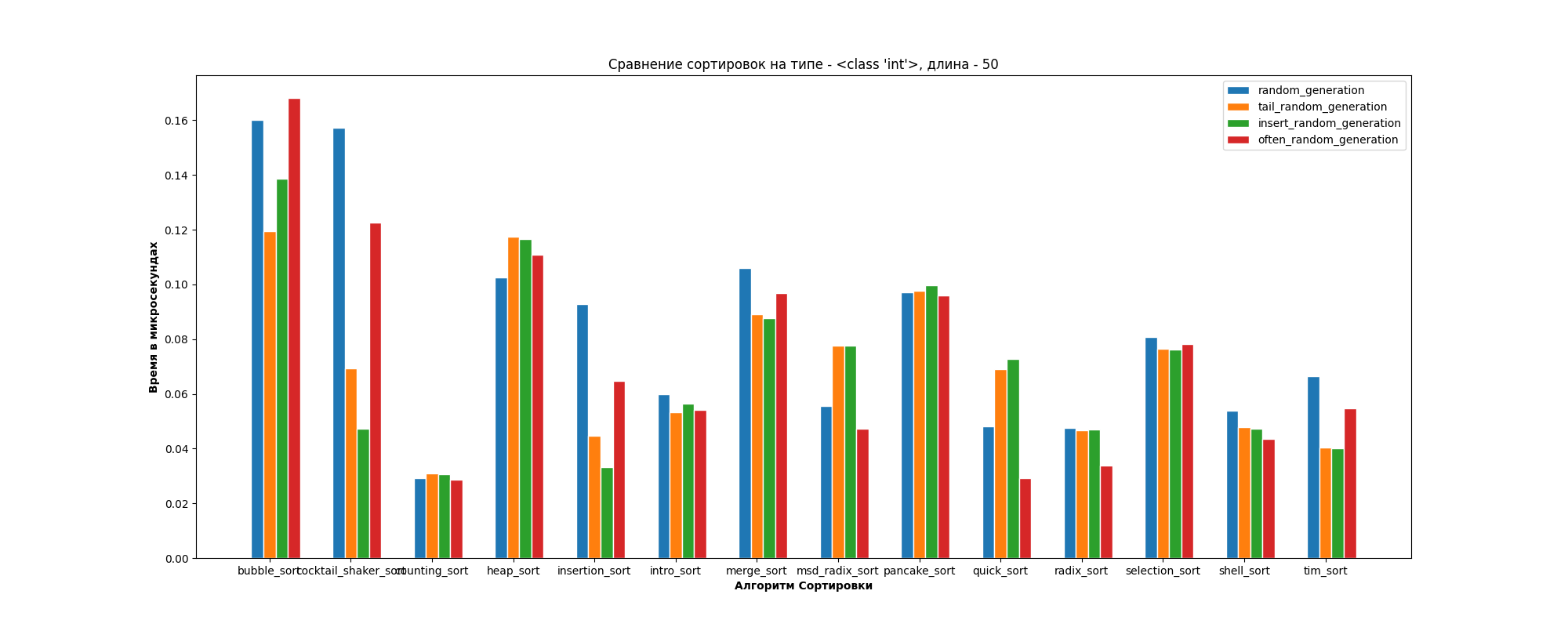
На 50к и 500к элементов ситуация такая же. Единственное что можно отметить это снова плохой результат интроспективной и быстрой сортировки на tail\_random\_arr и insert\_random\_arr. Лидером же на любом типе генерации является merge\_sort потому что в нем нет разницы какой массив подается и всегда показывает одинаковое время

Итог по Struct:

