МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №1
по дисциплине «Алгоритмы и структуры данных»
ТЕМА: «АЛГОРИТМЫ СОРТИРОВКИ»

Студент гр. 2309 _____ Шуббе Л. П.

Преподаватель _____ Пестерев Д. О.

Санкт-Петербург
2023

1. Постановка задачи

1. Реализовать следующие алгоритмы сортировки:
    - Сортировка выбором
    - Сортировка вставками
    - Сортировка пузырьком
    - Сортировка слиянием
    - Быстрая сортировка
    - Сортировка Шелла (не менее трех различных последовательностей, желательно приводящих к разным асимптотикам)
    - Пирамидальная сортировка
    - Timsort
    - IntroSort
2. Экспериментально определить время работы алгоритмов при различных размерах массива для:
    - Отсортированного массива
    - Почти отсортированного массива
    - Обратно отсортированного массива
    - Массива, о структуре которого дополнительных данных не дано

3. Постараться определить асимптотику для лучшего/среднего/худшего случая путем анализа экспериментальных данных и применения к ним нелинейной регрессии.

4. Постараться получить достоверные данные о времени работы алгоритмов сортировки при малом размере массива ~ 100 элементов

2. Временная и практическая сложность

Каждая сортировка была вызвана с четырмя необходмимыми вариантами массива. Для каждого такого варианта массив генериворался 10 раз и запускался механизм подсчёта времени работы. Далее все времена усреднялись. Также производилась проверка на правильность сортировки, в случае неверной сортировки вместо времени будет -1. Данные результаты записываются вместе с текущими настройками в .json файлы. Для сравнения добавлены встроенная и неработающая (случайное перемешивание) сортировки.

- Для всех сортировок измерены времена для размеров 50 и 1000, 2000, … 10000 — result_all.json

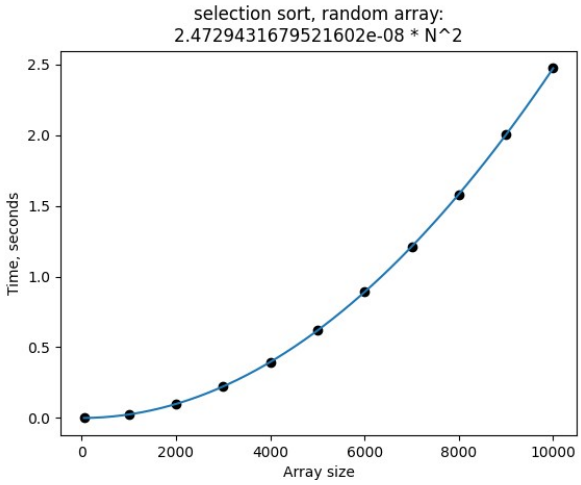
- Для быстрых сортировок (все кроме первых трёх) измерены времена для размеров 50 и 10000, 20000, … 100000 — result_fast.json

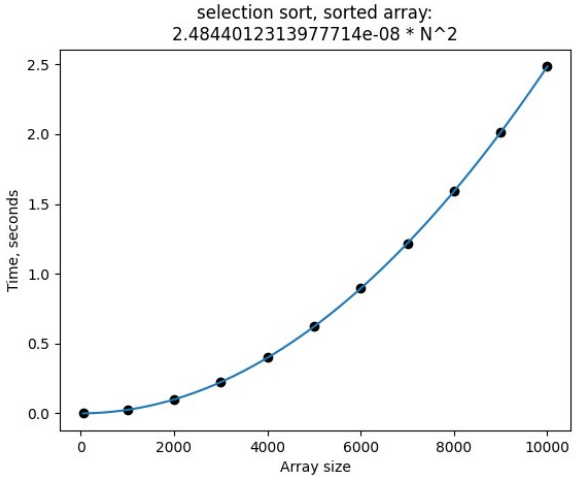
- Для всех сортировок измерены времена для размеров 50, 60, … 200 — result_low.json

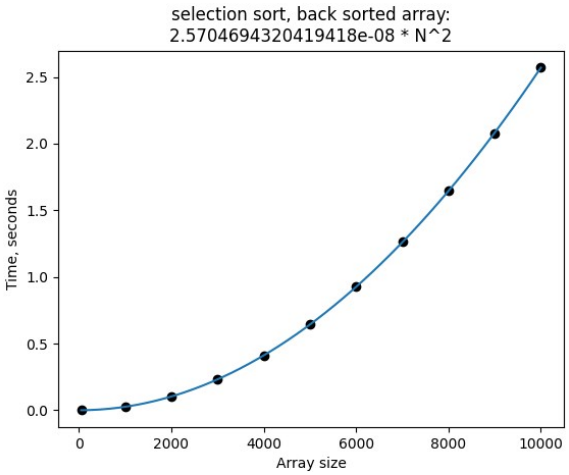
На основе этих данных скриптом render.py производится вычисление коэффициентов методом нелинейной регрессии. Для каждой из четырёх функций (ax, ax^2, ax^1.5, axlnx) подбираются коэффициенты, дающие наименьшее среднеквадратичное отклонение от экспериментальных точек. Далее из этих функций выбирается та, чьё отклонение меньше.

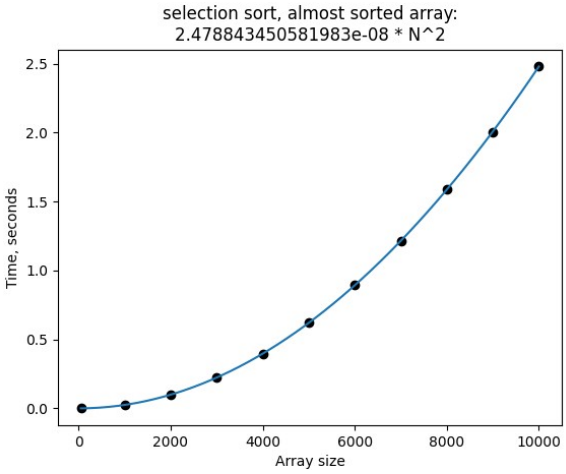
Ниже приведены названия функций, реализующих сортировку, графики и аппроксимирующие зависимости. Для первых трёх сортировок — из result_all.json, для остальных — из result_fast.json.

1. selection_sort(array)









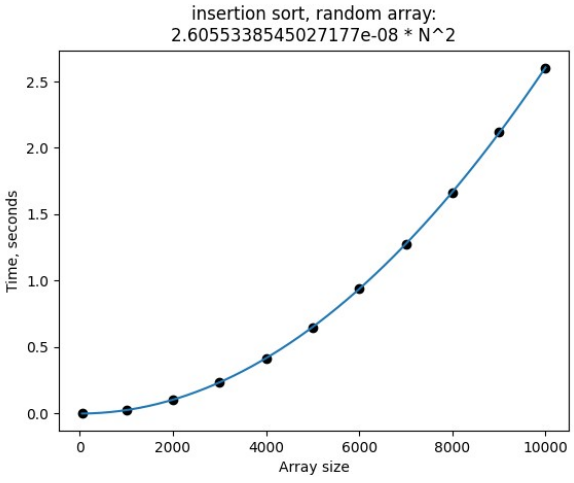
Худший случай — Θ(N^2)

