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.

Студент гр. 3382 _____ Копасова К. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

Цель работы

Изучить различные виды сортировок (вставками, быстрая, сортировка слияния, сортировка подсчётом, TimSort) и реализовать сортировку TimSort, проанализировав её работу в зависимости от количества элементов в массиве (10, 1000 и 10000) и сложности сортировки (лёгкая - элементы отсортированы, средняя - элементы расположены в различном порядке и сложная - элементы расположены в порядке возрастания).

Задание

Реализация

Имеется массив данных для сортировки *int arr[]* размера *n*

Необходимо отсортировать его алгоритмом сортировки *TimSort* по убыванию модуля.

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.

Кратко алгоритм сортировки можно описать так:

1. Вычисление *min_run* по размеру массива *n* (для упрощения отладки *n* уменьшается, пока не станет меньше **16**, а не 64)
2. Разбиение массива на частично-упорядоченные (в т.ч. и по убыванию) блоки длины не меньше *min_run*
3. Сортировка вставками каждого блока
4. Слияние каждого блока с сохранением инварианта и использованием галопа (галоп начинать после **3-х** вставок подряд)

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.

Для исследования используйте стандартный алгоритм вычисления *min_run* и начинайте галоп после 7-ми вставок подряд.

Примечание:

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

Сортировка должна быть **устойчивой**.

Обратите внимание на пример.

Формат ввода

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

Формат вывода

Выводятся разделённые блоки для сортировки в формате "Part i: *отсортированный разделённый массив*"

Затем для каждого слияния выводится количество вхождений в режим галопа и получившийся массив в формате

"Gallops i: *число вхождений в галоп*

Merge i: *итоговый массив после слияния*"

Последняя строчка содержит финальный результат сортировки массива с надписью "Answer: "

Пример #1 (min_run = 10)

Ввод

20

1 -2 3 -4 5 6 -7 -8 9 -10 11 -10 -9 8 7 -7 -6 6 5 4

Вывод

Part 0: 11 -10 9 -8 -7 6 5 -4 3 -2 1

Part 1: -10 -9 8 7 -7 -6 6 5 4

Gallops 0: 0

Merge 0: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1

Answer: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1

Пример #2 (min_run = 8)

Ввод

16

-1 2 3 4 5 -6 7 8 -8 -8 7 -7 7 6 -5 4

Вывод

Part 0: 8 7 -6 5 4 3 2 -1

Part 1: -8 -8 7 -7 7 6 -5 4

Gallops 0: 1

Merge 0: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1

Answer: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1

Выполнение работы

Для начала разделим нашу задачу на два файла: в первом файле main.py будет реализована сортировка TimSort.

Во втором файле test.py будут находится все тесты и вычисления для анализа работы сортировки TimSort.

Начнем с файла main.py – опишу все функции и классы данного файла.

Main.py:

1. Функция get_minrun(n)

Данная функция вычисляет минимальную длину ранды (minrun) для данного размера массива n. Алгоритм TimSort использует минимальную длину ранды, чтобы гарантировать эффективное слияние подмассивов.

Функция начинает с r = 0 и, пока n больше или равно 16 (64 для анализа работы сортировки), проверяет младший бит числа n. Если младший бит равен 1, он присоединяется к переменной r с помощью операции побитового ИЛИ. Затем n делится на 2 с помощью побитового сдвига вправо. В конце функция возвращает сумму n и r, что дает значение minrun.

2. Функция insertion_sort(arr)

Данная функция выполняет сортировку вставками на переданном массиве arr. Сортировка вставками эффективна для небольших подмассивов, что делает ее полезной частью гибридного алгоритма TimSort.

В цикле for перебираются элементы массива, начиная со второго (i = 1). Для каждого элемента arr[i] происходит сравнение с предыдущими элементами массива. Если абсолютное значение текущего элемента меньше абсолютного значения предыдущего (abs(arr[j]) < abs(arr[j + 1])), элементы меняются местами. Этот процесс продолжается до тех пор, пока не будет

найдено место для вставки текущего элемента, обеспечивая тем самым частичную сортировку массива.

3. Функция find_first_greater(arr, start, T)

Данная функция реализует неточный бинарный поиск для нахождения первого элемента в массиве arr, начиная с индекса start, который имеет абсолютное значение меньше заданного значения T. Она используется в процессе слияния ранды, чтобы оптимизировать количество сравнений и операций слияния.

