**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

1. 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

1. Постараться определить асимптотику для лучшего/среднего/худшего случая путем анализа экспериментальных данных и применения к ним нелинейной регрессии.
2. Постараться получить достоверные данные о времени работы алгоритмов сортировки при малом размере массива ~ 100 элементов

**2. Временная и практическая сложность**

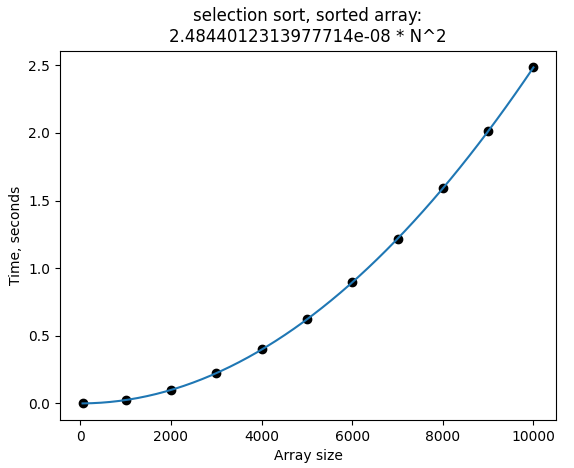
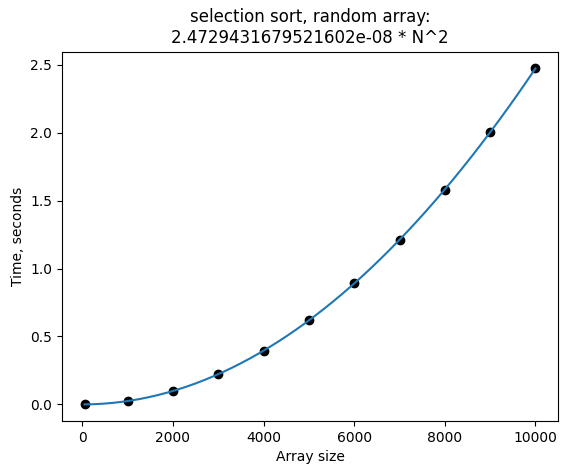
Каждая сортировка была вызвана с четырмя необходмимыми вариантами массива. Для каждого такого варианта массив генериворался 10 раз и запускался механизм подсчёта времени работы. Далее все времена усреднялись. Также производилась проверка на правильность сортировки, в случае неверной сортировки вместо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

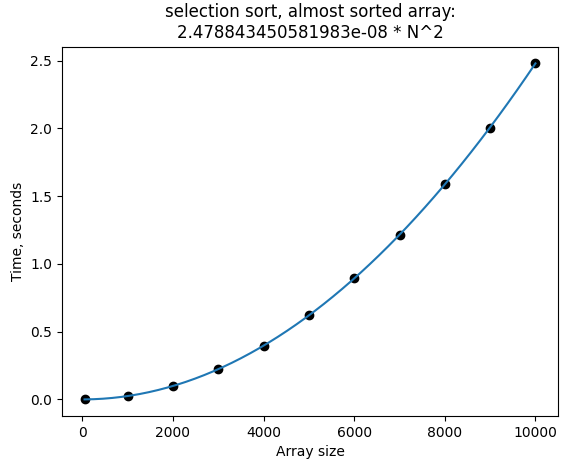
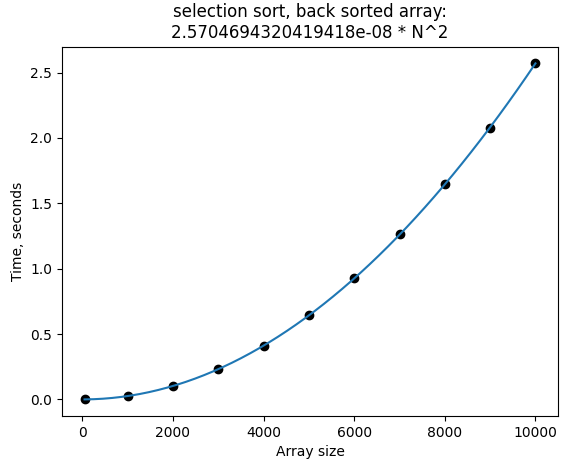
* Для всех сортировок измерены времена для размеров 50 и 1000, 2000, … 10000 — result\_all.json
* Для быстрых сортировок (все кроме первых трёх) измерены времена для размеров 50 и 10000, 20000, … 100000 — result\_fast.json
* Для всех сортировок измерены времена для размеров 50, 60, … 200 — result\_low.json

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)



