

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

**КУРСОВАЯ РАБОТА**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: Сравнительное исследование сортировки слабой кучей с**  
**быстрой сортировкой**

Студентка гр. 9304

\_\_\_\_\_

Рослова Л.С.

Преподаватель

\_\_\_\_\_

Филатов А.Ю.

Санкт-Петербург

2020

## **ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ**

Студентка Рослова Л.С.

Группа 9304

Тема работы: Сравнительное исследование сортировки слабой кучей с быстрой сортировкой

Исходные данные: программе на вход подаётся набор чисел

Содержание пояснительной записки:

- Аннотация
- Содержание
- Введение
- Постановка задачи
- Ход выполнения работы
- Тестирование
- Исследование
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 30 страниц.

Дата выдачи задания: 23.11.2020

Дата сдачи реферата: 27.12.2020

Дата защиты реферата: 28.12.2020

Студентка

\_\_\_\_\_

Рослова Л.С.

Преподаватель

\_\_\_\_\_

Филатов А.Ю.

## **АННОТАЦИЯ**

В данной курсовой работе производится реализация алгоритма сортировки слабой кучей. Помимо написания самого алгоритма происходит его исследование и сравнение с быстрой сортировкой. Данный процесс производится с помощью тестов, основанных на случайной генерации чисел. Результатами исследования являются числовые паросочетания, на базе которых формируются графики для оценки практических выводов с теоретическими оценками.

## **SUMMARY**

In this course work, a weak heap sorting algorithm is implemented. In addition to writing the algorithm itself, it is studied and compared with quick sort. This process is performed using tests based on randomly generated numbers. The research results are numerical matchings, on the basis of which graphs are formed to evaluate practical conclusions with theoretical estimates.

## СОДЕРЖАНИЕ

Введение	5
1. Постановка задачи	6
2. Ход выполнения работы	7
2.1. Определение слабой кучи	7
2.2. Описание алгоритма сортировки слабой кучей	8
2.3. Алгоритм быстрой сортировки	9
2.4. Описание структур данных и функций программы	10
3. Тестирование	12
4. Исследование	15
4.1. Исследование времени работы алгоритма сортировки слабой кучей	15
4.2. Исследование времени работы быстрой сортировки	15
4.3. Исследование затрат памяти для сортировки слабой кучей и быстрой сортировкой	16
4.4. План исследования	16
4.5. Результаты исследования	17
Заключение	20
Список использованных источников	21
Приложение А. Исходный код WeakHeap	22
Приложение Б. Исходный код customQsort	26
Приложение В. Исходный код main	29
Приложение Г. Исходный код script	31
Приложение Д. Исходный код Makefile	32
Приложение Е. Исходный код Plot.py	33

## ВВЕДЕНИЕ

Слабые кучи были введены Даттоном (1993) как часть варианта алгоритма сортировки кучи, который (в отличие от стандартной сортировки кучи с использованием двоичных куч) может использоваться для сортировки  $n$  элементов, используя только  $n \log_2 n + O(n)$  сравнений. Позже они были исследованы как более широко применимая структура данных очереди приоритетов.

Целью работы является изучение такой структуры данных как «слабая куча» и реализация на её основе сортировки. Также необходимо произвести сравнительное исследование сортировки слабой кучей с быстрой сортировкой и сопоставить экспериментальные значения с теоритическими.

## **1. ПОСТАНОВКА ЗАДАЧИ**

Реализация и экспериментальное машинное исследование алгоритма сортировки слабой кучей в сравнении с быстрой сортировкой.

В исследование входят следующие подзадачи:

- Генерацию представительного множества реализаций входных данных.
- Выполнение исследуемых алгоритмов на сгенерированных наборах данных.
- При этом в ходе вычислительного процесса фиксируется как характеристики работы программы, так и количество произведенных базовых операций алгоритма.
- Фиксацию результатов испытаний алгоритма, накопление статистики.
- Представление результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

## 2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

### 2.1. Определение слабой кучи

Обычная куча представляет собой сортирующее дерево, в котором любой родитель больше (или равен) чем любой из его потомков. В слабой куче это требование ослаблено — любой родитель больше (или равен) любого потомка только из своего правого поддеревя. В левом поддереве потомки могут быть и меньше и больше родителя.

В корне кучи находится максимальный по величине элемент и он не имеет левого поддеревя.

«Левый» и «правый» в слабой куче — явление ситуативное. Поддерево может быть для своего родительского узла как левым так и правым потомком — причём, это отношение «левый/правый» для поддеревя и родителя может по ходу процесса сортировки многократно переключаться с одного значения на противоположное.