Функция инициализирует left и right как начальные и конечные индексы массива. В цикле while вычисляется середина текущего диапазона (mid). Если абсолютное значение элемента в середине меньше T, поиск продолжается в левой половине массива, иначе — в правой. В конце функция возвращает индекс первого элемента, который удовлетворяет условию.

4. Функция find_runs(arr)

Данная функция отвечает за идентификацию и формирование ранды в массиве arr. Ранда — это уже отсортированная последовательность элементов, которую TimSort будет использовать для эффективного слияния.

Сначала вычисляется minrun с помощью функции get_minrun(n), где n — размер массива. Затем начинается цикл, который проходит по всему массиву. В каждой итерации формируется новая ранда, начиная с текущего элемента. Внутренний цикл проверяет, продолжается ли текущая ранда, сравнивая элементы по абсолютному значению. Если последовательность элементов не убывает, ранда переворачивается для обеспечения убывающего порядка.

Если длина сформированной ранды меньше minrun, добавляются дополнительные элементы из массива, чтобы достичь минимальной длины. После этого выполняется сортировка вставками для оптимизации порядка

элементов внутри ранды. Каждая сформированная ранда добавляется в список runs, который возвращается в конце функции.

5. Функция merging_arr(arr1, arr2)

Данная функция выполняет слияние двух отсортированных ранды arr1 и arr2. Она объединяет элементы из двух массивов в один отсортированный массив res, учитывая условия галопа — оптимизационной техники, уменьшающей количество сравнений при длительных последовательностях уже отсортированных элементов.

Индексы i и j используются для перебора элементов в arr1 и arr2 соответственно. Счетчики count1 и count2 отслеживают количество подряд идущих элементов, взятых из arr1 и arr2. Если count1 или count2 достигает значения 3 (7 для анализа работы сортировки), функция инициирует галоп, вызывая find_first_greater для быстрого слияния длинных последовательностей элементов.

После завершения основного цикла функция добавляет оставшиеся элементы из arr1 и arr2 в результат и возвращает объединенный массив res вместе с количеством выполненных галопов gallop_count.

6. Функция merge_runs(runs)

Данная функция отвечает за слияние всех сформированных ранды. Она использует стек stack, чтобы управлять порядком слияния ран. Функция проверяет инварианты (условия) TimSort, чтобы определить, какие ранды следует сливать в первую очередь для оптимизации процесса.

Внутри функции cheking_invar проверяется, удовлетворяет ли стек текущим инвариантам TimSort. Если инварианты нарушены, вызывается merge_in_stack, которая определяет, какие ранды следует сливать. После слияния соответствующие ранды удаляются из стека, а результат добавляется обратно.

Все выполненные слияния и количество галопов фиксируются в списке merges, который возвращается вместе с окончательным отсортированным массивом stack[0].

7. Функция timsort(arr)

Это основная функция алгоритма TimSort, которая объединяет все предыдущие функции для выполнения полной сортировки массива arr.

Сначала вызывается find_runs(arr) для идентификации и формирования ранды. Затем полученные ранды передаются в merge_runs(runs) для их последующего слияния в один отсортированный массив. После завершения слияний программа выводит информацию о каждом слиянии: количество выполненных галопов и содержимое объединенных ранды.

В конце функция возвращает отсортированный массив res.

В основной части программы пользователь вводит размер массива n и затем вводит n элементов массива. После этого вызывается функция timsort(arr), которая сортирует введенный массив и возвращает отсортированный результат вместе с информацией о слияниях.

В файле test.py находятся дополнительные функции, помогающие построить графики для анализа производительности сортировки Timsort в зависимости от количества элементов в массиве и сложности сортировки. В данном файле была создана функция, которая генерирует данные в зависимости от их количества и сложности. После этого программа пропускает все значения (всего 9) через сортировку TimSort и считает время. С помощью библиотек matplotlib и pandas был построен график зависимости.

Код программ находится в Приложении А.