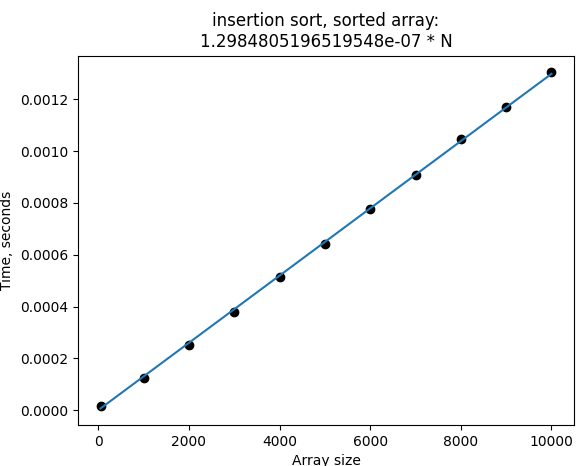


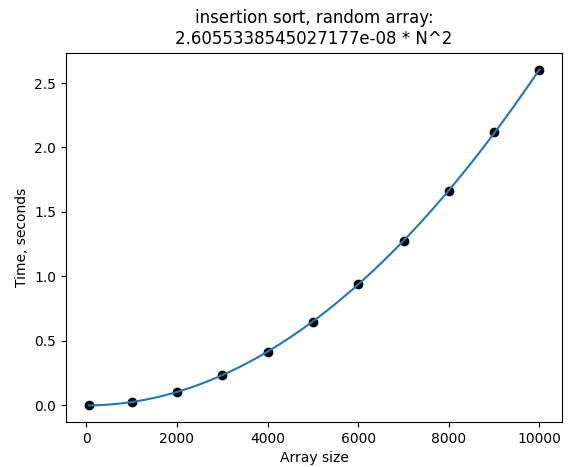
Худший случай — Θ(N^2)

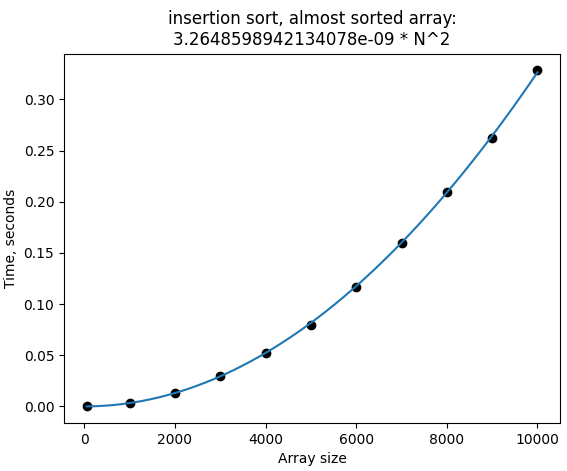
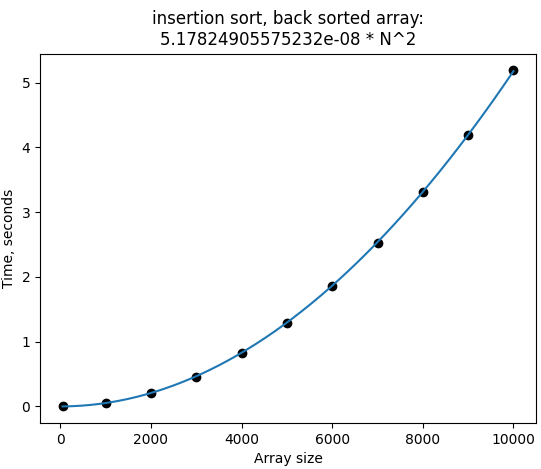
Средний случай — Θ(N^2)

Лучший случай — Θ(N^2)

Экспериментальные данные совпадают с теоритическими. Неустойчива.

1. insertion\_sort(array)





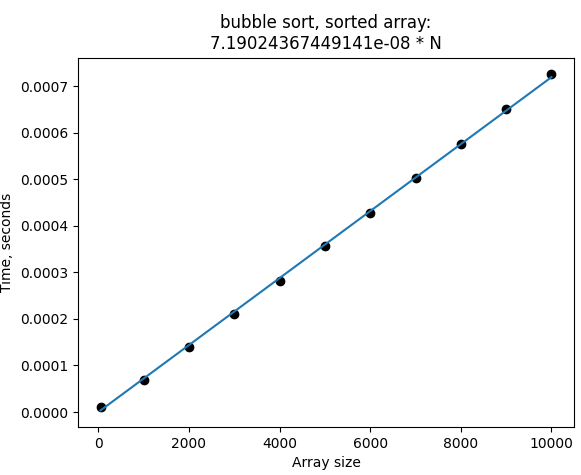
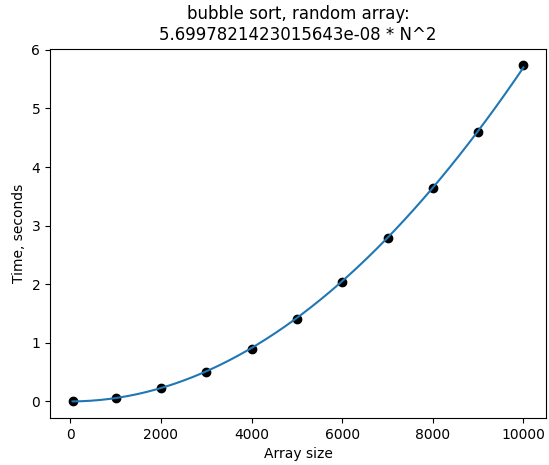
Худший случай — Θ(N^2)