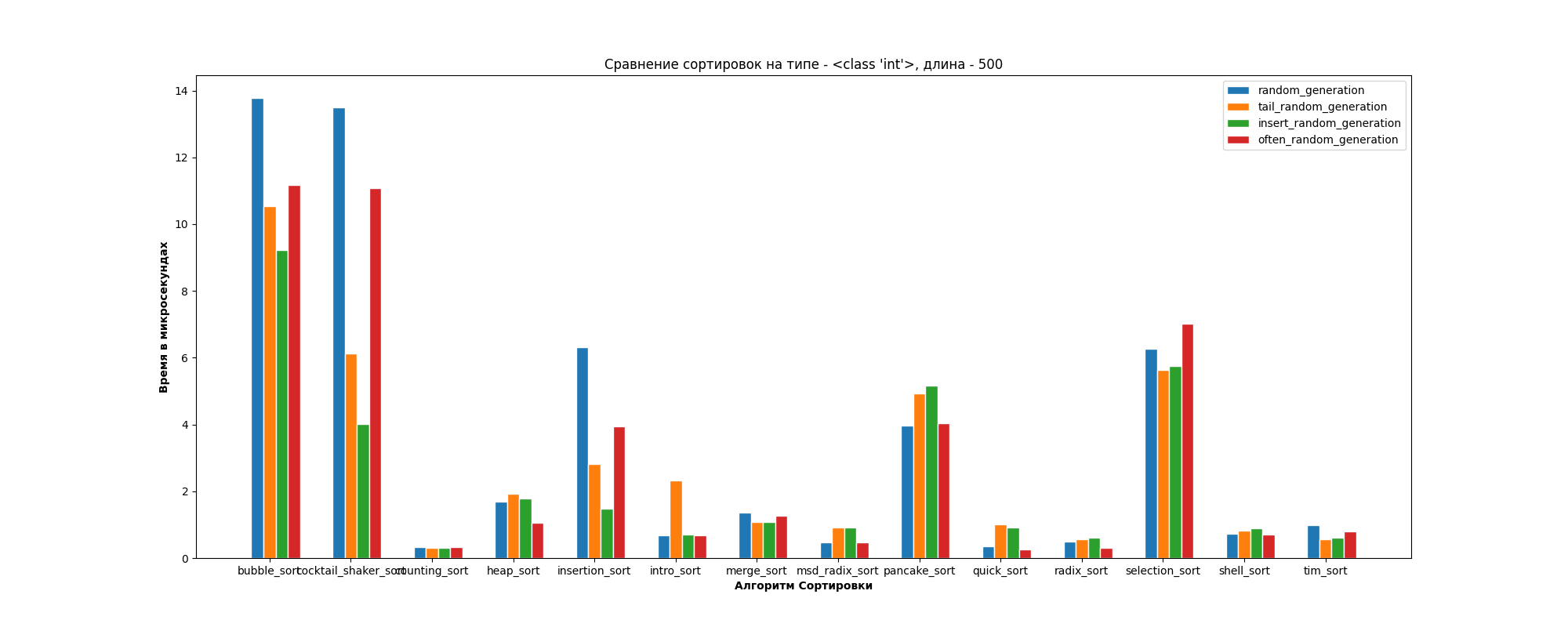
Итог по простым сортировкам такой, что блинная хуже всего справляется со всеми типами массивов и ее не стоит использовать со структурами. Среди эффективных лидером является merge\_sort которая одинаково обрабатывает все типа данных из-за своего алгоритма. А quick\_sort и intro\_sort показывают ужасные результаты на почти отсортированных массивах