Т.к в слабой куче корень имеет только правое поддерево, то формулы для индексов потомков получают обратный сдвиг на 1 позицию:

- левый потомок:  $2 \times i$
- правый потомок:  $2 \times i + 1$

Указать для родителя кто у него правый потомок, а кто левый, помогает дополнительный битовый массив BIT (состоящий только из значений 0/1) для тех узлов, у которых есть потомки. В нём для  $i$ -го элемента отмечено, был ли обмен местами между его левым и правым поддеревьями. Если значение для элемента равно 0, то значит обмена не было. Если значение равно 1, значит, левый и правый потомок идут в обратном порядке. А формулы при этом вот такие:

- левый потомок:  $2 \times i + \text{BIT}[i]$
- правый потомок:  $2 \times i + 1 - \text{BIT}[i]$

Пример слабой кучи и массива битов представлен на рисунке 1.

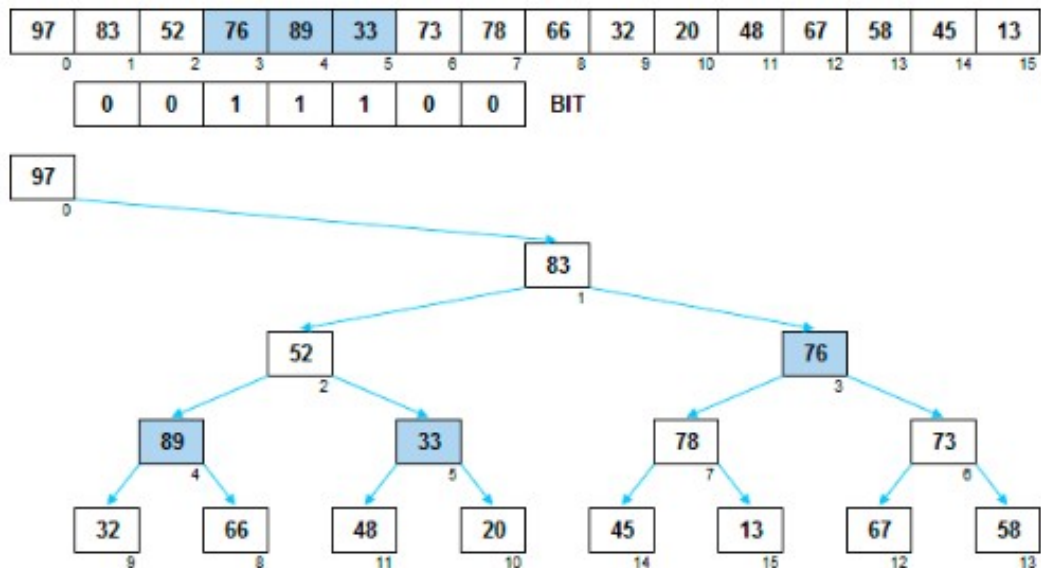


Рисунок 1 — Пример слабой кучи и битового массива на её основе.

Чтобы для любого родителя поменять местами левое и правое поддерево, в самом массиве группы элементов никуда передвигать не нужно.

Переключается только значение 0/1 для родителя в массиве BIT и всё.

## 2.2. Описание алгоритма сортировки слабой кучей

Обмен местами левого и правого потомков — основной инструмент для преобразования в слабую кучу набора данных из массива и последующей его сортировки.

Алгоритм сортировки выглядит следующим образом:

### I. Формируем из массива слабую кучу:

**I.1.** Перебираем элементы массива слева-направо.

**I.2.** Для текущего элемента поднимаемся вверх по родительской ветке до ближайшего «правого» родителя.

**I.3.** Сравниваем текущий элемент и ближайшего правого родителя.

**I.4.** Если ближайший правый родитель меньше текущего элемента, то:

**I.4.a.** Меняем местами (левый  $\Leftrightarrow$  правый) поддеревья с потомками для узла, в котором находится текущий элемент.



**I.4.б.** Меняем значениями ближайший «правый» родитель и узел с текущим элементом.

**II.** Из корня кучи текущий максимальный элемент перемещаем в конец неотсортированной части массива, после чего восстанавливаем слабую кучу:

**II.1.** В корне кучи находится текущий максимальный элемент для неотсортированной части массива.

**II.2.** Меняем местами максимум из корня кучи и последний элемент в неотсортированной части массива. Последний элемент с максимумом перестаёт быть узлом слабой кучи.

**II.3.** После этого обмена дерево перестало быть слабой кучей, так как в корне оказался не максимальный элемент. Поэтому делаем просейку:

**II.3.а.** Опускаемся из корня кучи по левым потомкам как можно ниже.

**II.3.б.** Поднимаемся по левым потомкам обратно к корню кучи, сравнивая каждый левый потомок с корнем.

**II.3.в.** Если элемент в корне меньше, чем очередной левый потомок, то:

**II.3.в.1.** Меняем местами (левый  $\Leftrightarrow$  правый) поддеревья с потомками для узла, в котором находится текущий левый потомок.