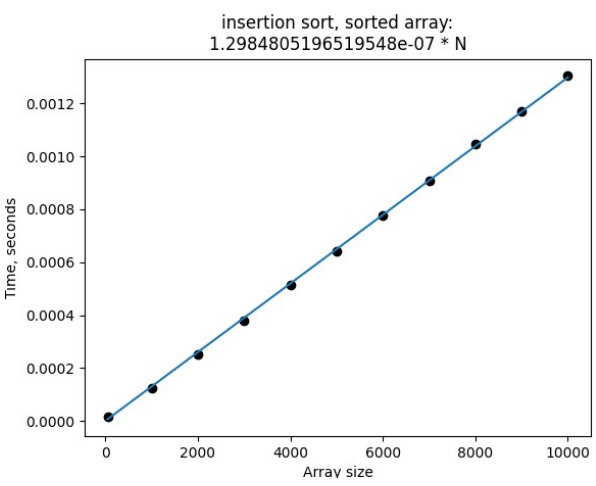
Средний случай — Θ(N^2)

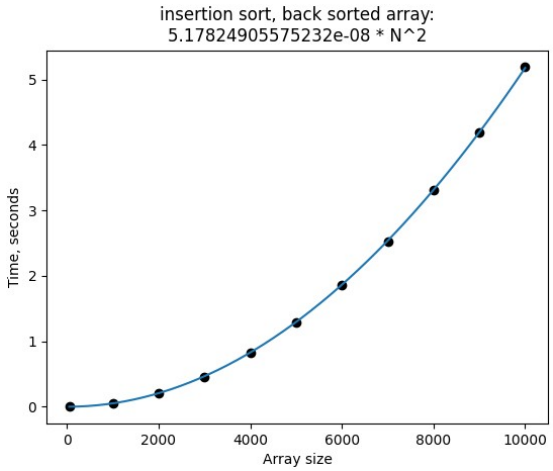
Лучший случай — Θ(N^2)

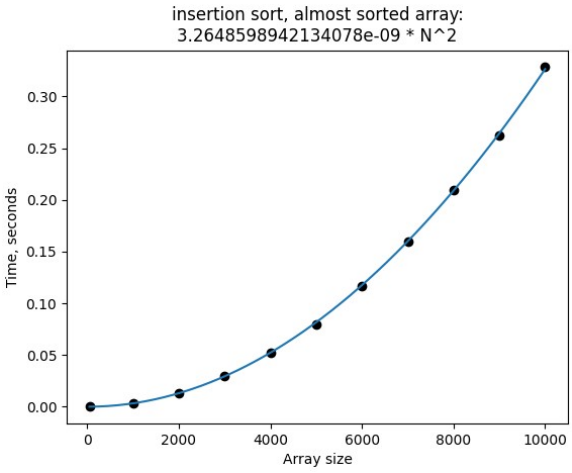
Экспериментальные данные совпадают с теоритическими. Неустойчива.

2. insertion_sort(array)









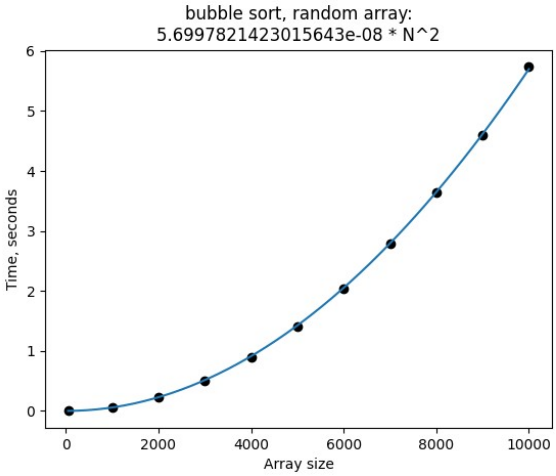
Худший случай — Θ(N^2)

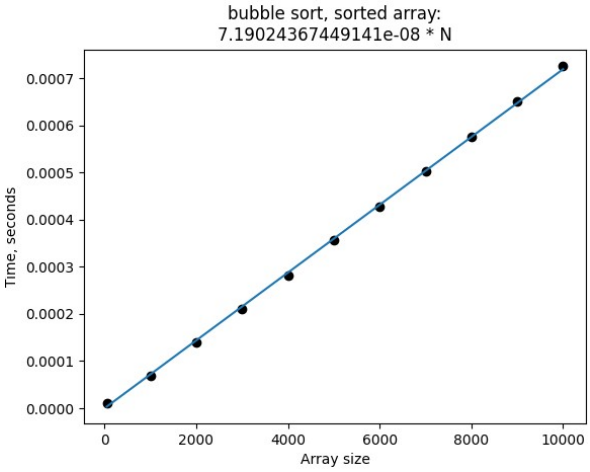
Средний случай — Θ(N^2)

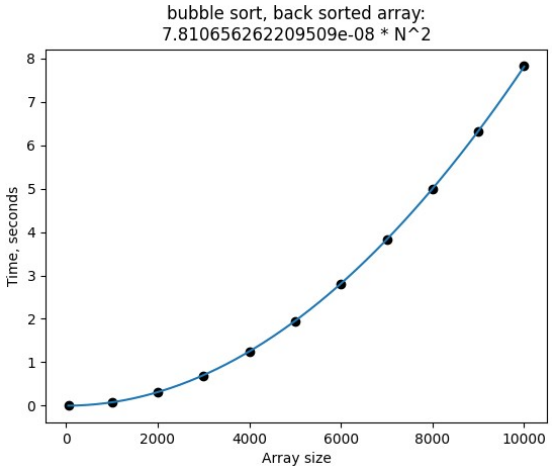
Лучший случай — Θ(N)

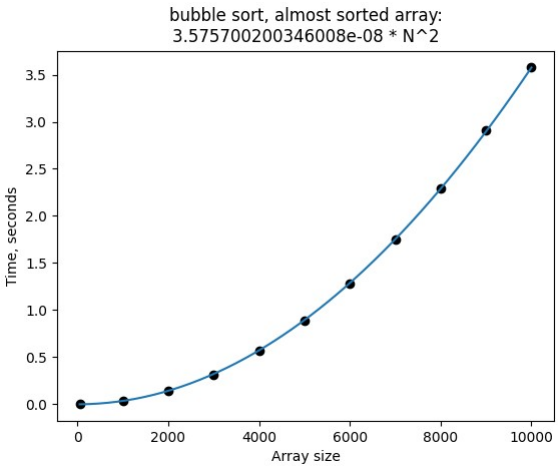
Экспериментальные данные совпадают с теоритическими. Устойчива.

3. bubble_sort(array)









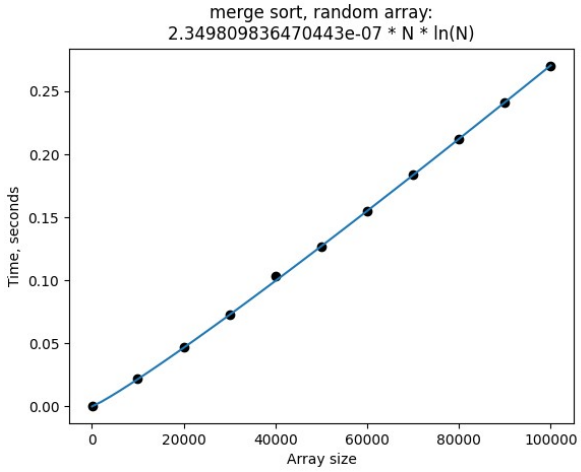
Худший случай — Θ(N^2)

