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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Реализация и исследование алгоритма сортировки TimSort

Студентка гр. 2384 _____ Валеева А.А.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2023

Цель работы

Изучить алгоритм сортировки TimSort и реализовать ее без использования стандартных методов сортировки.

Задание

Реализация

Имеется массив данных для сортировки *int arr[]* размера *n*. Необходимо отсортировать его алгоритмом сортировки Timsort по следующему критерию: по наименьшему значению квадрата элемента (в случае равенства значений элементов в квадрате - сортировка происходит по убыванию). Так как Timsort - это гибридный алгоритм, содержащий в себе сортировку слиянием и сортировку вставками, то вам предстоит использовать оба этих алгоритма. Поэтому нужно выводить разделённые блоки, которые уже отсортированы сортировкой вставками.

Исследование

После успешного решения задачи в рамках курса проведите исследование данной сортировки на различных размерах данных (10/1000/100000), сравнив полученные результаты с теоретической оценкой (для лучшего, среднего и худшего случаев), и разного размера min_run. Результаты исследования предоставьте в отчете.

Примечание:

Нельзя пользоваться готовыми библиотечными функциями для сортировки, нужно сделать реализацию сортировки вручную.

0 < n < 65

Обратите внимание на пример. ***(min_run = 32)***

Формат входа.

Первая строка содержит натуральное число n - размерность массива, следующая строка содержит элементы массива через пробел.

Формат выхода.

Выводятся разделённые блоки для сортировки в формате "Part i: *отсортированный разделённый массив*". Последняя строчка содержит финальный результат сортировки массива с надписью "Answer: "

Выполнение работы

Описание кода:

Файл *my_sort.py*

В данном файле была реализован алгоритм сортировки *Timesort*, использующий также следующие функции:

1) *get_len_run(arr, start)* — функция, анализирующая переданный на вход массив и ищущая *run*, возвращает длину найденного run. Работает следующим образом: проходим по массиву с поставленной точки до конца, первые два элемента включены в *run* по дефолту, по ним определяется возвращающий или убывающий будет массив через переменную *is_ascending.* Далее анализируем перебираемые элементы и увеличиваем длину *run*, если текущий элемент подходит. Если массив шел в обратном порядке, то переворачиваем его через функцию *reversed*.

2) *def insertion_sort(arr, start, end)* — функция сортировки вставками. Реализована классическим образом, сравнение элементов происходит через квадрат числа, если же он равен, то сортируется по убыванию.

3) *def merge(arr, start, mid, end)* — функция сортировки слиянием, на вход котороq передаются две части массива (два *run),* далее элементы из каждого из данных массивов анализируются и в нужном порядку добавляются в исходный переданный в функцию массив *arr*. Сравнение элементов происходит тем де способом, как и в предыдущей функции.

4) Заключительная функция, реализующая алгоритм сортировки *Timsort* — *def timsort(arr).* Для начала делаем поиск run через их длину, получаемую через функцию *get_len_run*, если данное значение меньше *min_run*, равному 32, то мы дополняем *run* следующими недостающими элементами. В таком случае мы получаем отчасти отсортированную часть массива, для который мы применяем функцию *insertion_sort* (работает быстро, так как подмассивы

небольшого размера). Получаем массив из отсортированных *run*-в, теперь с помощью сортировки *merge* сортируем итоговый массив, передавая в нее пары *run*-в. Данная функция работает оптимально, так как есть отсортированные подмассивы.

Файл *main.py*

В основном файле импортируется функция сортировки *timesort* из файла *my_sort*. Далее принимаются введенные пользователем данные – число элементов массива и сам массив. К массиву применяется функция сортировки и выводится отсортированный массив.

Файл *test_main.py*

В файле *tests.py* содержаться тесты для проверки работоспособности написанной функции *timsort*, а также дополнительно описанных функций.

Исходный код программы представлен в приложении А.

Результаты тестирования представлены в приложении В.

Исследование сложности сортировки

Далее была протестирована сортировка на различных размерах данных (10/1000/100000), проведено сравнение полученных результатов с теоретической оценкой (для лучшего, среднего и худшего случаев), и разного размера *min_run*. Результаты исследования предоставьте в отчете.