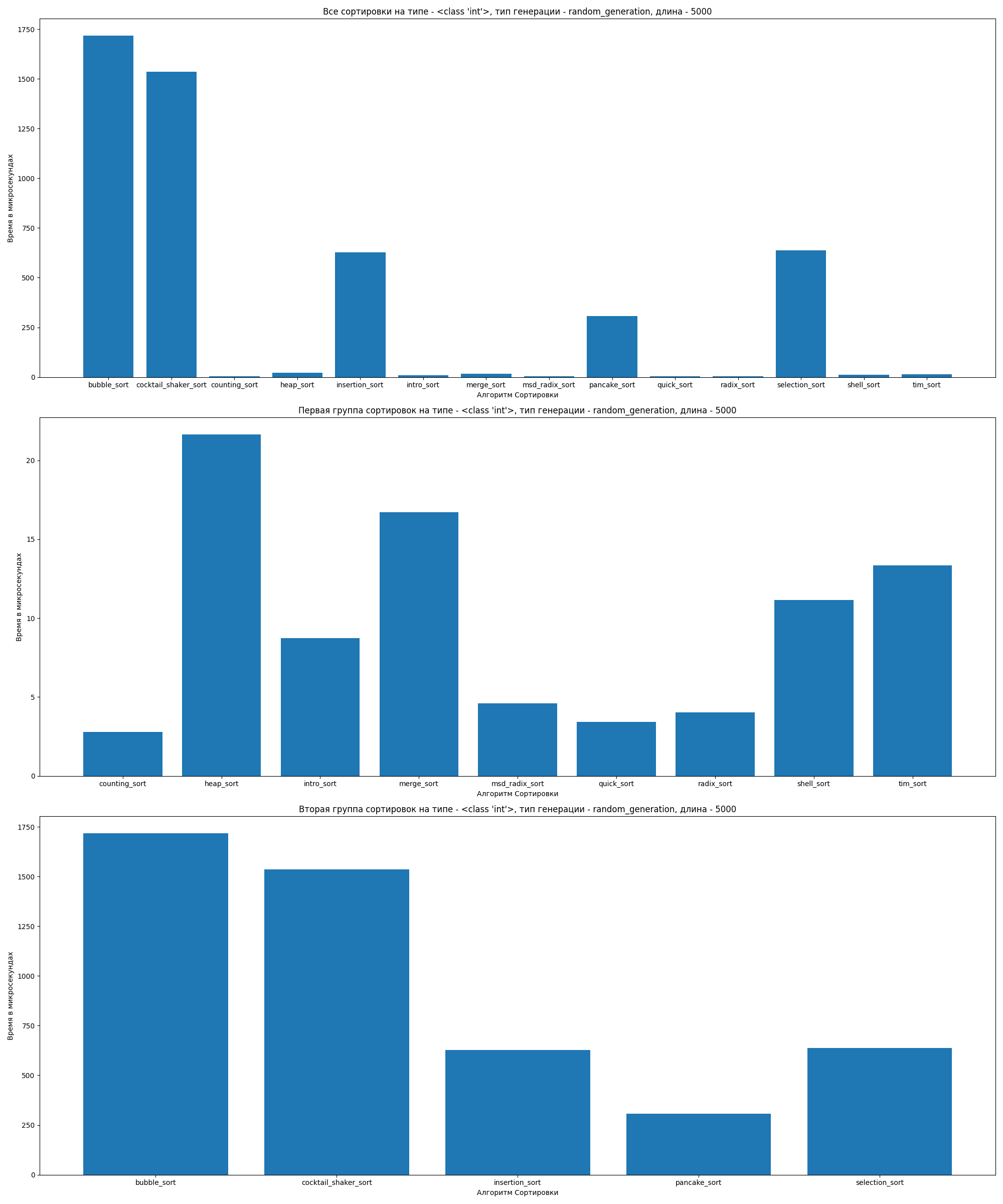
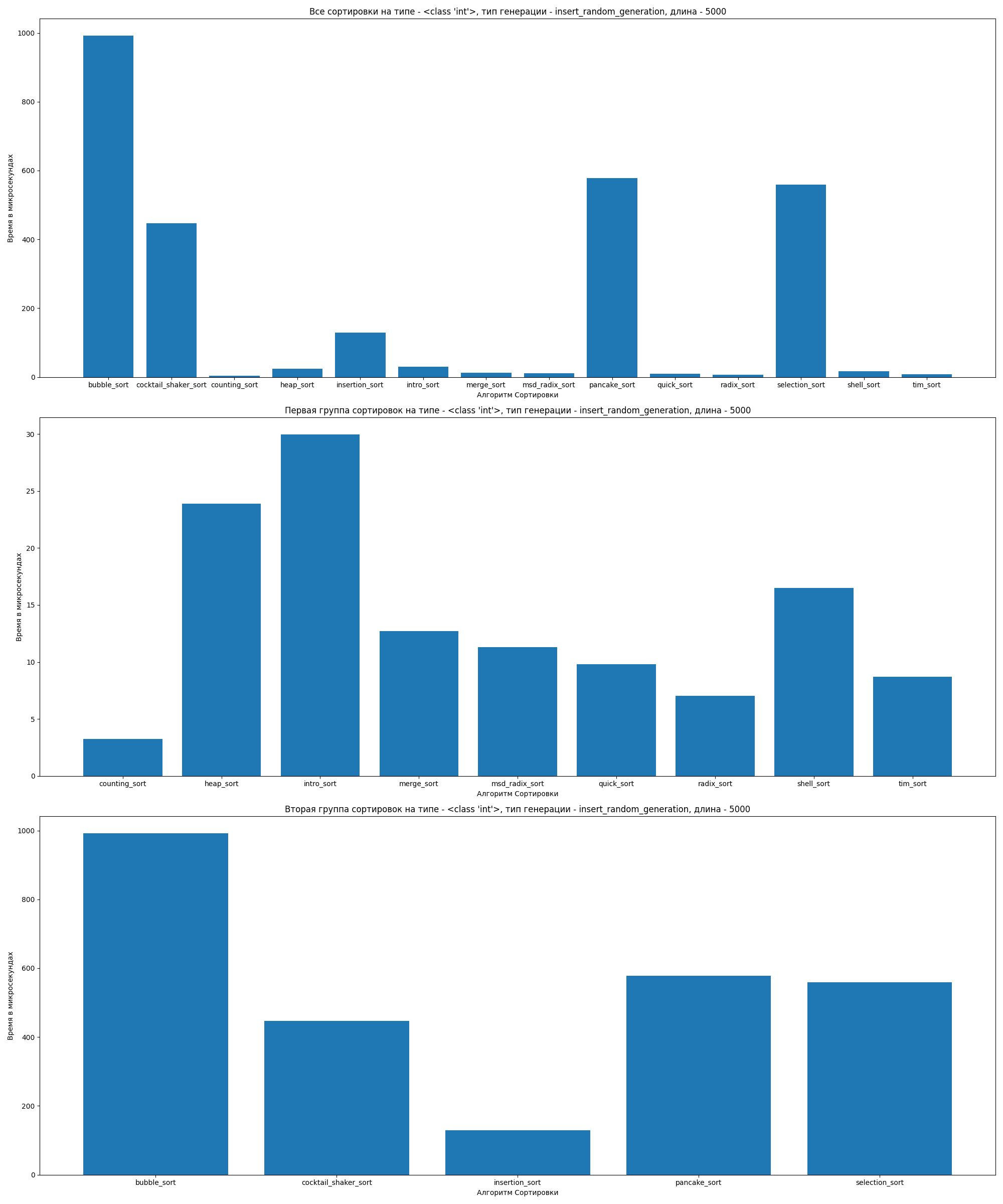
**Тип данных int:**

Для типа данных int я сравнивал кроме основных сортировок Msd, Lsd и Counting\_sort потому что они подходят под заранее известные данные.