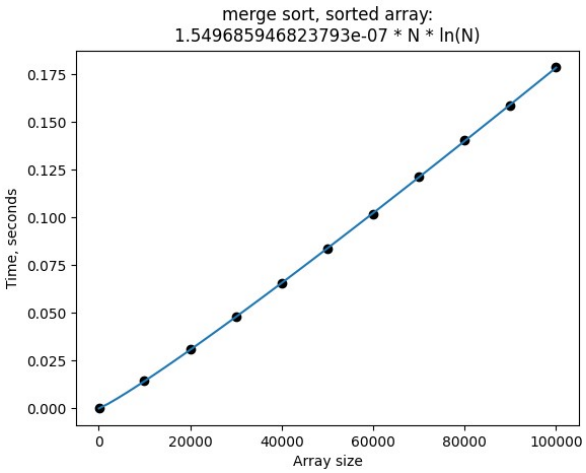
Средний случай — Θ(N^2)

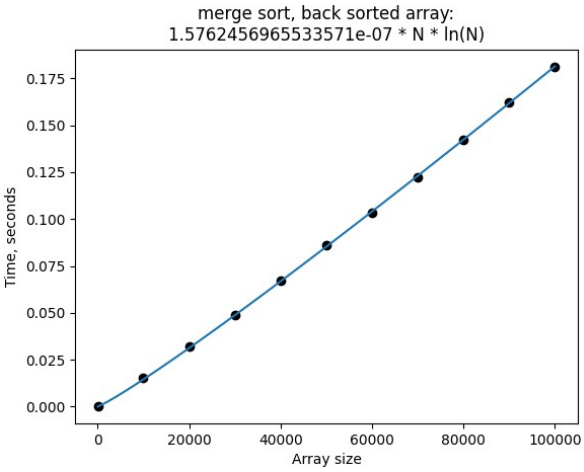
Лучший случай — Θ(N)

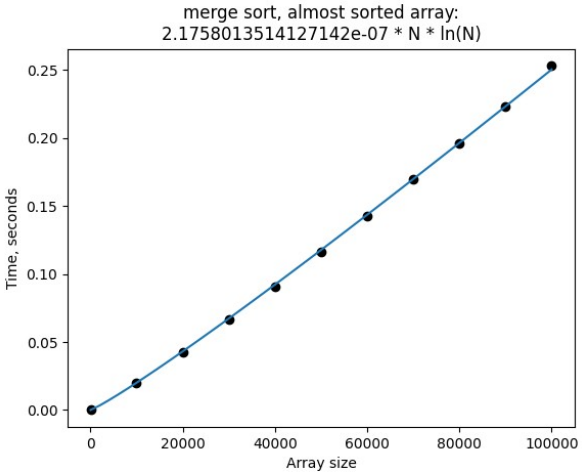
Экспериментальные данные совпадают с теоритическими. Устойчива.

4. merge_sort(array)









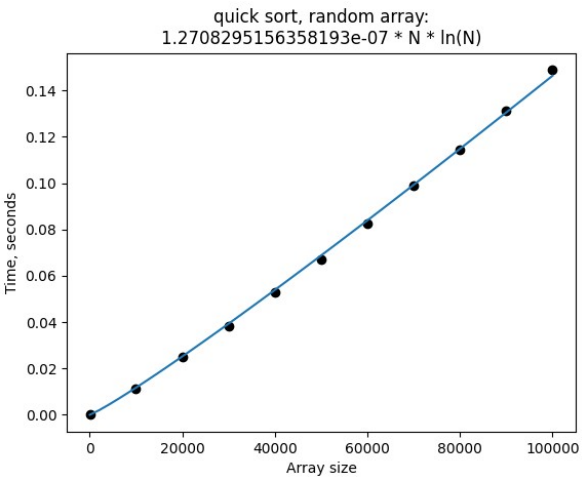
Худший случай — Θ(Nln(N))

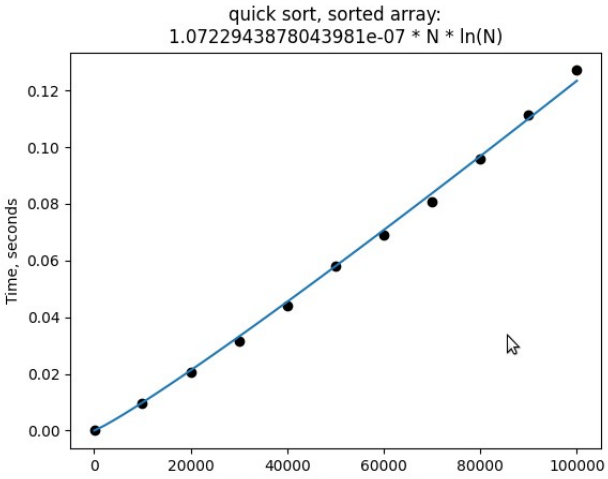
Средний случай — Θ(Nln(N))

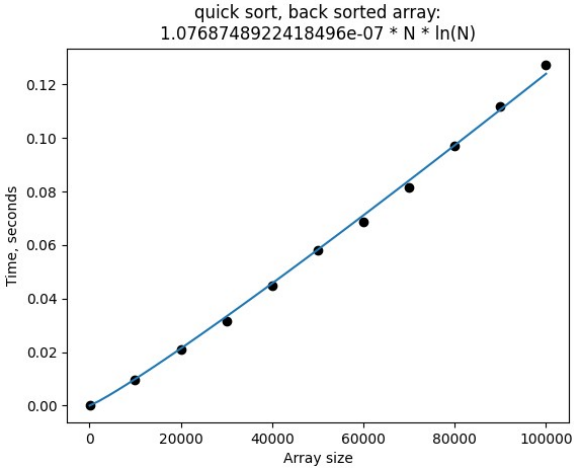
Лучший случай — Θ(Nln(N))

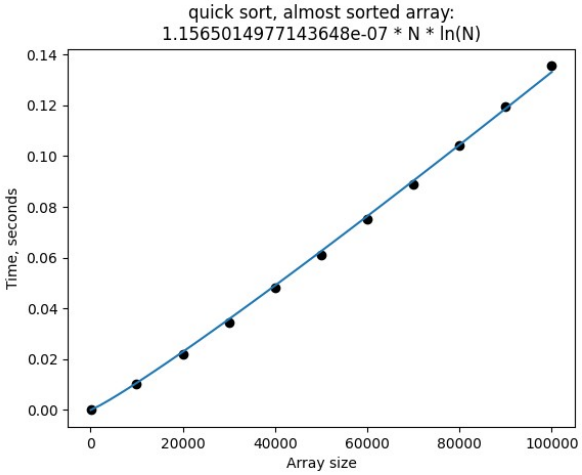
Экспериментальные данные совпадают с теоритическими. Устойчива.

5. quick_sort(array)









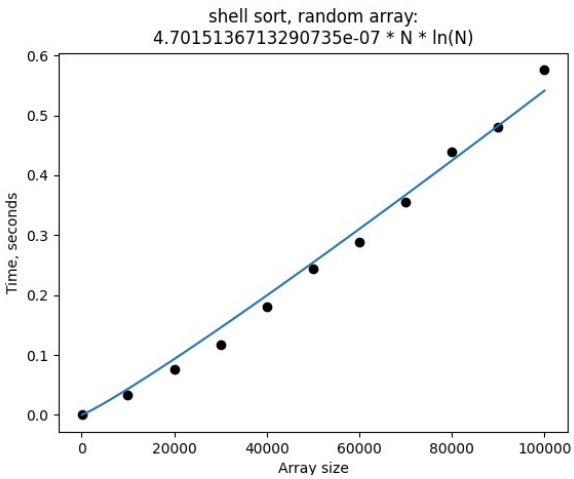
Худший случай — Θ(N^2)