Тестирование

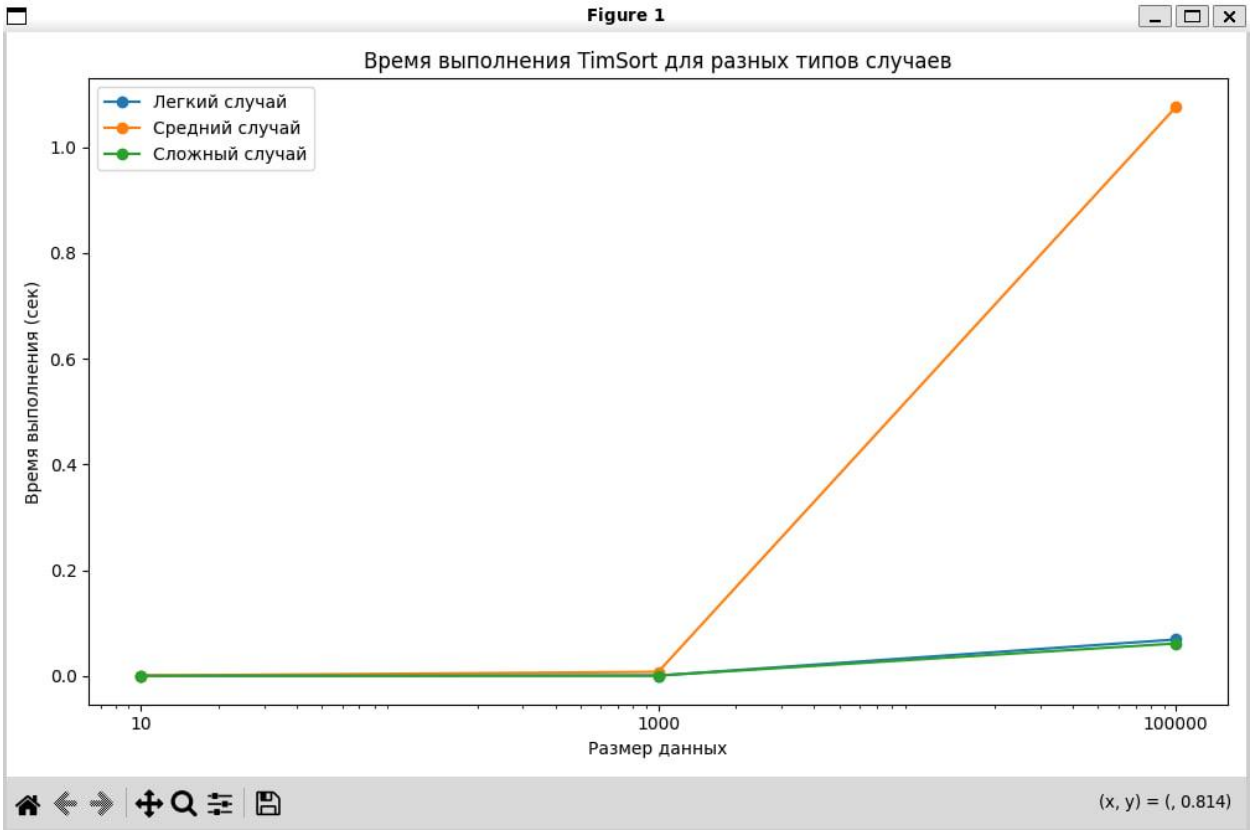
Таблица 1 — Результаты тестирования основного кода программы:

№п/п	Входные данные	Выходные данные	Коммен тарии
1	20 1 -2 3 -4 5 6 -7 -8 9 -10 11 -10 -9 8 7 -7 -6 6 5 4	Part 0: 11 -10 9 -8 -7 6 5 -4 3 -2 1 Part 1: -10 -9 8 7 -7 -6 6 5 4 Gallops 0: 0 Merge 0: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1 Answer: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1	OK
2	16 -1 2 3 4 5 -6 7 8 -8 -8 7 -7 7 6 -5 4	Part 0: 8 7 -6 5 4 3 2 -1 Part 1: -8 -8 7 -7 7 6 -5 4 Gallops 0: 1 Merge 0: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1 Answer: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1	OK
3	32 1 2 5 7 -4 -8 -9 -8 7 4 51 654 684 124 548 45 7 4 -4 8 -54 -565 -10 12 13 18 5 95 87 5 -4 -56	Part 0: -9 -8 -8 7 5 -4 2 1 Part 1: 684 654 548 124 51 45 7 4 Part 2: -565 -54 12 -10 8 7 4 -4 Part 3: 95 87 -56 18 13 5 5 -4 Gallops 0: 2 Merge 0: 684 654 548 124 51 45 -9 -8 -8 7 7 5 -4 4 2 1 Gallops 1: 2 Merge 1: -565 95 87 -56 -54 18 13 12 -10 8 7 5 5 4 -4 -4	OK