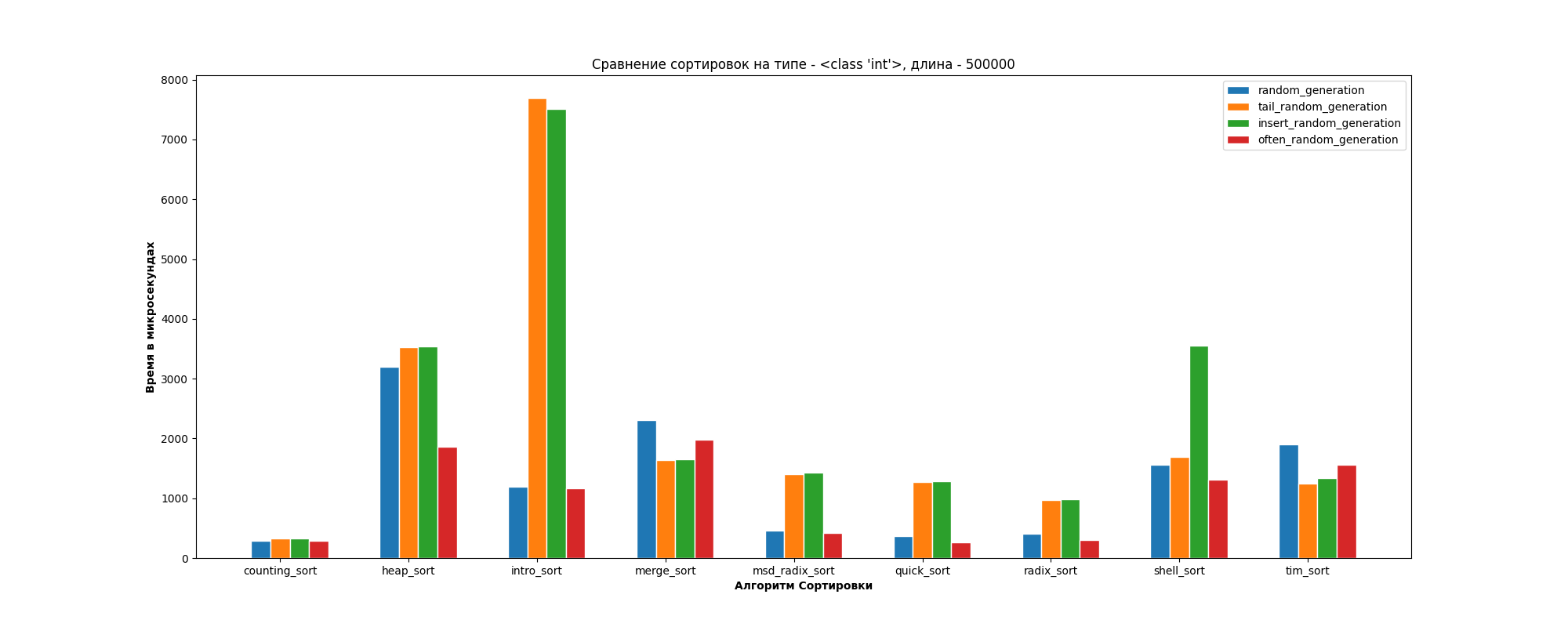
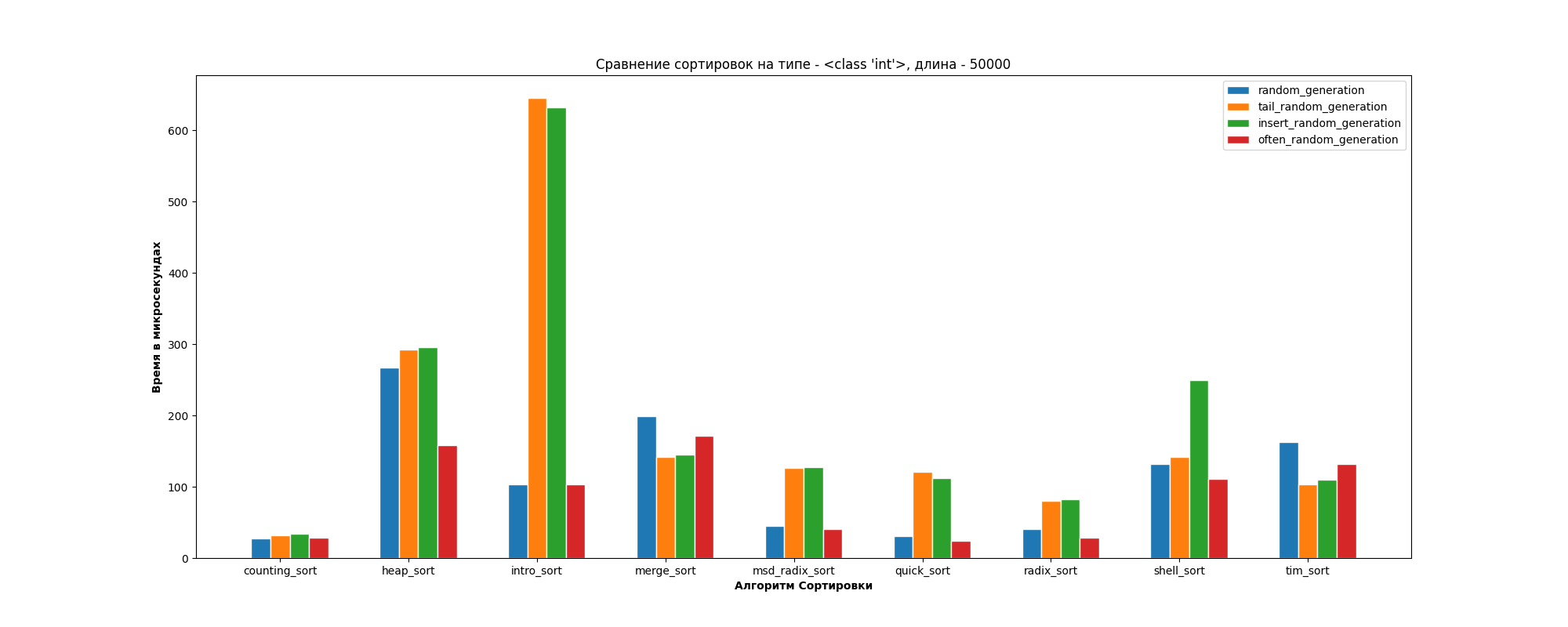