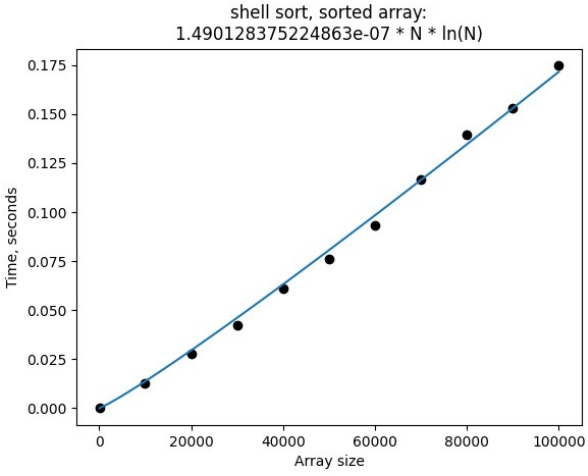
Средний случай — Θ(Nln(N))

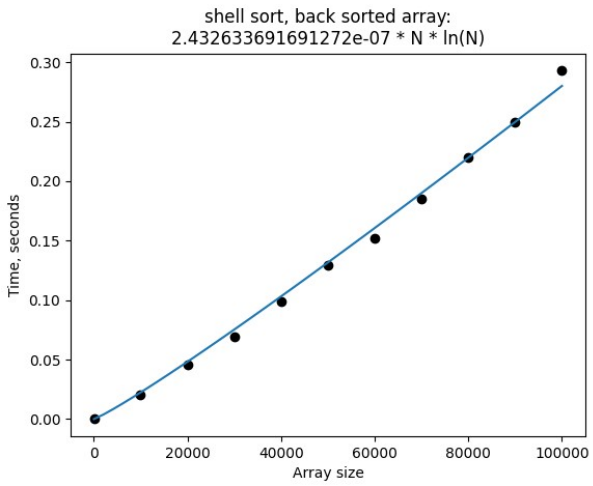
Лучший случай — Θ(Nln(N))

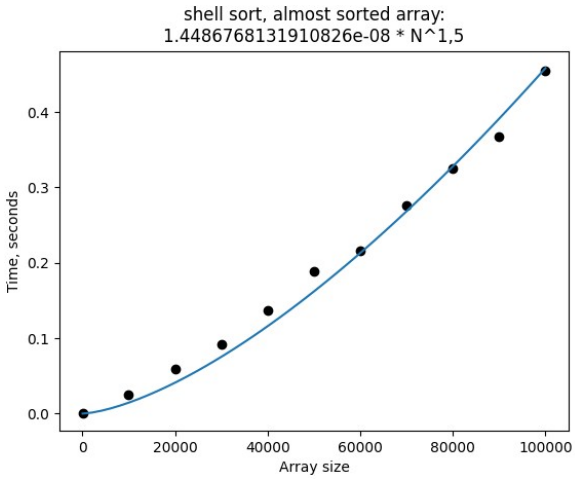
В выборке не представлен худший случай поэтому экспериментальные данные совпадают с теоритическими. Неустойчива.

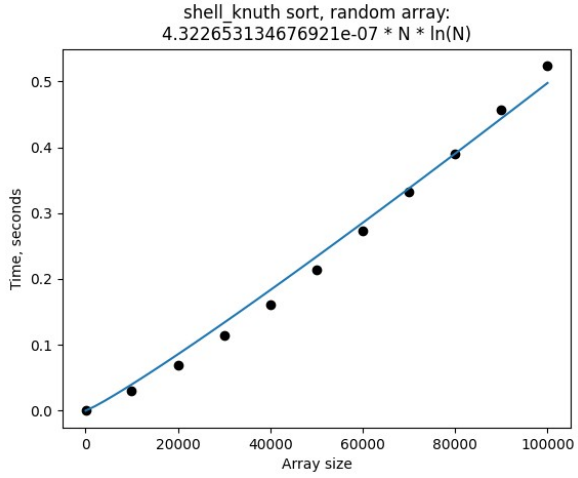
6. shell_sort(array, mode=0)

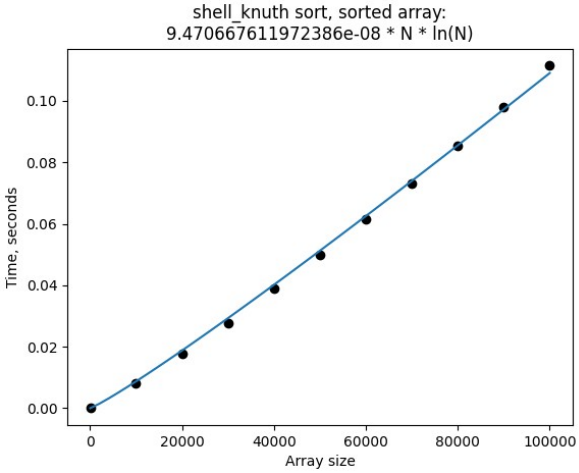


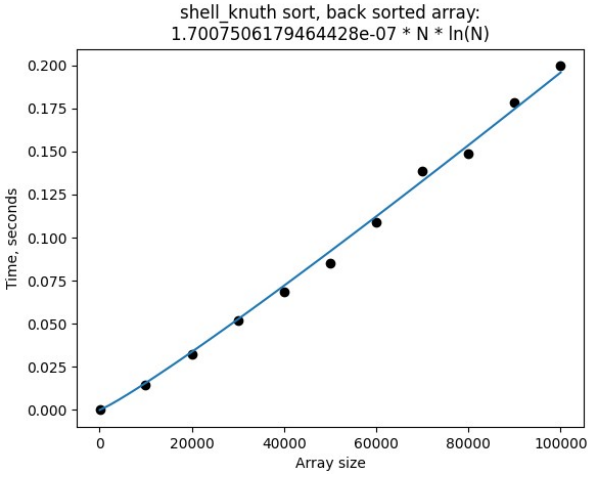


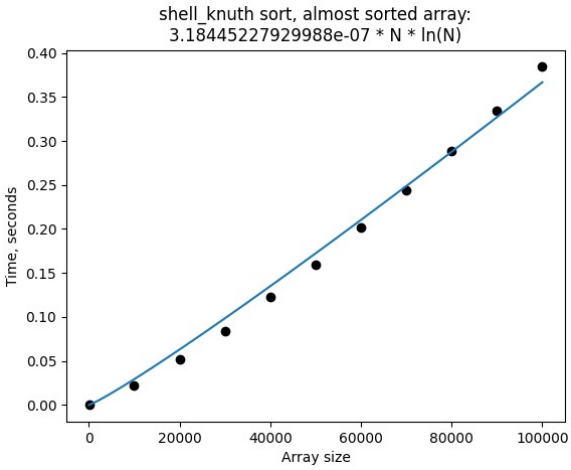

shell sort, back sorted array:
2.432633691691272e-07 * N * ln(N)
Time, seconds
Array size

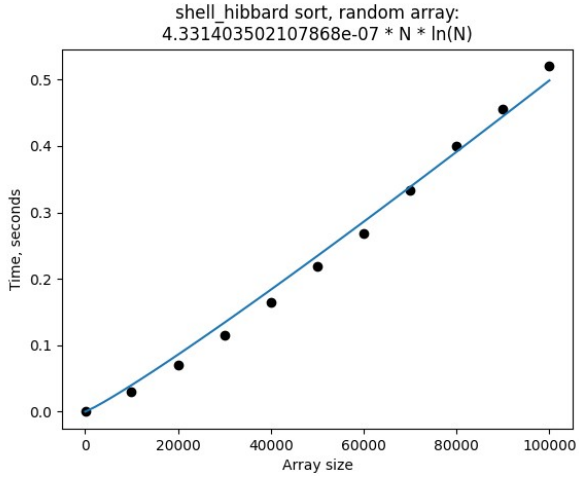

shell sort, almost sorted array:
1.4486768131910826e-08 * N^1,5
Time, seconds
Array size