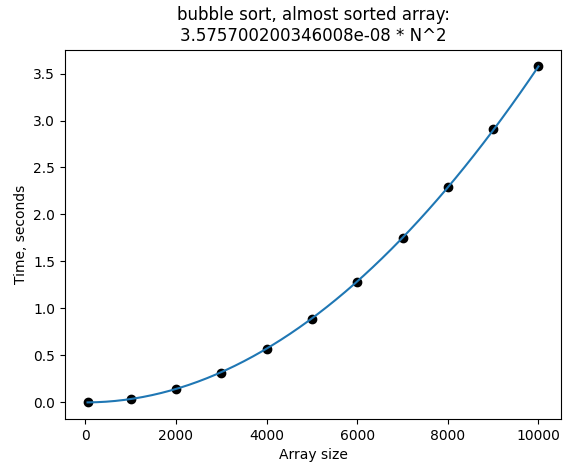
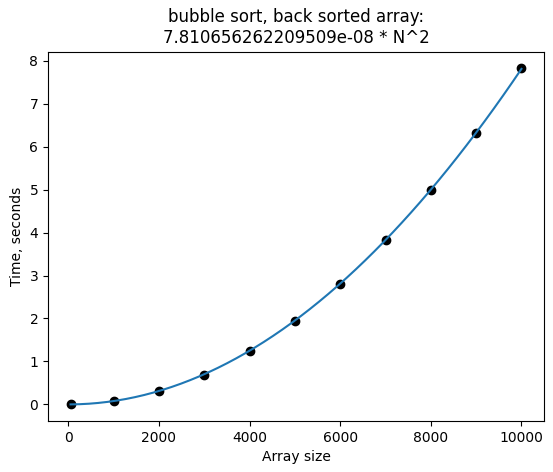
Средний случай — Θ(N^2)

Лучший случай — Θ(N)

Экспериментальные данные совпадают с теоритическими. Устойчива.

1. bubble\_sort(array)





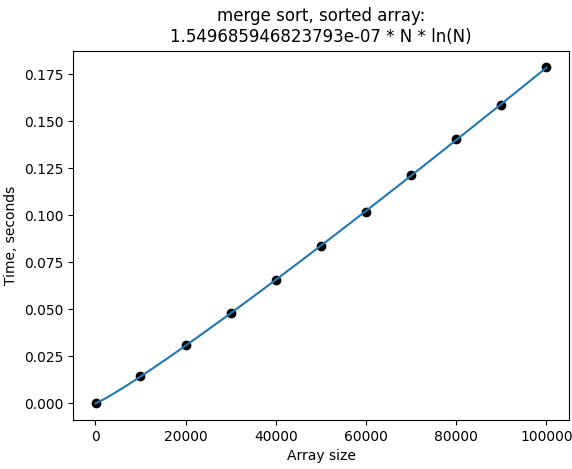
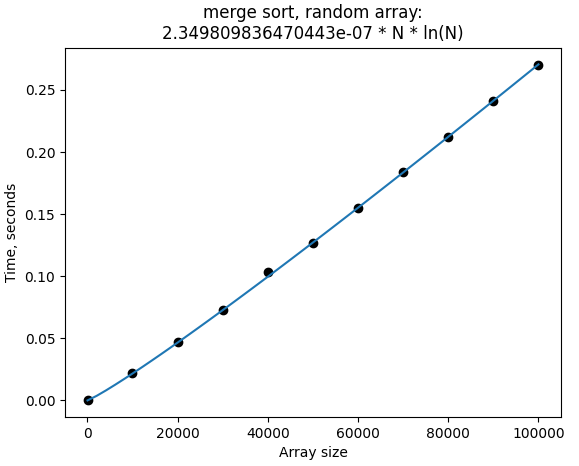
Худший случай — Θ(N^2)

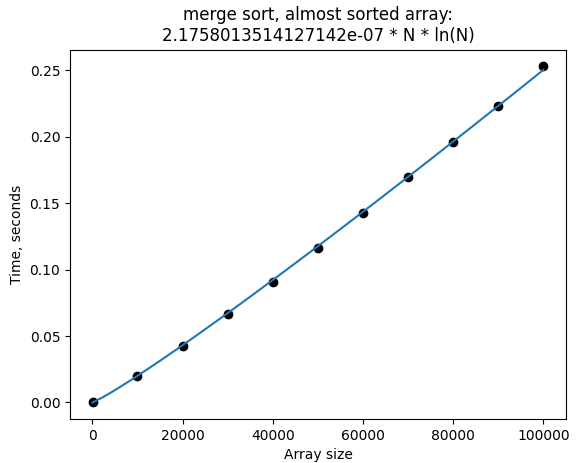
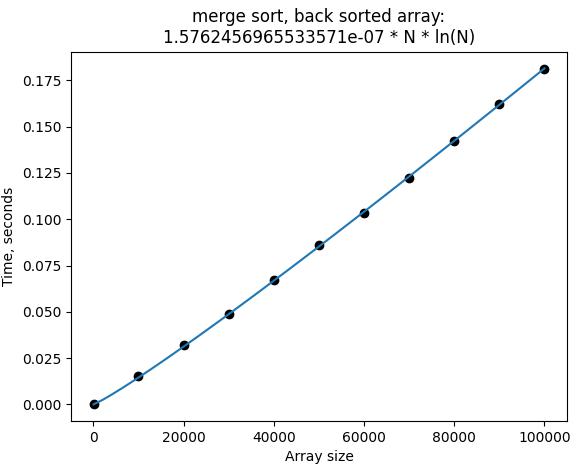
Средний случай — Θ(N^2)

Лучший случай — Θ(N)

Экспериментальные данные совпадают с теоритическими. Устойчива.