На числовых типах данных как не удивительно в лидеры сразу выбиваются counting\_sort, radix\_sort и тим сорт из-за огранниченного диапазонаа чисел

Простые сортировки в виде bubble\_sort и cocktail\_sort показывают худшее время потому что для сортировки необходим полный перебор в цикле всех значений. Другие же простые сортировки показывают более хороший результат



На 5к данных и более интерес представляют только эффективные сортировки. Видно что radix\_sort, msd и lsd независимо от типа генерации показывают самый идеальный вариант, потому что специально создавались под заранее известный тип данных. Quick\_sort и intro\_sort как всегда показывают неплохой результат на полностью рандомных массивах, а на остальных у них проблеммы.

50к и 500к в данном случае можно рассмотреть вместе.

Counting\_sort чем больше данные тем сильнее выбивается в лидеры, остальные показываютт такой же результат.

Итог по int:

Самой лучшей сортировкой для массивов с числами является counting\_sort, lsd и msd. Также неплохой результат показывает quick\_sort если брать его на полностью отсортированных массивах. Отсальные показывают более медленный результат.

Заключение

После того как я провел анализ, можно придти к выводу, что простые сортировки показывают ужасные результаты на больших массивах любых типов данных. Также на практике сложность этих алгоритмов близка к теоретической оценке O(n\*\*2). Также можно отметить что на типе данных Struct блинная показывает худший результат. На типе данных Bytes insertion\_sort является лучшей.

Если сравнивать практическую сложность эффективных алгоритмов, то она тоже приближается к теоретической O(log(n)\*n). Можно сделать вывод, что при использовании массивов со строками наилучшим вариантом будет quick\_sort или intro\_sort, особенно на массивах с повторяющимеся элементами. Однако у этих двух сортировок большие проблемы если надо отсортировать немного испорченные массивы.  
heap\_sort почти во всех случаях показывает самый худший вариант поэтому нет смысла его использовать. Встроенная tim\_sort стабильно хороша на всех типах данных, но в некоторых случаях проигрывает quick\_sort. Ее можно использовать в большинстве случаев. На типе данных int самый лучших результат показывают конечно Lsd, Msd и counting\_sort из-за заранее известного типа массива. Поэтому если точо известно какой будет массив или какой диапазон значений, то обязательно используем counting\_sort

Приложение 1. Программный код

Код сортировок:

1) Bubbble sort

def bubble\_sort(arr: list) -> list:

for i in range(len(arr)):

for j in range(len(arr)-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

2) Cocktail sort

def cocktail\_sort(arr: list) -> list:

start, end = 0, len(arr) - 1

while start < end:

for i in range(start, end):

if arr[i] > arr[i + 1]:

arr[i], arr[i + 1] = arr[i + 1], arr[i]

end -= 1

for i in range(end, start, -1):

if arr[i] < arr[i - 1]:

arr[i], arr[i - 1] = arr[i - 1], arr[i]

start += 1

return arr

3) Counting sort

def counting\_sort(arr: list) -> list:

counts\_dict = {i: arr.count(i) for i in arr}

result = []

for i in sorted(counts\_dict):

result.extend([i]\*counts\_dict[i])

return result

**4) Heap sort**

def heap(arr: list, i: int, size: int) -> None:

max\_value = i

l = 2 \* i + 1

r = l + 1

max\_value = l if l < size and arr[l] > arr[max\_value] else max\_value

max\_value = r if r < size and arr[r] > arr[max\_value] else max\_value

if max\_value != i:

arr[max\_value], arr[i] = arr[i], arr[max\_value]

heap(arr, max\_value, size)

def heap\_sort(arr: list) -> list:

for i in range(len(arr) // 2 - 1, -1, -1):

heap(arr, i, len(arr))

for i in range(len(arr) - 1, 0, -1):

arr[0], arr[i] = arr[i], arr[0]

heap(arr, 0, i)

return arr

5) Insertion sort

def insertion\_sort(arr: list) -> list:

for i in range(1, len(arr)):

temp = arr[i]

j = i - 1

while j >= 0 and temp < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = temp

return arr

6) Introspective Sort

from . import heap\_sort

import math

def insertion\_sort(array, start=0, end=0):

for i in range(start, end):

temp\_index = i

while temp\_index != start and array[i] < array[temp\_index - 1]:

array[temp\_index] = array[temp\_index - 1]

temp\_index -= 1

array[temp\_index] = array[i]

return array

def partition(array, low, high, pivot):

i = low

j = high

while True:

while array[i] < pivot: i += 1

j -= 1

while pivot < array[j]: j -= 1

if i >= j: return i

array[i], array[j] = array[j], array[i]

i += 1

def intro\_sort(arr):

max\_depth = 2 \* math.log2(len(arr))

return intro(arr, 0, len(arr) - 1, 5, int(max\_depth))

def intro(array, start, end, size\_threshold, max\_depth):

while end - start > size\_threshold:

if max\_depth == 0: return heap\_sort.heap\_sort(array)

max\_depth -= 1

pivot = sorted(array[start],array[start + (end - start)//2 + 1],array[end - 1])[0]

p = partition(array, start, end, pivot)

intro(array, p, end, size\_threshold, max\_depth)

end = p

return insertion\_sort(array, start, end)

7) LSD Radix Sort

from typing import List

def lsd\_radix\_sort(arr: List[int]) -> List[int]:

placement = 1

while placement <= max(arr):

buckets = [[] for \_ in range(13)]

for i in arr:

tmp = int((i / placement) % 13)

buckets[tmp].append(i)

a = 0

for b in range(13):

for i in buckets[b]:

arr[a] = i

a += 1

placement \*= 13

return arr

8) MSD Radix Sort

from typing import List

def msd\_radix\_sort(arr: List[int]) -> List[int]:

exp = 1

while max(arr)//exp > 0:

exp \*= 10

exp //= 10

return \_msd\_radix\_sort(arr, exp)

def \_msd\_radix\_sort(arr: List[int], exp: int) -> List[int]:

if exp < 1:

return arr

output = [0] \* len(arr)

count = [0] \* 10

for num in arr:

index = num // exp

count[index % 10] += 1

for i in range(1, 10):

count[i] += count[i - 1]

i = len(arr) - 1

while i >= 0:

num = arr[i]

index = num // exp

output[count[index % 10] - 1] = num

count[index % 10] -= 1

i -= 1

i = 0

for i in range(len(arr)):

arr[i] = output[i]

next\_exp = exp // 10

for i in range(10):

start = count[i]

end = count[i + 1] if i + 1 < 10 else len(arr)

\_msd\_radix\_sort(arr[start:end], next\_exp)

return arr

9) Merge Sort

def merge\_sort(arr: list) -> list:

if len(arr) <= 1:

return arr

center = len(arr) // 2

return merge\_arrays(merge\_sort(arr[:center]), merge\_sort(arr[center:]))

def merge\_arrays(left: list, right: list) -> list:

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 += left[i:]

result += right[j:]

return result

10) Pancake Sort

def pancake\_sort(arr: list) -> list:

for cur in range(len(arr), 1, -1):

temp = arr.index(max(arr[:cur]))

arr = arr[temp::-1] + arr[temp + 1:] if temp else arr[::-1]

return arr

11) Quick Sort

def quick\_sort(arr: list) -> list:

if len(arr) <= 1:

return arr

pivot = sorted(arr[0], arr[len(arr)//2], arr[-1])[1]

less = [x for x in arr if x < pivot]

equal = [x for x in arr if x == pivot]

greater = [x for x in arr if x > pivot]

return quick\_sort(less) + equal + quick\_sort(greater)