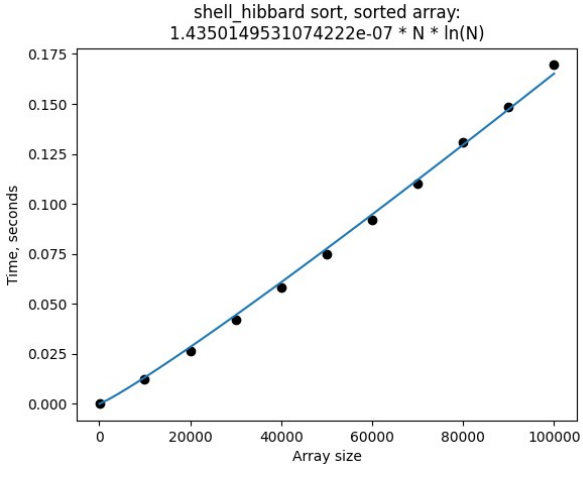

shell_knuth sort, random array:
4.322653134676921e-07 * N * ln(N)
Time, seconds
Array size

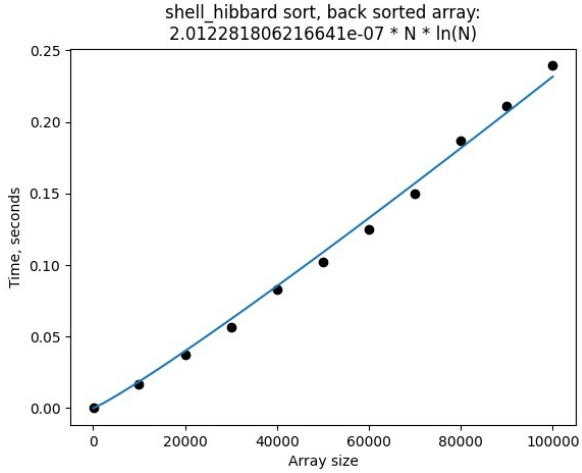

shell_knuth sort, sorted array:
9.470667611972386e-08 * N * ln(N)
Time, seconds
Array size

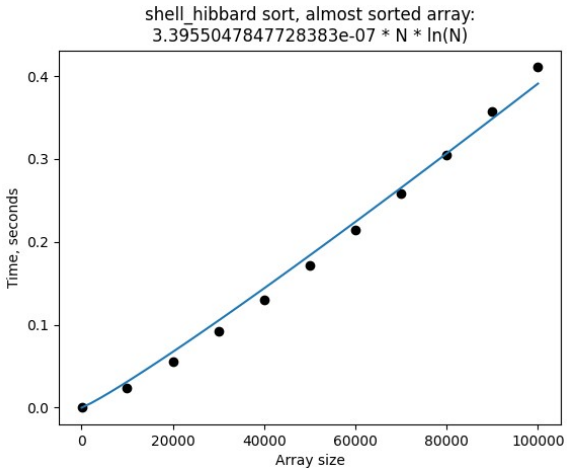

shell_knuth sort, back sorted array:
1.700750617946442 8e-07 * N * ln(N)
Time, seconds
Array size


shell_knuth sort, almost sorted array:
3.18445227929988e-07 * N * ln(N)
Time, seconds
Array size


shell_hibbard sort, random array:
4.331403502107868e-07 * N * ln(N)
Time, seconds
Array size


shell_hibbard sort, sorted array:
1.4350149531074222e-07 * N * ln(N)
Time, seconds
Array size





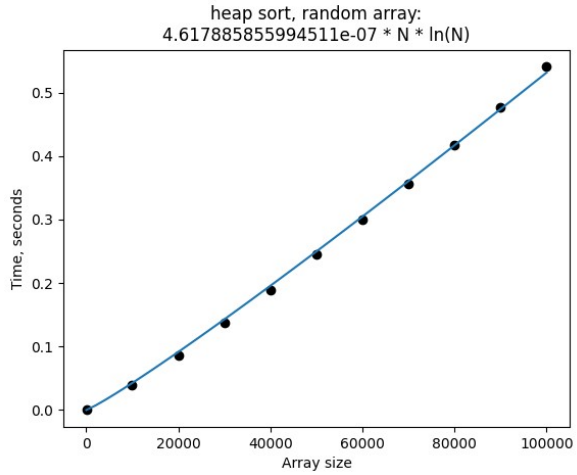
Худший случай — Θ(N^2) в первом и Θ(N^1.5) в двух других

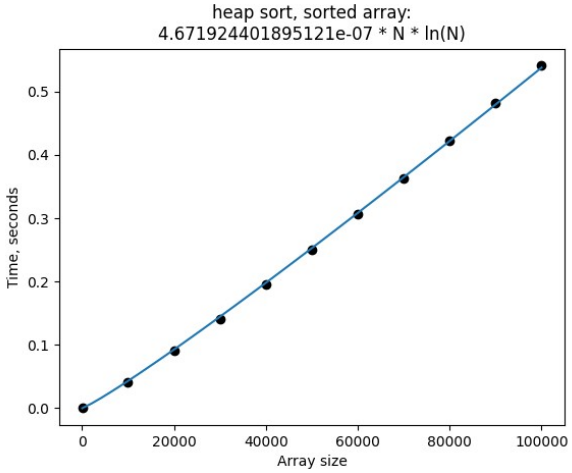
Средний случай — Θ(Nln(N))

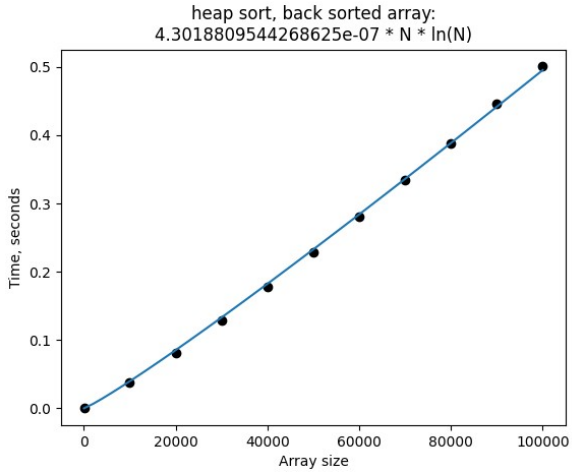
Лучший случай — Θ(Nln(N))

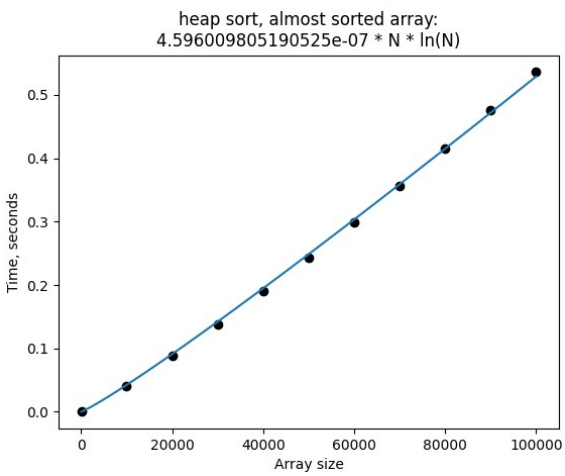
Из-за довольно большой погрешности некоторые зависимости не совпадают. Неустойчива.