1. merge\_sort(array)





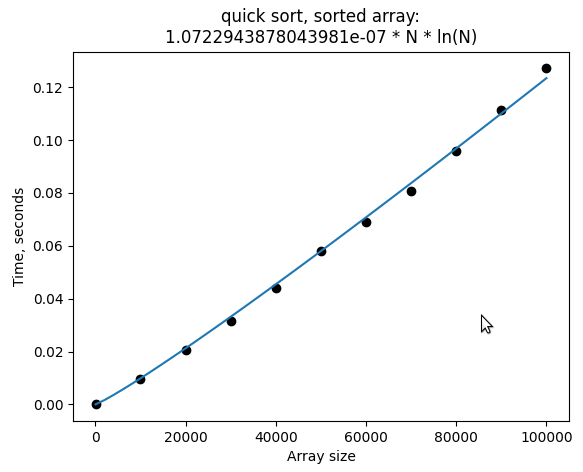
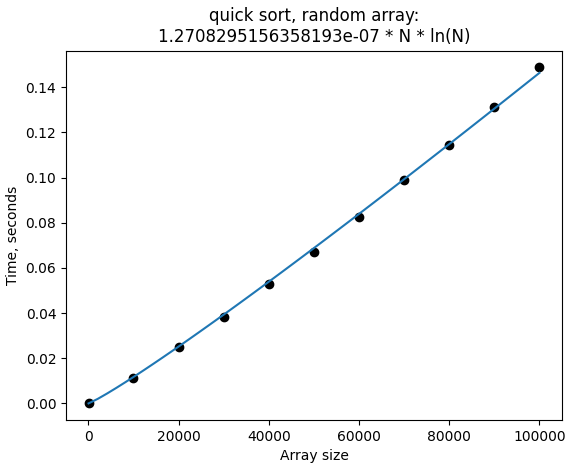
Худший случай — Θ(Nln(N))

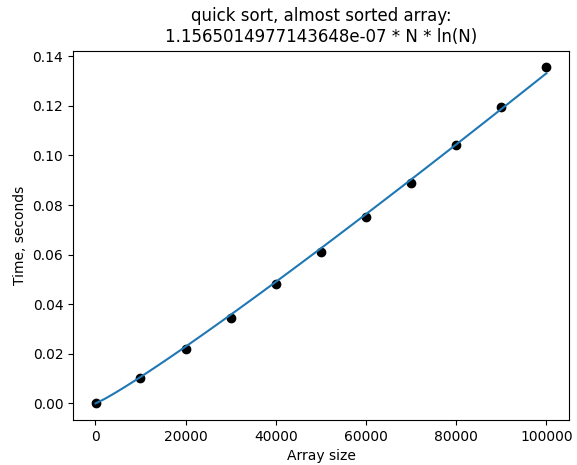
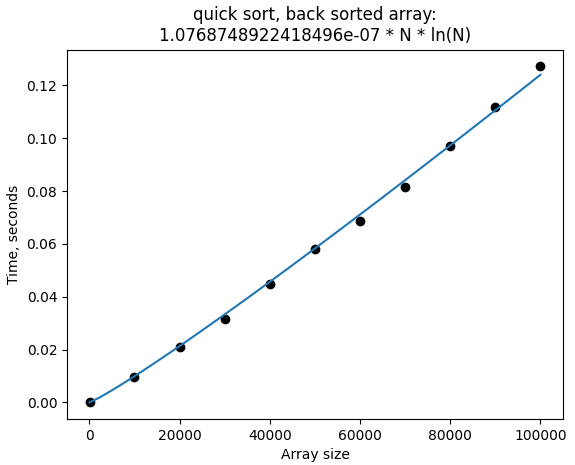
Средний случай — Θ(Nln(N))

Лучший случай — Θ(Nln(N))

Экспериментальные данные совпадают с теоритическими. Устойчива.

1. quick\_sort(array)





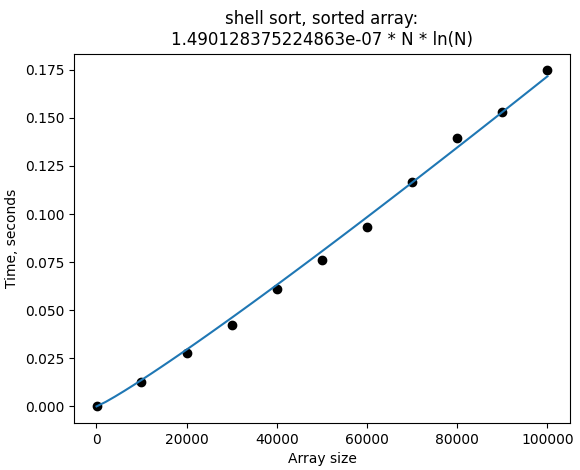
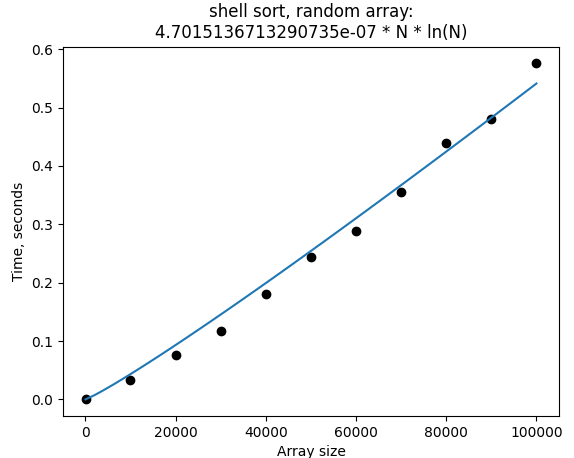
Худший случай — Θ(N^2)