12) Selection Sort

def selection\_sort(arr: list) -> list:

for i in range(len(arr)):

index = i

for j in range(i+1, len(arr)):

index = j if arr[j] < arr[index] else index

arr[i], arr[index] = arr[index], arr[i]

return arr

13) Shell Sort

def shell\_sort(arr: list) -> list:

gap = n // 2

while gap > 0:

for i in range(gap, len(arr)):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

gap //= 2

return arr

14) Tim Sort

def merge\_sort(arr, start, mid, end):

if mid == end:

return arr

first, last = arr[start:mid+1], arr[mid+1:end+1]

ind1 = ind2 = 0

for ind in range(start, end+1):

if ind1 < len(first) and (ind2 == len(last) or first[ind1] < last[ind2]):

arr[ind] = first[ind1]

ind1 += 1

else:

arr[ind] = last[ind2]

ind2 += 1

return arr

def tim\_sort(arr: list) -> list:

n = len(arr)

for start in range(0, n, 30):

end = min(start+30-1, n-1)

arr = insertion\_sort(arr, start, end)

curr\_size = 30

while curr\_size < n:

for start in range(0, n, curr\_size\*2):

mid = min(n-1, start+curr\_size-1)

end = min(n-1, mid+curr\_size)

arr = merge\_sort(arr, start, mid, end)

curr\_size \*= 2

return arr

Decorators:

from typing import Callable

import time

import copy

def iterations(iter\_count: int):

def wrapper(func: Callable):

def inner\_wrapper(arr: list):

total\_time = 0

for \_ in range(iter\_count):

start\_time = time.perf\_counter\_ns()

new\_arr = copy.deepcopy(arr)

func(new\_arr)

end\_time = time.perf\_counter\_ns()

total\_time += end\_time - start\_time

return total\_time/iter\_count

return inner\_wrapper

return wrapper

iterations\_10 = iterations(iter\_count=10)

iterations\_50 = iterations(iter\_count=50)

Generators:

import random

from typing import Callable, List

from structure import Student

import string

def generate\_integers\_arr(count: int, f: int, l: int) -> List[int]:

return [random.randint(f, l) for \_ in range(count)]

def generate\_bytes\_arr(count: int, f: int = 0, l: int = 9) -> List[bytes]:

return [(random.randint(f, l)).to\_bytes() for \_ in range(count)]

def generate\_strings\_arr(count: int, f: int = 5, l: int = 10, weight: int = 1) -> List[str]:

return [''.join(random.choices(string.ascii\_letters, weights=[weight, weight]+[1 for \_ in range(50)], k=random.randint(f, l))) for \_ in range(count)]

def generate\_structs\_arr(count: int, f: int, l: int) -> List[Student]:

return [Student(average\_score=random.randint(f, l), name=''.join(random.choices(string.ascii\_letters)))]

def generate\_repeated\_value(gen\_type: type, count: int, f: int) -> list:

if gen\_type == str:

return generate\_strings\_arr(count, 2, 2, 1000)

elif gen\_type == int:

return generate\_integers\_arr(count, f, f+1)

elif gen\_type == bytes:

return generate\_bytes\_arr(count, f, f+1)

elif gen\_type == Student:

return generate\_structs\_arr(count, f, f+1)

def generate\_partially\_random(type: type, count: int, f: int, l: int, random\_elements\_count: int, gen\_type: str) -> list:

generate\_func: Callable[[int, int, int], List[type]]

arr: List[type] = []

if type == str:

alph = string.ascii\_letters

max\_count = (count-random\_elements\_count) // len(alph)

break\_flag = False

for el in alph:

if break\_flag:

break

for i in range(f, f+max\_count+1):

if len(arr) == (count-random\_elements\_count):

break\_flag = True

break

arr.append(el\*i)

generate\_func = generate\_strings\_arr

elif type == int:

generate\_func = generate\_integers\_arr

arr = [i for i in range(1, count-random\_elements\_count)]

elif type == bytes:

l = 255 if l > 256 else l+1

max\_count = (count-random\_elements\_count) // l

break\_flag = False

for i in range(l):

if break\_flag:

break

for \_ in range(max\_count):

if len(arr) == (count-random\_elements\_count):

break\_flag = True

break

arr.append(i.to\_bytes())

generate\_func = generate\_bytes\_arr

elif type == Student:

generate\_func = generate\_structs\_arr

arr = [Student(average\_score=i, name=''.join(random.choices(string.ascii\_letters)))

for i in range(1, count-random\_elements\_count)]

random\_arr: List[type] = generate\_func(random\_elements\_count, f, l)

if gen\_type == "tail":

arr += random\_arr

elif gen\_type == "insert":

for el in random\_arr:

arr.insert(random.randint(0, len(arr)-1), el)

return arr

Generate\_data:

from typing import List, Tuple

from generators import \*

def generate\_data(cf: int, f: int, l: int):

gens = [

{"function":generate\_bytes\_arr,"type":bytes,},

{"function":generate\_strings\_arr,"type":str,},

{"function":generate\_integers\_arr,"type":int,},

{"function":generate\_structs\_arr,"type":Student,}]