7. heap_sort(array)









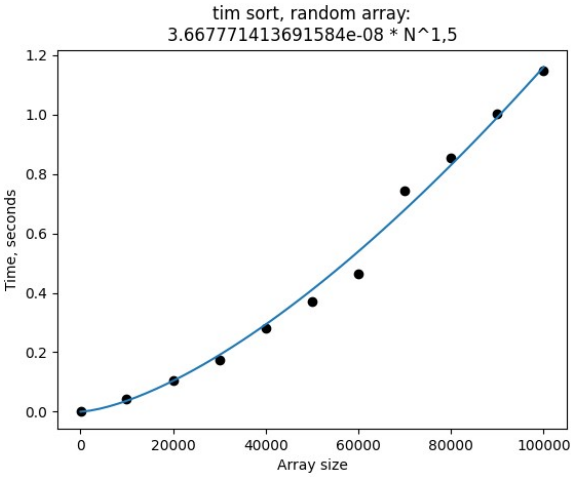
Худший случай — Θ(Nln(N))

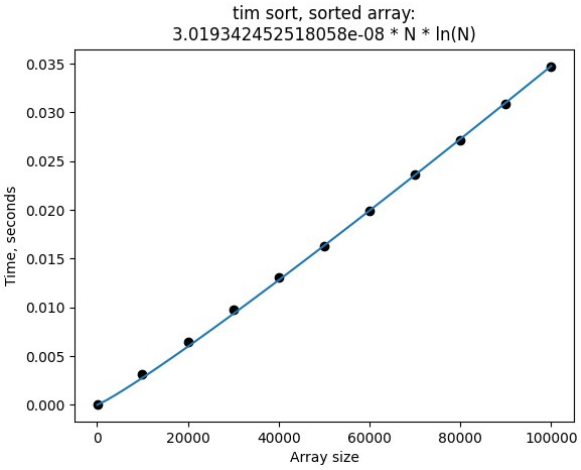
Средний случай — Θ(Nln(N))

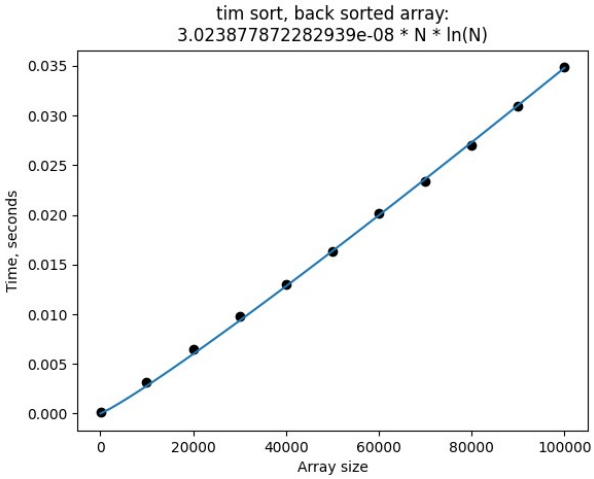
Лучший случай — Θ(Nln(N))

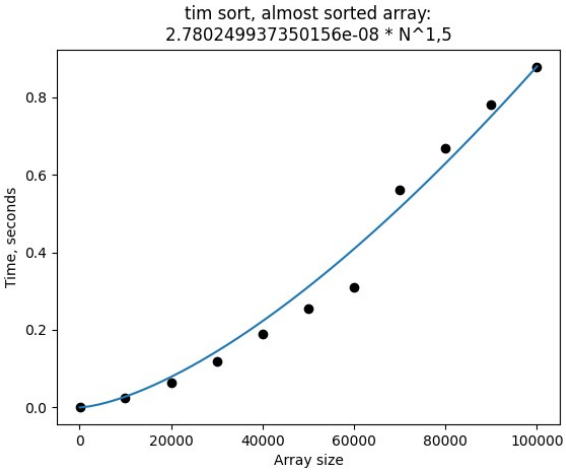
Экспериментальные данные совпадают с теоритическими. Неустойчива.

8. tim_sort(array)









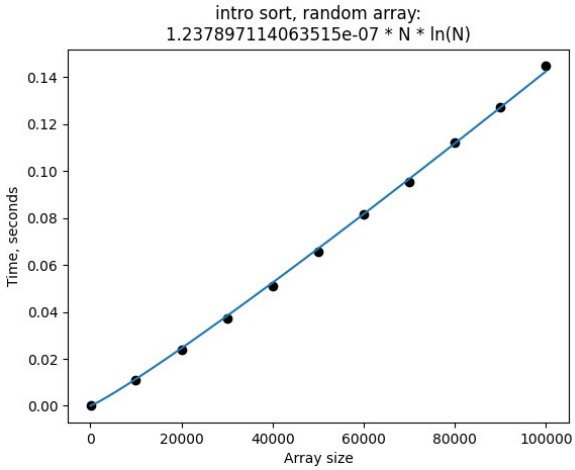
Худший случай — Θ(Nln(N))

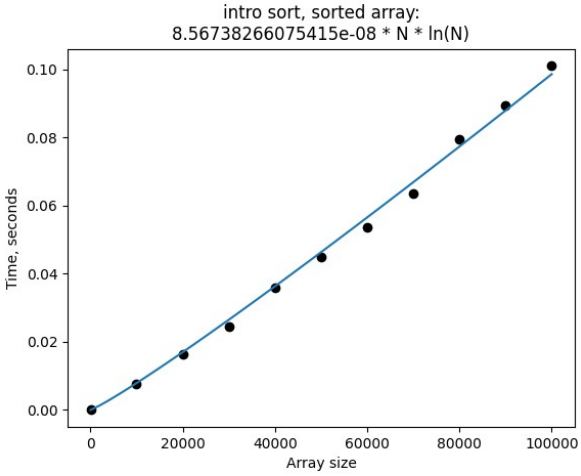
Средний случай — Θ(Nln(N))

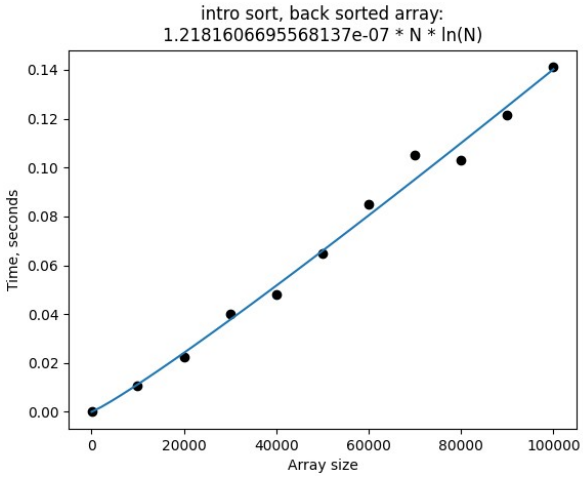
Лучший случай — Θ(N)

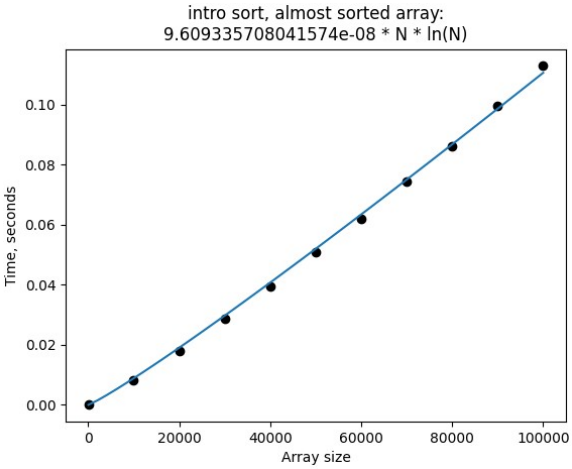
Из-за довольно большой погрешности зависимости не совпадают, однако можно видеть довольно быстрое выполение на прямо и обратно-отсортированных массивах. Устойчива.