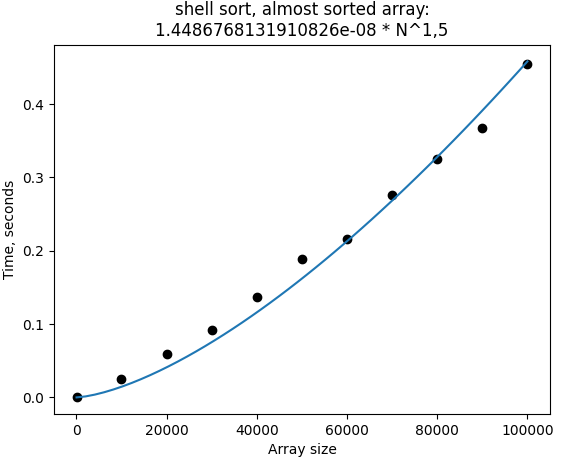
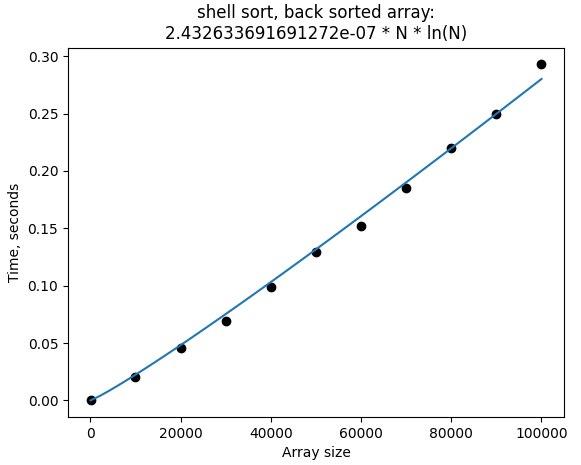
Средний случай — Θ(Nln(N))

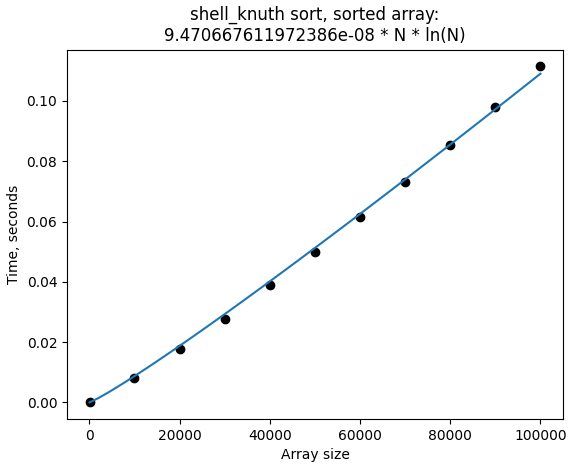
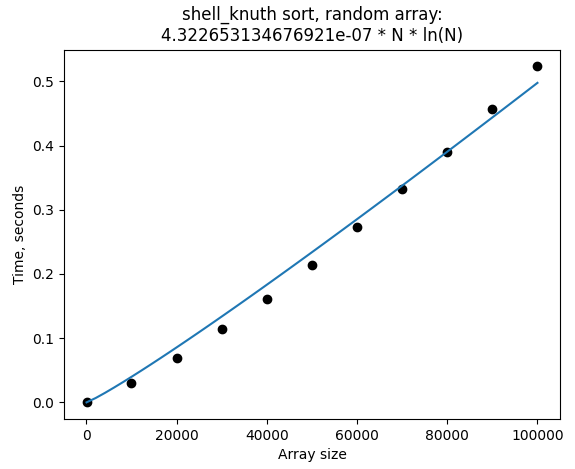
Лучший случай — Θ(Nln(N))

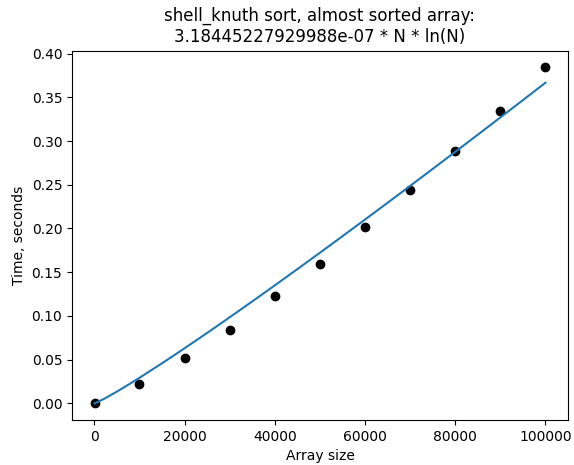
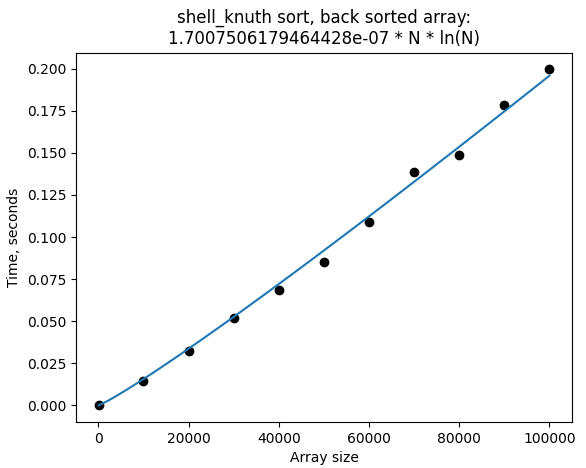
В выборке не представлен худший случай поэтому экспериментальные данные совпадают с теоритическими. Неустойчива.