Результаты исследования:

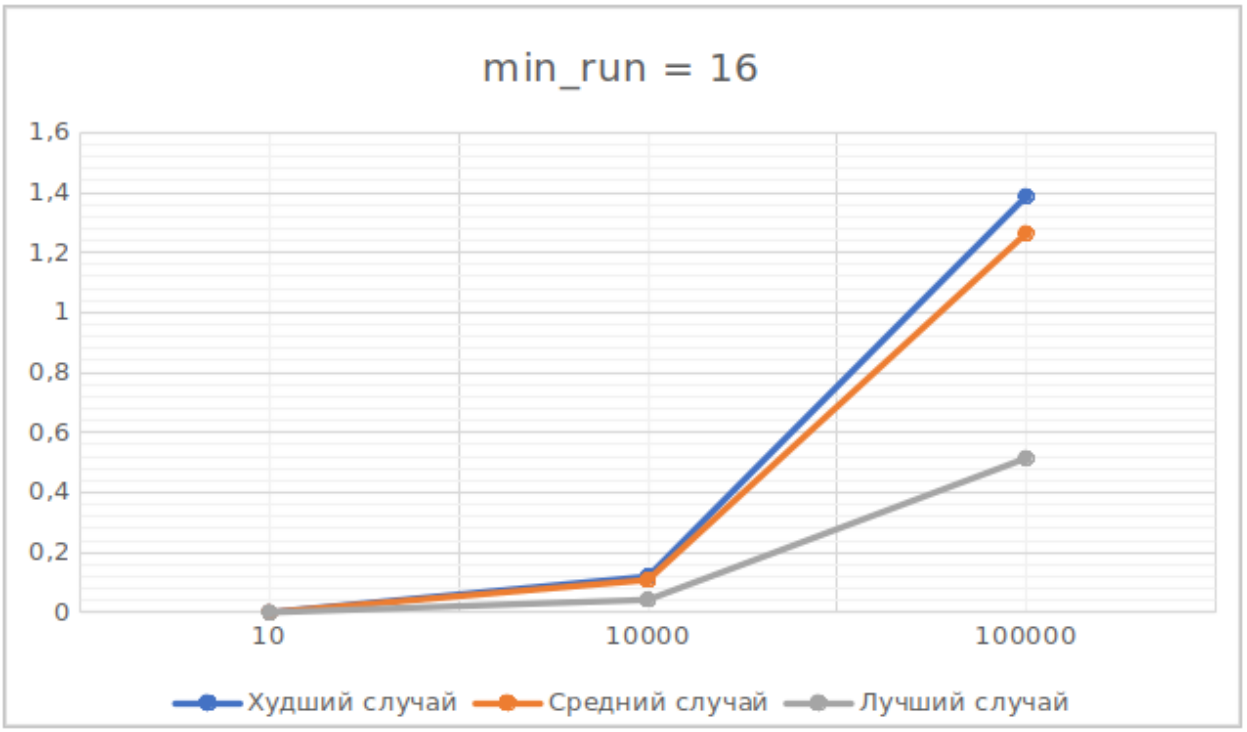

Рисунок 1 — Значение *min_run* = 16

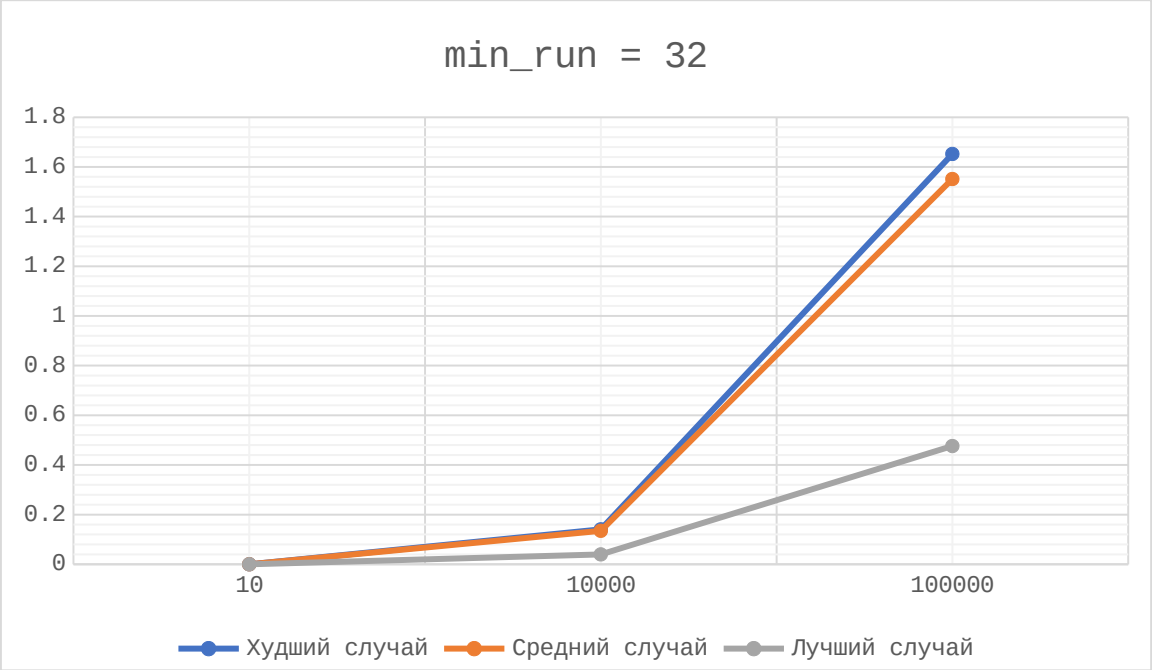

Рисунок 2 — Значение *min_run* = 32

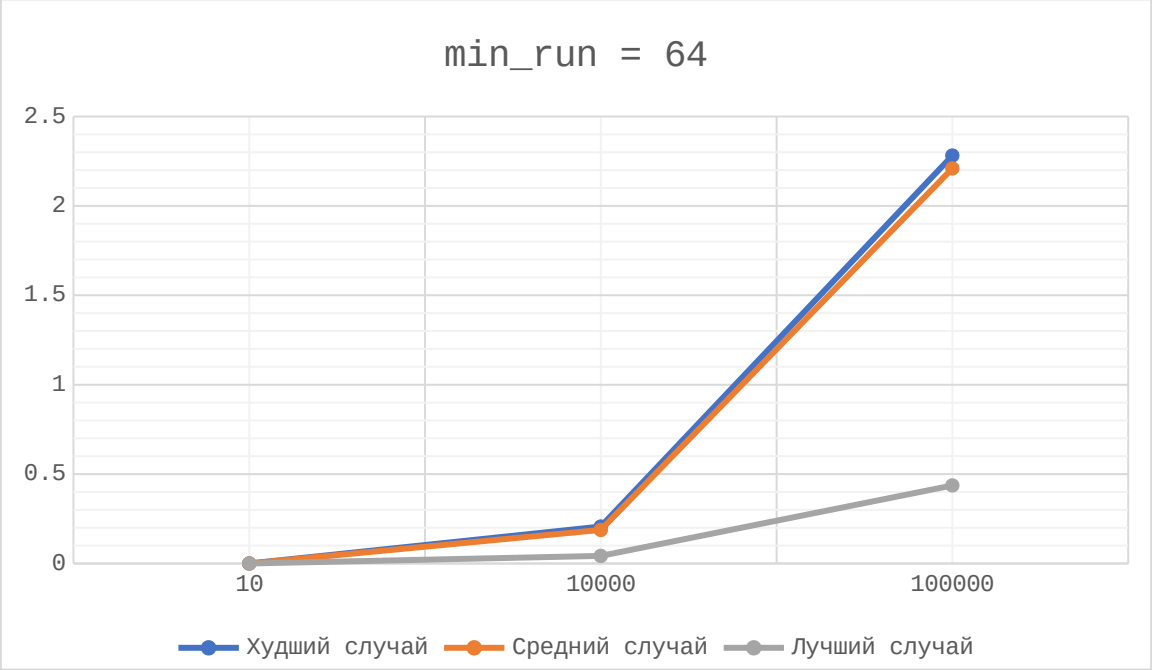

Рисунок 3 — Значение *min_run* = 64

Выводы

Был исследован и реализован алгоритм сортировки *TimSort*. Также в работе продемонстрированы замеры ее производительности, из которых видно, что сортировку лучше использовать с размером *minrun*, вычисляемым отдельно для размера сортируемого массива. Было также выяснено, что худший случай не сильно отличается от среднего, поскольку в сортировке учитывается множество факторов, которые были оптимизированы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from my_sort import timesort
if __name__ == '__main__':
n = int(input())
arr = list(map(int, input().split(' ')))
timesort(arr)
print("Answer:", *arr)
```

Название файла: test_msin.py