		Gallops 2: 4 Merge 2: 684 654 -565 548 124 95 87 -56 -54 51 45 18 13 12 -10 -9 -8 -8 8 7 7 7 5 5 5 -4 4 4 -4 -4 2 1 Answer: 684 654 -565 548 124 95 87 -56 -54 51 45 18 13 12 -10 -9 -8 -8 8 7 7 7 5 5 5 -4 4 4 -4 -4 2 1	

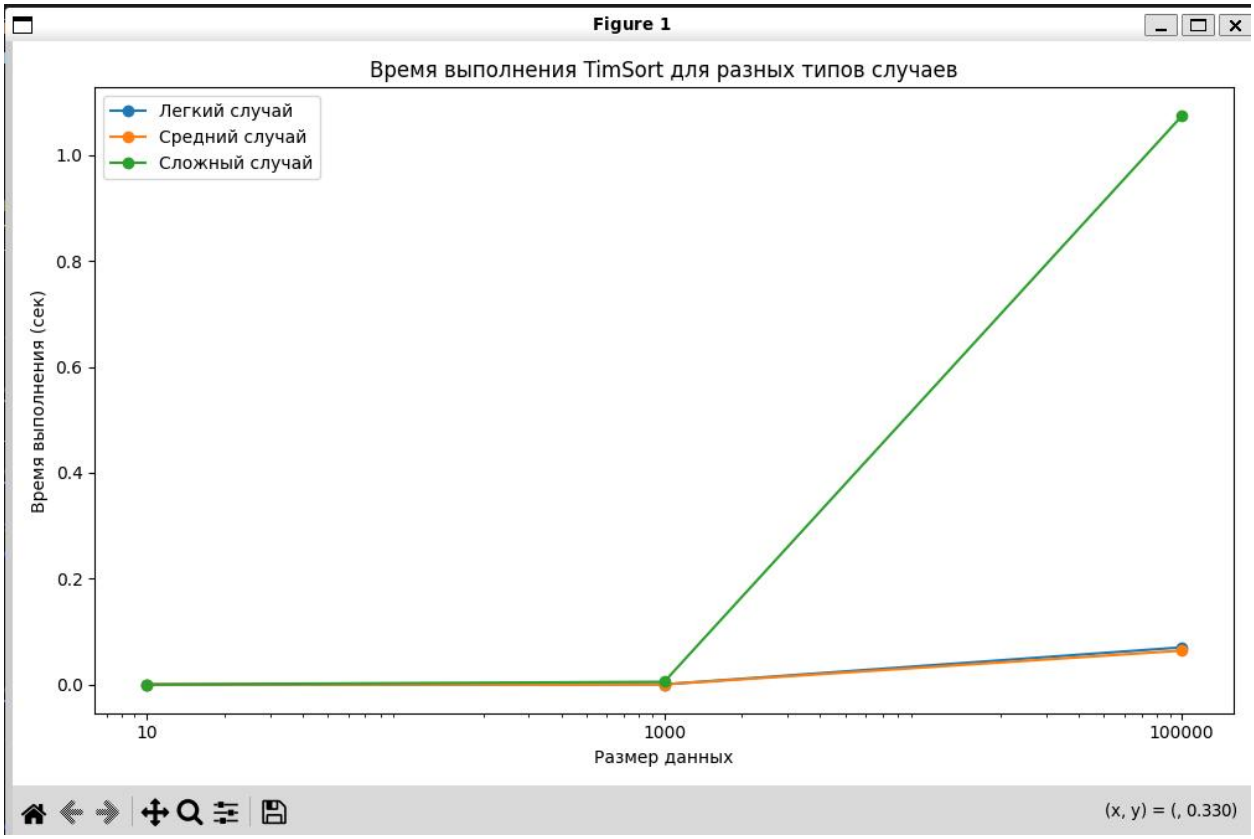
Анализ полученных данных

После анализа работы сортировки TimSort получим следующий график:



Из графика видно, что, несмотря на различное количество данных, используемых в сортировке, прослеживается одинаковая тенденция - чем больше элементов, тем больше времени на обработку элементов затрачивается, что вполне логично для сортировок. По графику так же видно, что сортировки 10 и 1000 элементов выполнялись практически одинаково, вне зависимости от генерации изначальных данных для сортировки (хотя можно отследить на 1000, что средний случай занял чуть больше времени, чем легкий и сложный). Интересно, что при большом количестве данных средний случай занял больше всего времени работы. На самом деле, это произошло из-за того, что в среднем случае мы генерировали определенное количество чисел в отрезке от 1 до 10000 и разброс у данных чисел гораздо больше, чем у последовательности «обратной» (в зависимо от порядка сортировки - по возрастанию или по убыванию) последовательности чисел.

В итоге, казалось бы, «средний» случай стал «сложным», а «сложный» наоборот - средним. Исправим данные для корректности графика:



Выводы

В ходе лабораторной работы удалось изучить различные сортировки (вставками, быстрая, сортировка слияния, сортировка подсчётом, TimSort), а так же реализовать сортировку TimSort, проанализировав её работу в зависимости от количества элементов в массиве (10, 1000 и 10000) и сложности сортировки. Была определена зависимость - чем больше данных, тем больше времени занимает сортировка, а так же чем больше размах выборки, тем больше времени занимает сортировка.

В результате лабораторной работы удалось создать код с реализацией сортировки TimSort на языке программирования Python.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.py:

```
def get_minrun(n):
        r = 0
        while n >= 16:
            r |= n & 1
            n >>= 1
        return n + r

def insertion_sort(arr):
        n = len(arr)
        for i in range(1, n):
            j = i - 1
            while (j > -1) and abs(arr[j]) < abs(arr[j + 1]):
                elem = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = elem
                j -= 1

def find_first_greater(arr, start, T):
        left, right = start, len(arr) - 1
        while left <= right:
            mid = (left + right) // 2
            if abs(arr[mid]) < abs(T):
                right = mid - 1
            else:
                left = mid + 1
        return left

def find_runs(arr):
    n = len(arr)
    minrun = get_minrun(n)
    runs = []
    i = 0
    while i < n:
        run = [arr[i]]

        if i + 1 < n:
            flag = abs(arr[i]) >= abs(arr[i+1])
        else: flag = False

        while i + 1 < n and (abs(arr[i]) >= abs(arr[i+1])) == flag:
            run.append(arr[i+1])
            i += 1

        if not flag:
            run.reverse()

        i += 1
        if len(run) < minrun:
            stop = i + minrun - len(run)
            run.extend(arr[i:stop])
            i = stop

        insertion_sort(run)
        runs.append(run)

    return runs
```

```
def merging_arr(arr1, arr2):
    i = j = 0
    count1, count2 = 0, 0
    gallop_count = 0
    res = []

    while i < len(arr1) and j < len(arr2):
        if abs(arr1[i]) >= abs(arr2[j]):
            res.append(arr1[i])
            i += 1
            count1 += 1
            count2 = 0

        else:
            res.append(arr2[j])
            j += 1
            count2 += 1
            count1 = 0

        if count1 == 3:
            gallop_count += 1
            stop = find_first_greater(arr1, i, arr2[j])
            while i < stop:
                res.append(arr1[i])
                i += 1

        if count2 == 3:
            gallop_count += 1
            stop = find_first_greater(arr2, j, arr1[i])
            while j < stop:
                res.append(arr2[j])
                j += 1

    while i < len(arr1):
        res.append(arr1[i])
        i += 1

    while j < len(arr2):
        res.append(arr2[j])
        j += 1

    return res, gallop_count

def merge_runs(runs):
    def cheking_invar(stack):
        return  len(stack[-2])  >  len(stack[-1])  and  (len(stack)  <=  2  or
len(stack[-3]) > len(stack[-1]) + len(stack[-2]))
    def merge_in_stack(stack):
        if len(stack) == 2 or len(stack[-1]) <= len(stack[-3]):
            stack[-2], gallop_count = merging_arr(stack[-2], stack[-1])
            stack.pop()
            return gallop_count, stack[-1]
        else:
            stack[-3], gallop_count = merging_arr(stack[-2], stack[-3])
            stack.pop(-2)
            return gallop_count, stack[-2]

    stack = []
    merges = []
    for run in runs:
```

```
            stack.append(run)
            while len(stack) >= 2 and not cheking_invar(stack):
                merges.append(merge_in_stack(stack))

        while len(stack) > 1:
            merges.append(merge_in_stack(stack))

        return stack[0], merges

    def timsort(arr):
        runs = find_runs(arr)
        res, gallop_and_merge = merge_runs(runs)

        for i in range(len(runs)):
            print(f'Part {i}: {" ".join(map(str,runs[i]))}')

        for i in range(len(gallop_and_merge)):
            print(f'Gallops {i}: {gallop_and_merge[i][0]}')
            print(f'Merge {i}: {" ".join(map(str,gallop_and_merge[i][1]))}')

        return res

    n = int(input())
    arr = list(map(int,input().split()))
    res = timsort(arr)
    print(f'Answer:', " ".join(map(str,res)))
```