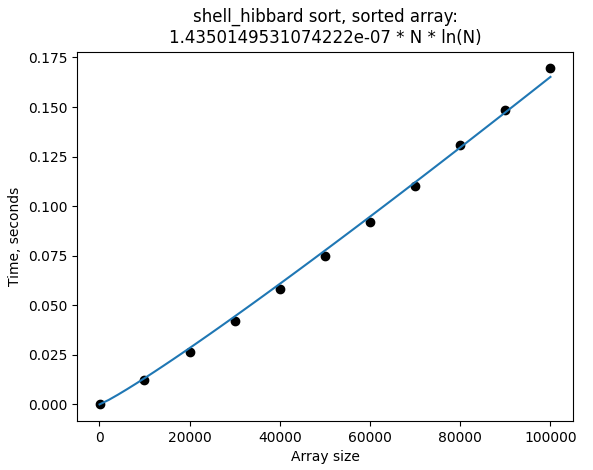
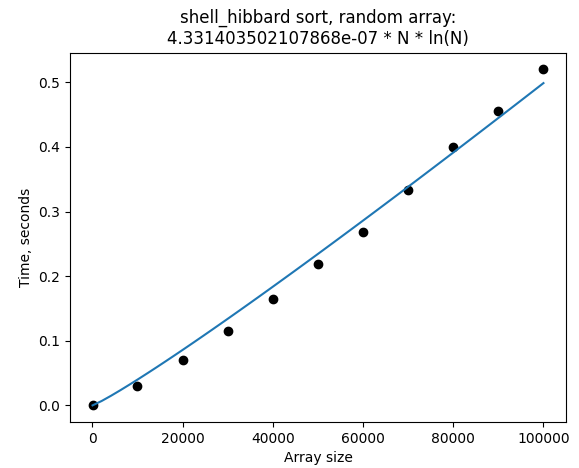
1. shell\_sort(array, mode=0)

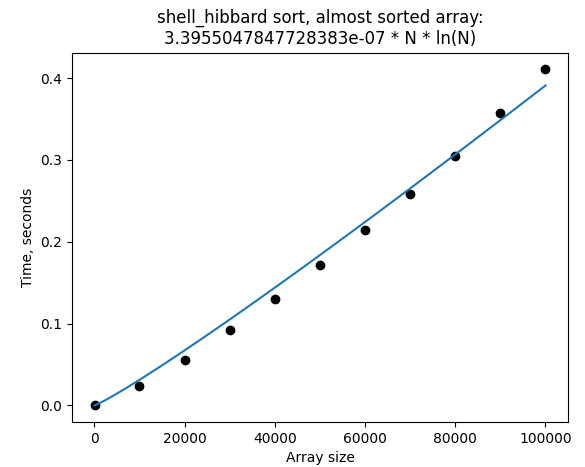
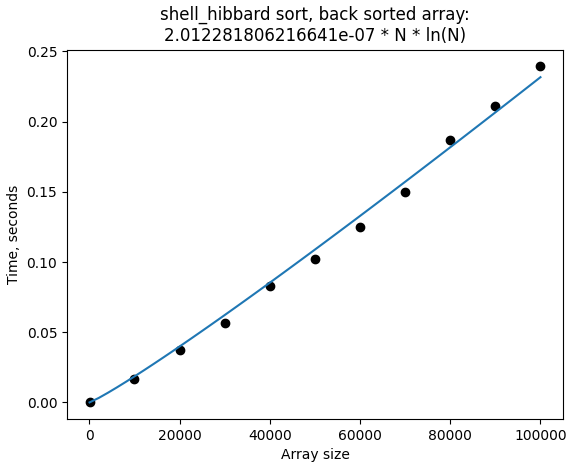