9. intro_sort(array, depth=None)









Худший случай — Θ(Nln(N))

Средний случай — Θ(Nln(N))

Лучший случай — Θ(Nln(N))

Экспериментальные данные совпадают с теоритическими. Неустойчива.

Из файла result_low.json получим следующие данные о времени работы сортировок при N = 100 и случайных данных при ста повторениях одной точки (в секундах):

selection 0.00024138450622558595

insertion 0.0002324700355529785

bubble 0.000468285083770752

merge 0.00013326644897460938

quick 6.440162658691406e-05

shell 9.486913681030274e-05

shell_knuth 8.228302001953125e-05

shell_hibbard 8.91280174255371e-05

heap 0.00017186164855957032

tim 0.00014412879943847656

intro 6.150245666503906e-05

4. Вывод

На небольших данных хорошо работают quick, tim, shell и intro.

На больших — quick, tim и intro.

Однако из них только timsort устойчив, поэтому выбор сортировки должен основываться на размере массива и необходимости в устойчивости. Также в работе не была разобрана пространственная сложность, которая также должна влиять на выбор сортировки.

4. Пример работы

Запуск 2309ShubbeLPlr2.py выводит данные следующего формата:

```
SORT_METHOD ARRAY_MODE  N   TIME, seconds
selection random 50 6.747245788574219e-05
selection random 1000 0.024847745895385742
selection random 2000 0.1001887321472168
selection random 3000 0.22422575950622559
selection random 4000 0.3974313735961914
selection random 5000 0.6215808391571045
selection random 6000 0.8944754600524902
selection random 7000 1.2138140201568604
```

...

И записывает их в файл result.json

Запуск render.py выводит на экран графики, которые можно наблюдать во втором пункте.

5. Листинг

render.py

```
#!/usr/bin/env python3
import json
import matplotlib
import matplotlib.pyplot as plt
import numpy
from scipy.optimize import curve_fit
from scipy.optimize import differential_evolution
import warnings


def sum_of_squared_error(parameterTuple):  # function for genetic algorithm to minimize (sum of squared error)
    warnings.filterwarnings("ignore")
    return numpy.sum((yData - func(xData, *parameterTuple)) ** 2.0)


def generate_initial_parameters():
    parameterBounds = [[min(xData), max(xData)], [min(xData), max(xData)]]
    r = differential_evolution(sum_of_squared_error, parameterBounds, seed=3)
    return r.x


if __name__ == "__main__":
    try:
        with open("result_all.json", 'r') as file:
            result = json.load(file)["result"]
        functions = [
            lambda x, a, b: a * x,
            lambda x, a, b: a * x**2,
            lambda x, a, b: a * x**1.5,
            lambda x, a, b: a * x * numpy.log(x),
        ]
        for sort, sort_result in result.items():
            for mode, mode_result in sort_result.items():
                xData = numpy.array(list(map(float, mode_result.keys())))
                yData = numpy.array(list(map(float, mode_result.values())))


                min_MSE, i = None, 0
                for func in functions:
                    i += 1
                    genetic_parameters = generate_initial_parameters()
                    fitted_parameters, pcov = curve_fit(func, xData, yData, genetic_parameters)


                    model_predictions = func(xData, *fitted_parameters)
                    abs_error = model_predictions - yData
                    SE = numpy.square(abs_error)  # squared errors
                    MSE = numpy.mean(SE)  # mean squared errors
                    if min_MSE is None or min_MSE > MSE:
                        params, case, min_MSE = fitted_parameters, i, MSE
                    R_squared = 1.0 - (numpy.var(abs_error) / numpy.var(yData))


                if case == 1:
                    func_string = " * N"
                elif case == 2:
                    func_string = " * N^2"
                elif case == 3:
                    func_string = " * N^1,5"
                else:
                    func_string = " * N * ln(N)"
                xModel = numpy.linspace(min(xData), max(xData))
                yModel = functions[case - 1](xModel, *params)
                matplotlib.pyplot.scatter(xData, yData, color="#000000")
                matplotlib.pyplot.plot(xModel, yModel)
                matplotlib.pyplot.title(sort + " sort, " + mode + " array:\n" + str(params[0]) + func_string)# + " + " + str(params[1]))
                matplotlib.pyplot.xlabel("Array size")
                matplotlib.pyplot.ylabel("Time, seconds")
                matplotlib.pyplot.show()
    except AttributeError:
        pass
```

2309ShubbeLPlr2.py

```python
#!/usr/bin/env python3
import math
import time
import random
from enum import Enum
import json


def bubble_sort(array):
    n = len(array)
    for i in range(n):
        already_sorted = True
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
                already_sorted = False
        if already_sorted:
            break
    return array


def selection_sort(array):
    for i in range(len(array)):
        min_index = i
        for j in range(i + 1, len(array)):
            if array[j] < array[min_index]:
                min_index = j
        (array[i], array[min_index]) = (array[min_index], array[i])
    return array


def insertion_sort(array):
    for i in range(1, len(array)):
        key_item = array[i]
        j = i - 1
        while j >= 0 and array[j] > key_item:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = key_item
    return array


def shell_sort(array, mode=0):
    def get_first(N, mode=0):
        if mode == 0:  # Shell
            return 0, N // 2
        if mode == 1:  # Knuth
            iterator = 1
            while (3**iterator - 1) // 2 < N // 3:
                iterator += 1
            step = (3**iterator - 1) // 2
            return iterator - 1, step
        if mode == 2:  # Hibbard
            iterator = 1
            while 2**iterator - 1 < N:
                iterator += 1
            step = 2**(iterator - 1) - 1
            return iterator - 1, step

    def get_next(gap, iterator, mode=0):
        if mode == 0:  # Shell
            return 0, gap // 2
        if mode == 1:  # Knuth
            step = (3**iterator - 1) // 2
            return iterator - 1, step
        if mode == 2:  # Hibbard
            step = 2**(iterator - 1) - 1
```

```
            return iterator - 1, step

    N = len(array)
    iterator, gap = get_first(N, mode)
    while gap > 0:
        for i in range(gap, N):
            temp = array[i]
            j = i
            while j >= gap and array[j - gap] > temp:
                array[j] = array[j - gap]
                j -= gap
            array[j] = temp
        iterator, gap = get_next(gap, iterator, mode)
    return array


def quick_sort(array):
    if len(array) <= 1:
        return array
    else:
        left = array[0]
        mid = array[len(array) // 2]
        right = array[-1]
        if left >= right:
            if right >= mid:
                sup = right
            elif left >= mid:
                sup = mid
            else:
                sup = left
        else:
            if right <= mid:
                sup = right
            elif left <= mid:
                sup = mid
            else:
                sup = left
        left_array = []
        right_array = []
        mid_array = []
        for el in array:
            if el < sup:
                left_array.append(el)
            elif el > sup:
                right_array.append(el)
            else:
                mid_array.append(el)
        return quick_sort(left_array) + mid_array + quick_sort(right_array)


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