Файл test.py:

```
from main import timsort
import time
import random
import pandas as pd
import matplotlib.pyplot as plt

def data_generation():
    easy_arrs = []
    normal_arrs = []
    hard_arrs = []

    # данные уже отсортированы
    easy_arr_10 =  list(range(10, 1, -1))
    easy_arr_1000 = list(range(1000, 1, -1))
    easy_arr_100000 = list(range(100000, 1, -1))
    easy_arrs = [easy_arr_10, easy_arr_1000, easy_arr_100000]
    # данные случайные
    normal_arr_10 = [random.randint(0, 10000) for _ in range(10)]
    normal_arr_1000 = [random.randint(0, 10000) for _ in range(1000)]
    normal_arr_100000 = [random.randint(0, 10000) for _ in range(100000)]
    normal_arrs = [normal_arr_10, normal_arr_1000, normal_arr_100000]
    # данные расположены в обратном порядке
    hard_arr_10 =  list(range(0, 10))
    hard_arr_1000 =  list(range(0, 1000))
    hard_arr_100000 =  list(range(0, 100000))
    hard_arrs = [hard_arr_10, hard_arr_1000, hard_arr_100000]
    print(easy_arr_10)
    easy_time = []
    normal_time = []
    hard_time = []
    # print(easy_arrs)
    #проверка "простого" случая
```

```
    for value in easy_arrs:
        start_time = time.time()
        data = timsort(value)
        end_time = time.time()
        easy_time.append(end_time - start_time)

    #проверка "среднего" случая
    for value in normal_arrs:
        start_time = time.time()
        data = timsort(value)
        end_time = time.time()
        normal_time.append(end_time - start_time)

    #проверка "сложного" случая
    for value in hard_arrs:
        start_time = time.time()
        data = timsort(value)
        end_time = time.time()
        hard_time.append(end_time - start_time)

    print(easy_time, normal_time, hard_time)


    # Вывод времени выполнения (опционально)
    print("Время выполнения (сек):")
    print("Легкий случай:", easy_time)
    print("Средний случай:", hard_time)
    print("Сложный случай:", normal_time)

    # Определение размеров данных
    sizes = [10, 1000, 100000]

    # Создание DataFrame для удобства обработки
    data = {
        'size': sizes * 3,
        'case': ['Легкий случай'] * 3 + ['Средний случай'] * 3 + ['Сложный
случай'] * 3,
        'time': easy_time + hard_time + normal_time
    }

    df = pd.DataFrame(data)

    # Построение графика времени выполнения
    plt.figure(figsize=(10, 6))
    for case_label in ['Легкий случай', 'Средний случай', 'Сложный случай']:
        subset = df[df['case'] == case_label]
        plt.plot(subset['size'], subset['time'], marker='o', label=case_label)

    plt.xlabel('Размер данных')
    plt.ylabel('Время выполнения (сек)')
    plt.title('Время выполнения TimSort для разных типов случаев')
    plt.xscale('log')  # Логарифмическая шкала для оси X
    plt.xticks(sizes, sizes)  # Установка меток по оси X
    plt.legend()
    plt.tight_layout()
    plt.savefig('time_comparison.png')  # Сохранение графика в файл
    plt.show()

data_generation()
```