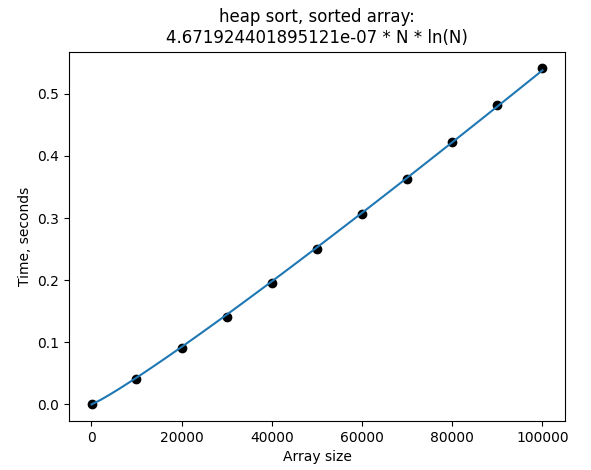


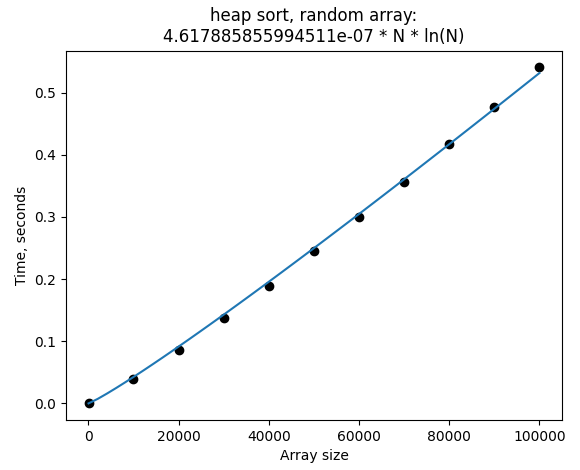
Худший случай — Θ(N^2) в первом и Θ(N^1.5) в двух других

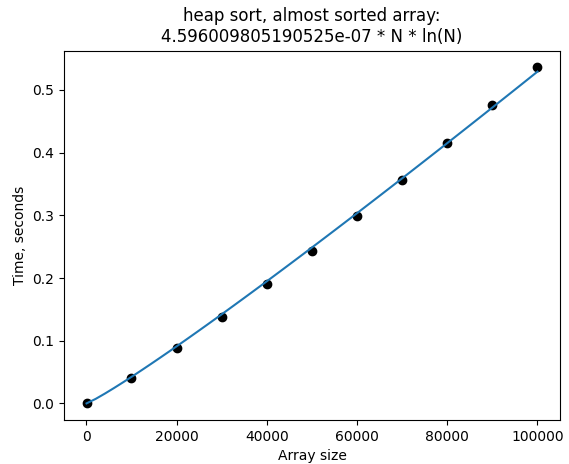
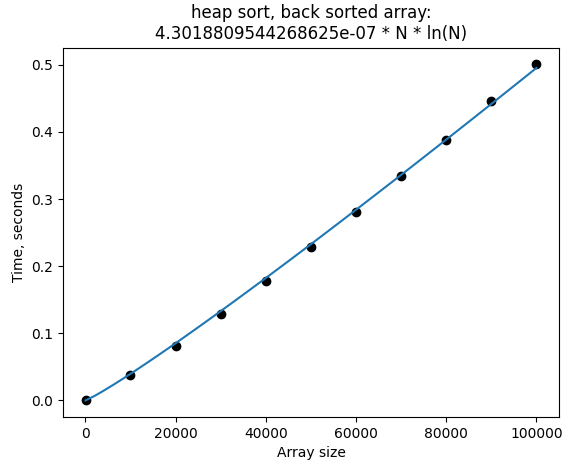
Средний случай — Θ(Nln(N))

Лучший случай — Θ(Nln(N))

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

1. heap\_sort(array)



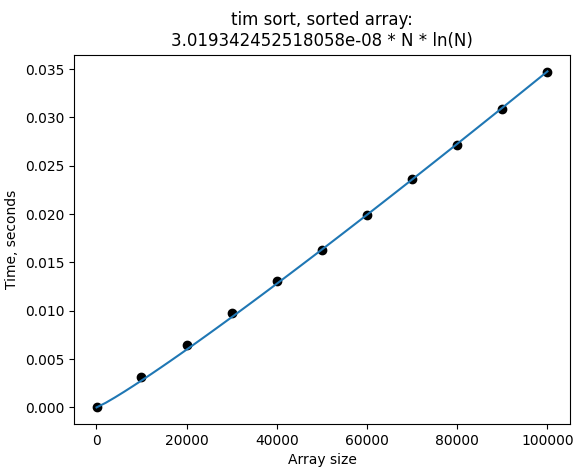


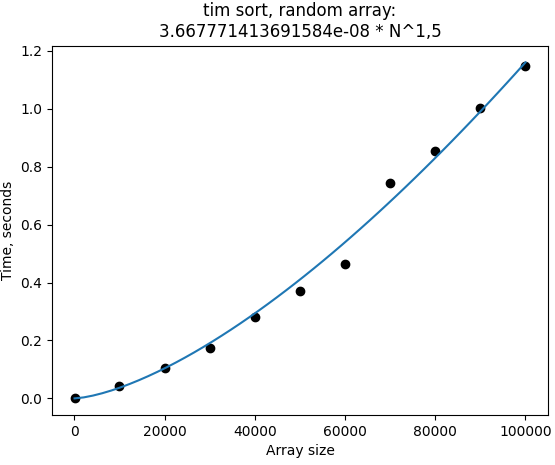
Худший случай — Θ(Nln(N))