```
            array += right[j:]
        return array

    if len(array) < 2:
        return array
    mid = len(array) // 2
    return merge(left=merge_sort(array[:mid]), right=merge_sort(array[mid:]))


def heap_sort(array):
    def heapify(array, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2
        if l < n and array[i] < array[l]:
            largest = l
        if r < n and array[largest] < array[r]:
            largest = r
        if largest != i:
            (array[i], array[largest]) = (array[largest], array[i])
            heapify(array, n, largest)
        return array

    n = len(array)
    for i in range(n // 2 - 1, -1, -1):
        array = heapify(array, n, i)
    for i in range(n - 1, 0, -1):
        (array[i], array[0]) = (array[0], array[i])
        heapify(array, i, 0)
    return array


def tim_sort(array):
    def find_min_run(N):
        R = 0
        while N >= TIM_SORT_MIN_RUN:
            R |= N & 1
            N >>= 1
        return N + R

    def binary_search(array, left, right, x):
        while left <= right:
            mid = (right + left) // 2
            if array[mid] == x:
                return mid
            elif array[mid] < x:
                left = mid + 1
            else:
                right = mid - 1
        return left

    def merge(left, right):
        N = len(left)
        K = len(right)
        if N == 0:
            return right
        if K == 0:
            return left
        array = []
        i = j = 0
        galop_l = galop_r = TIM_SORT_GALOP
        while i < N and j < K:
            if left[i] <= right[j]:
                if galop_l >= TIM_SORT_GALOP:
                    index = binary_search(left, i, N - 1, right[j])
                    array += left[i:index]
                    i = index
                    galop_l = 0
                    continue
                array.append(left[i])
```

```
            i += 1
            galop_l += 1
            galop_r = 0
        else:
            if galop_r >= TIM_SORT_GALOP:
                index = binary_search(right, j, K - 1, left[i])
                array += right[j:index]
                j = index
                galop_r = 0
                continue
            array.append(right[j])
            j += 1
            galop_r += 1
            galop_l = 0
    if j == K:
        array += left[i:]
    if i == N:
        array += right[j:]
    return array

def check(runs):
    while len(runs) > 2:
        run1 = runs.pop()
        run2 = runs.pop()
        run3 = runs.pop()
        if len(run1) > len(run2) + len(run3):
            if len(run2) < len(run3):
                runs.append(run3)
                runs.append(merge(run1, run2))
            else:
                runs.append(run3)
                runs.append(run2)
                runs.append(run1)
                break
        else:
            if len(run1) > len(run3):
                runs.append(merge(run2, run3))
                runs.append(run1)
            else:
                runs.append(run3)
                runs.append(merge(run1, run2))
    return runs

N = len(array)
if N <= TIM_SORT_MIN_RUN:
    return insertion_sort(array)
min_run = find_min_run(N)
runs_stack = []
start = index = 0
cntr = 0
increasing = None
while index < N - 1:
    if array[index] == array[index + 1]:
        index += 1
        cntr += 1
        continue
    if array[index] < array[index + 1]:
        if increasing is None:
            increasing = True
            index += 1
            continue
        if increasing is True:
            index += 1
            continue
        index = min(max(index, start + min_run), N)
        run = insertion_sort(array[start:index])
    else:
        if increasing is None:
            increasing = False
            index += 1
```

```
                continue
            if increasing is False:
                index += 1
                continue
            index = min(max(index, start + min_run), N)
            run = insertion_sort(array[start:index][::-1])
        start = index
        increasing = None
        runs_stack.append(run)
        runs_stack = check(runs_stack)
    if increasing is True:
        runs_stack.append(insertion_sort(array[start:]))
    if increasing is False:
        runs_stack.append(insertion_sort(array[start:][::-1]))
    while i := len(runs_stack) > 1:
        if i > 2 and len(runs_stack[-3]) < len(runs_stack[-1]):
            i -= 1
        left = runs_stack.pop(i - 2)
        right = runs_stack.pop(i - 2)
        runs_stack.insert(i - 2, merge(left, right))
    return runs_stack[0]


def intro_sort(array, depth=None):
    if depth is None:
        depth = int(math.log2(len(array)) * 2)
    if len(array) <= INTRO_SORT_INSERTION_SIZE:
        return insertion_sort(array)
    if depth == 0:
        return heap_sort(array)
    else:
        left = array[0]
        mid = array[len(array) // 2]
        right = array[-1]
        if left >= right:
            if right >= mid:
                sup = right
            elif left >= mid:
                sup = mid
            else:
                sup = left
        else:
            if right <= mid:
                sup = right
            elif left <= mid:
                sup = mid
            else:
                sup = left
        left_array = []
        right_array = []
        mid_array = []
        for el in array:
            if el < sup:
                left_array.append(el)
            elif el > sup:
                right_array.append(el)
            else:
                mid_array.append(el)
        return intro_sort(left_array, depth - 1) + mid_array + intro_sort(right_array, depth - 1)


def generate(N, min_el, max_el, mode):  # Генерация случайного списка
    new_array = [random.randint(min_el, max_el) for _ in range(N)]
    if mode == Modes.SORTED:
        new_array.sort(reverse=False)
    elif mode == Modes.BACK_SORTED:
        new_array.sort(reverse=True)
    elif mode == Modes.ALMOST_SORTED:
        new_array.sort(reverse=False)
        for i in range(N // 10):
```

```
            new_array[random.randint(0, N - 1)] = random.randint(min_el, max_el)
    return new_array


def check_sort(func):  # Получение времени работы сортировки и её правильности
    times = {}
    for mode in Modes:
        times[mode.value] = {}
        for N in POINTS:
            times[mode.value][N] = 0
            for i in range(REPEAT):
                array = generate(N, MIN_EL, MAX_EL, mode)
                ts = time.time()
                sorted_array = func(array)
                if times[mode.value][N] != -1:
                    times[mode.value][N] += time.time() - ts
                if sorted_array != sorted(array):
                    times[mode.value][N] = -1
            times[mode.value][N] /= REPEAT
            print(list(sorts.keys())[list(sorts.values()).index(func)], mode.value, N,
times[mode.value][N])
    return times


class Modes(Enum):
    RANDOM = 'random'
    SORTED = 'sorted'
    BACK_SORTED = 'back sorted'
    ALMOST_SORTED = 'almost sorted'


sorts = {
    'selection': selection_sort,
    'insertion': insertion_sort,
    'bubble': bubble_sort,
    'merge': merge_sort,
    'quick': quick_sort,
    'shell': shell_sort,
    'shell_knuth': lambda array: shell_sort(array, 1),
    'shell_hibbard': lambda array: shell_sort(array, 2),
    'heap': heap_sort,
    'tim': tim_sort,  # ~ 40 times slower than built-in
    'intro': intro_sort,  # ~ 30 times slower than built-in
    'built-in': sorted,
    'not-working': lambda array: random.shuffle(array)  # to show all sorts are properly working
}
TIM_SORT_MIN_RUN = 64
TIM_SORT_GALOP = 7
INTRO_SORT_INSERTION_SIZE = 16
MIN_EL = 0  # Минимальное допустимое значение элемента
MAX_EL = 10**6  # Максимальное допустимое значение элемента
POINTS = list(range(50, 200, 10))  # Точки (количества элементов в списке для разных вызовов
функции)
REPEAT = 100  # Times one point will be executed, then average from all results

if __name__ == "__main__":
    output = {"settings": {"REPEAT": REPEAT, "MIN_EL": MIN_EL, "MAX_EL": MAX_EL, "POINTS":
POINTS}, "result": {}}
    print('SORT_METHOD\tARRAY_MODE\tN\tTIME, seconds')
    for key, sort in sorts.items():
        output["result"][key] = check_sort(sort)
    with open('result.json', 'w') as file:
        json.dump(output, file, indent=4)
    print('Successfully finished, check result.json')
```

Репозиторий с кодом и отчётом: https://github.com/ShubbeLeontij/aisd_lab1