```
from my_sort import timesort
from my_sort import get_len_run
from my_sort import insertion_sort

def test_timsort_1():
    array = [-4, 7, 5, 3, 5, -4, 2, -1, -9, -8, -3, 0, 9, -7, -4, -10,
-4, 2, 6, 1, -2, -3, -1, -8, 0, -8, -7, -3, 5, -1, -8, -8, 8, -1, -3,
3, 6, 1, -8, -1, 3, -9, 9, -6]
    timesort(array)
    answer_arr = [0, 0, 1, 1, -1, -1, -1, -1, -1, 2, 2, -2, 3, 3, 3, -
3, -3, -3, -3, -4, -4, -4, -4, 5, 5, 5, 6, 6, -6, 7, -7, -7, 8, -8, -
8, -8, -8, -8, -8, 9, 9, -9, -9, -10]
    assert array == answer_arr



def test_timsort_2():
    array = [1, -1, 1, 1, -1, 1]
    timesort(array)
    answer_arr = [1, 1, 1, 1, -1, -1]
    assert array == answer_arr

def test_len_run():
    array = [5, 6, 8, 9, 7]
    my_len = get_len_run(array, 0)
```

```
    len_answer = 4
    assert my_len == len_answer

def test_insert_sort():
    array = [3, -7, 6, 5, 4, 3]
    insertion_sort(array, 0, 6)
    answer_array = [3, 3, 4, 5, 6, -7]
    assert array == answer_array
```

Название файла: my_sort.py

```
min_run = 32
def insertion_sort(arr, start, end):
    for i in range(start+1, end):
        current = arr[i]
        sort_ind = i-1
        while sort_ind >= start and (pow(current, 2) <
pow(arr[sort_ind], 2) or ((pow(current, 2) == pow(arr[sort_ind], 2)
and current > arr[sort_ind]) )):
            arr[sort_ind+1] = arr[sort_ind]
            sort_ind -= 1
        arr[sort_ind+1] = current
def merge(arr, start, mid, end):
    left = arr[start:mid+1]
    right = arr[mid+1:end+1]
    i = j = 0
    k = start

    while i < len(left) and j < len(right):
        if pow(left[i], 2) < pow(right[j], 2) or (pow(left[i], 2) ==
pow(right[j], 2) and left[i] > 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

def get_len_run(arr, start):
    len_run = 2
    if start == len(arr)-1:
        return 1
    is_ascending = arr[start] <= arr[start + 1]

    i = start + 1
    while i < len(arr)-1 and (arr[i] <= arr[i+1] if is_ascending else
arr[i] > arr[i+1]):
        len_run += 1
        i += 1
    if not is_ascending:
        arr[start:len_run+start] = reversed(arr[start:len_run+start])
    return len_run
def timesort(arr):
    start = 0
    count_part = 0
    while start < len(arr):
        len_run = get_len_run(arr, start)
        if len_run < min_run and len_run != 1:
            len_run = min(len(arr) - start, min_run)
        insertion_sort(arr, start, start+len_run)
        print(f'Part {count_part}:', *arr[start:(start+len_run)])
        start += len_run
        count_part += 1
    curr_size = min_run
    while curr_size < len(arr):
        for start in range(0, len(arr), 2*curr_size):
            mid = min(start + curr_size - 1, len(arr) - 1)
            end = min(start + 2*curr_size - 1, len(arr) - 1)
            merge(arr, start, mid, end)
```

```
            curr_size *= 2
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД ПРОГРАММЫ

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	arr = 6 32 7 5 4 1 -9	Part 0: 1 4 5 6 7 -9 32	Разбиение по частям и

		Answer: 1 4 5 6 7 -9 32	выведенный отсортированный массив верны.
2.	arr = -1 1	Part 0: 1 -1 Answer: 1 -1	Так как квадраты чисел равны, то элементы стоят по убыванию. Верно
3.	arr = -4 7 5 3 5 -4 2 -1 -9 -8 -3 0 9 -7 -4 -10 -4 2 6 1 -2 -3 -1 -8 0 -8 -7 -3 5 -1 -8 -8 8 -1 -3 3 6 1 -8 -1 3 -9 9 -6	Part 0: 0 0 1 -1 -1 -1 2 2 -2 3 -3 -3 -3 -4 -4 -4 -4 5 5 5 6 7 -7 -7 -8 -8 -8 -8 -8 9 -9 -10 Part 1: 1 -1 -1 3 3 -3 6 -6 8 -8 9 -9 Answer: 0 0 1 1 -1 -1 -1 -1 -1 2 2 -2 3 3 3 -3 -3 -3 -3 -4 -4 -4 -4 5 5 5 6 6 -6 7 -7 -7 8 -8 -8 -8 -8 -8 -8 9 9 -9 -9 -10	Верный ответ