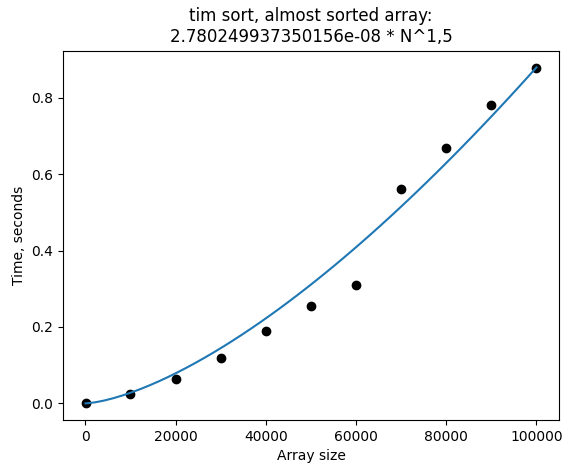
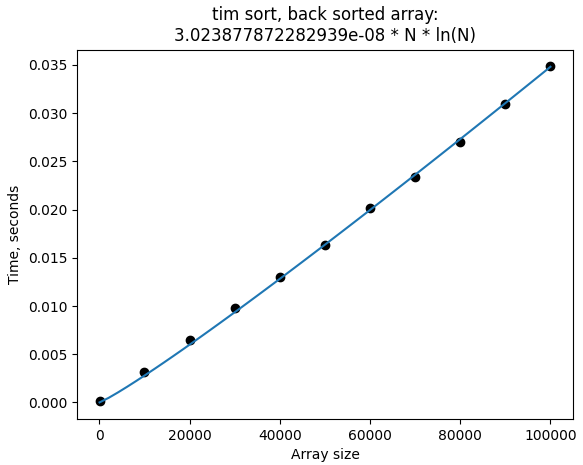
Средний случай — Θ(Nln(N))

Лучший случай — Θ(Nln(N))

Экспериментальные данные совпадают с теоритическими. Неустойчива.

1. tim\_sort(array)



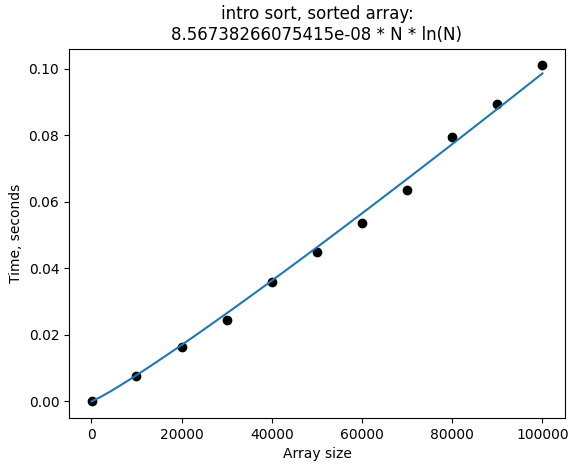


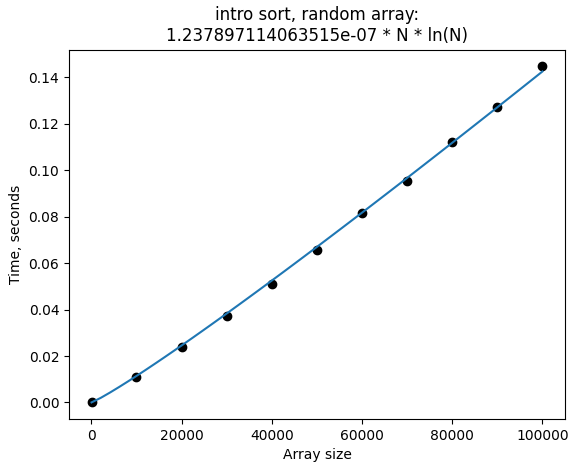
Худший случай — Θ(Nln(N))

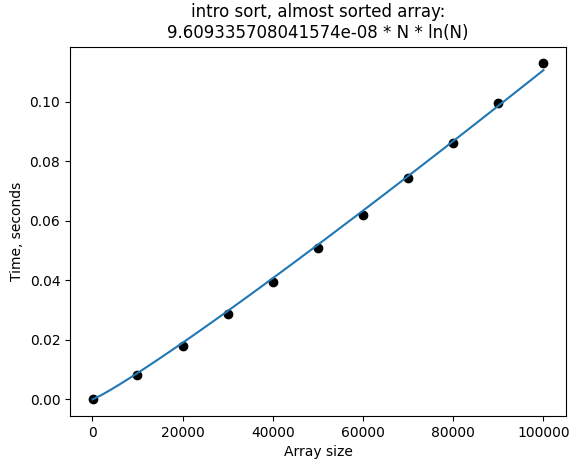
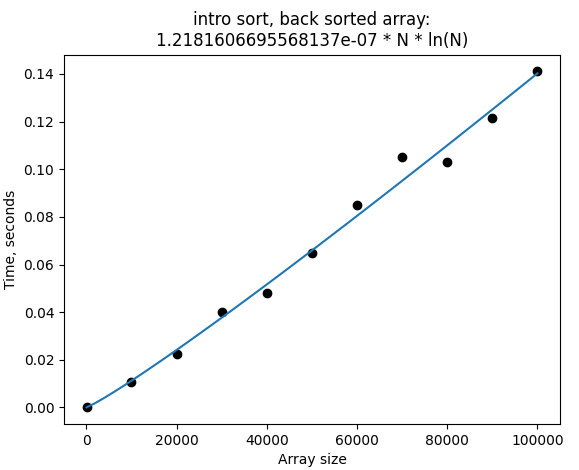
Средний случай — Θ(Nln(N))

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.

1. 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

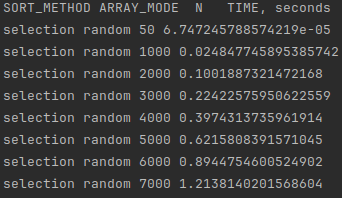
**4. Вывод**

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:

...

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**

#!/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**\t**ARRAY\_MODE**\t**N**\t**TIME, 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