results = {}

for val in gens:

gen\_func = val["function"]

gen\_type = val["type"]

results[gen\_type] = {'random\_generation': {}, 'tail\_random\_generation': {}, 'insert\_random\_generation': {}, 'often\_random\_generation': {}}

for i in range(1,cf):

length = (10\*\*i)\*5

coef = int(length/10)

results[gen\_type]['random\_generation'][length] = gen\_func(length, f, l)

results[gen\_type]['tail\_random\_generation'][length] = generate\_partially\_random(gen\_type, length, f, l, coef, "tail")

results[gen\_type]['insert\_random\_generation'][length] = generate\_partially\_random(gen\_type, length, f, l, coef, "insert")

results[gen\_type]['often\_random\_generation'][length] = generate\_repeated\_value(gen\_type, length, f)

return results

Sorts:

from sorts import \*

from typing import Any

sorts = [

{

"func": bubble\_sort,

"max\_count": 5000,

"allowed\_types": Any

},

{

"func": cocktail\_sort,

"max\_count": 5000,

"allowed\_types": Any

},

{

"func": counting\_sort,

"max\_count": 500000,

"allowed\_types": int

},

{

"func": heap\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

{

"func": insertion\_sort,

"max\_count": 5000,

"allowed\_types": Any

},

{

"func": intro\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

{

"func": merge\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

{

"func": msd\_radix\_sort,

"max\_count": 500000,

"allowed\_types": int

},

{

"func": pancake\_sort,

"max\_count": 5000,

"allowed\_types": Any

},

{

"func": quick\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

{

"func": lsd\_radix\_sort,

"max\_count": 500000,

"allowed\_types": int

},

{

"func": selection\_sort,

"max\_count": 5000,

"allowed\_types": Any

},

{

"func": shell\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

{

"func": tim\_sort,

"max\_count": 500000,

"allowed\_types": Any

},

]

Structure:

from dataclasses import dataclass

@dataclass

class Student:

name: str

average\_score: float

def \_\_lt\_\_(self, other: Student) -> bool:

return self.average\_score < other.average\_score

def \_\_le\_\_(self, other: Student) -> bool:

return self.average\_score <= other.average\_score

Run sorting algorithms:

from typing import Any

from get\_data\_map import get\_data\_map

from decorators import iterations\_10, iterations\_50

from sorts\_dict import sorts

import json

import gc

gc.disable()

data\_map = generate\_data(6, 3, 10)

results = {}

for type in data\_map.keys():

results[str(type)] = {}

for data\_type in data\_map[type].keys():

results[str(type)][data\_type] = {}

for count in data\_map[type][data\_type].keys():

results[str(type)][data\_type][count] = {}

for sort in sorts:

if sort['allowed\_types'] != Any and sort['allowed\_types'] == type or sort['allowed\_types'] == Any:

if sort['max\_count'] >= count:

if count>=50000:

results[str(type)][data\_type][count][sort["func"].\_\_name\_\_] = iter\_10(sort["func"])(data\_map[type][data\_type][count])

else:

results[str(type)][data\_type][count][sort["func"].\_\_name\_\_] = iter\_50(sort["func"])(data\_map[type][data\_type][count])

with open("results.json", "w") as file:

json.dump(res\_map, file)

Generate visualization figures:

coef = 15

for type in data:

for genType in data[type]:

for length in data[type][genType]:

values = [float(x) / 1000000 for x in data[type][genType][length].values()]

keys = list(data[type][genType][length].keys())

min\_value = min(values)

first\_values = [v for v in values if v <= coef \* min\_value]

first\_keys = [k for k, v in zip(keys, values) if v <= coef \* min\_value]

second\_values = [v for v in values if v > coef \* min\_value]

second\_keys = [k for k, v in zip(keys, values) if v > coef \* min\_value]

fig, axs = plt.subplots(3, figsize=(20, 24))

axs[0].bar(keys, values)

axs[0].set\_xlabel('Алгоритм Сортировки')

axs[0].set\_ylabel('Время в микросекундах')

axs[0].set\_title(f'Все сортировки на типе - {type}, тип генерации - {genType}, длина - {length}')

if second\_values:

axs[1].bar(first\_keys, first\_values)

axs[1].set\_xlabel('Алгоритм Сортировки')

axs[1].set\_ylabel('Время в микросекундах')

axs[1].set\_title(f'Первая группа сортировок на типе - {type}, тип генерации - {genType}, длина - {length}')

axs[2].bar(second\_keys, second\_values)

axs[2].set\_xlabel('Алгоритм Сортировки')

axs[2].set\_ylabel('Время в микросекундах')

axs[2].set\_title(f'Вторая группа сортировок на типе - {type}, тип генерации - {genType}, длина - {length}')

plt.tight\_layout()

plt.savefig(f'r100/{type}\_{genType}\_{length}\_all.png')

plt.close()