**II.3.в.2.** Меняем значениями корень кучи и узел с текущим левым потомком.

**II.4.** В корне слабой кучи снова находится максимальный элемент для оставшейся неотсортированной части массива. Возвращаемся в пункт II.1 и повторяем процесс, пока не будут отсортированы все элементы.

## **2.3. Алгоритм быстрой сортировки**

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе

своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Общая идея алгоритма состоит в следующем:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньше опорного», «равные» и «большие».
3. Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

## **2.4. Описание структур данных и функций программы**

1. Класс WeakHeap - класс представления слабой кучи.

Поля класса:

- `int binSize` — количество узлов кучи, имеющих двух потомков;
- `std::vector<int> arr` — вектор значений, который подаём алгоритму для сортировки;
- `std::vector<bool> BIN` — вектор, в котором для *i*-го элемента отмечено, был ли обмен местами между его левым и правым поддеревьями.

Методы класса:

- `bool isWeakHeap()` - проверка того, является ли массив чисел слабой кучей;
- `void createWeakHeap()` - создание слабой кучи на базе массива, если входные данные не являются этой структурой данных изначально;
- `void sortArr()` - сортировка массива алгоритмом слабой кучи.

Исходный код в приложении А.

## 2. Функции:

- `void customQsort(std::vector<int> arrDigit)` — быстрая сортировка массива чисел;
- `void generator(const size_t count, std::vector<int>& arr)` — функция для генерации случайных чисел.

Исходный код в приложении Б.

## 3. Функция `main`:

Функция `main` имеет 2 возможных варианта работы:

- Если на вход подаётся один аргумент, то программа генерирует такое количество тестов, которое ввёл пользователь. Сгенерированные данные подаются на вход сортировке слабой кучей.
- Если в `main` подаётся несколько аргументов, то из этих значений формируется один тест, который проходит сортировку реализованным алгоритмом.

Все тесты также проходят и быструю сортировку, это необходимо в дальнейшем для сравнения двух алгоритмов между собой по времени выполнения.

Исходный код в приложении В.

### 3. ТЕСТИРОВАНИЕ

Входные данные: на вход программе подаётся текстовый файл, котором через пробел записаны численные значения. Либо пользователь вводит значения самостоятельно через консоль.

Выходные данные: программа выводит элементы в отсортированном порядке в консоль.

Тестирование производится при помощи скрипта script1, написанного на Bash. Исходный код представлен в приложении Г.

Для сборки программы был также написан Makefile. Исходный код представлен в приложении Д.

Пример запуска тестирующего скрипта представлен на рисунке 2.

Результаты тестирования представлены в таблице 1.

№	Входные данные	Выходные данные	Комментарии
1	59 52 40 29 87 16 30 86 45 49 50 22 92 17 60 91	16 17 22 29 30 40 45 49 50 52 59 60 86 87 91 92	Случайные положительные числа
2	92 87 50 91 59 22 30 60 45 49 40 16 52 17 29 86	16 17 22 29 30 40 45 49 50 52 59 60 86 87 91 92	Случайные положительные числа
3	97 83 52 76 89 33 78 73 32 66 48 20 45 13 67 58	13 20 32 33 45 48 52 58 66 67 73 76 78 83 89 97	Случайные положительные числа
4	10 9 8 7 6 5 4 3 2 1 0	0 1 2 3 4 5 6 7 8 9 10	Значения от большого к меньшему
5	0 1 2 3 4 5 6 7 8 9 10	0 1 2 3 4 5 6 7 8 9 10	Значения от меньшего к большому
6	-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10	-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10	Числа от меньшего к большему с отрицательными значениями

Продолжение таблицы 1

7	-1 -2 -3 -4 -5 -6 -7 -8 -9 -10	-10 -9 -8 -7 -6 -5 -4 -3 -2 -1	Отрицательные значения
8	23 54 7 92 16 37 59 152 43 65 10	7 10 16 23 37 43 54 59 65 92 152	Случайные положительные числа
9	-23 56 -34 -78 45 23 -21 76 38	-78 -34 -23 -21 23 38 45 56 76	Случайные числа разных знаков

```

larzi@Larzi:~/Рабочий стол/CW$ ./script1

Test 1:
Test string = 59 52 40 29 87 16 30 86 45 49 50 22 92 17 60 91
16 17 22 29 30 40 45 49 50 52 59 60 86 87 91 92

Test 2:
Test string = 92 87 50 91 59 22 30 60 45 49 40 16 52 17 29 86
16 17 22 29 30 40 45 49 50 52 59 60 86 87 91 92

Test 3:
Test string = 97 83 52 76 89 33 78 73 32 66 48 20 45 13 67 58
13 20 32 33 45 48 52 58 66 67 73 76 78 83 89 97

Test 4:
Test string = 10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10

Test 5:
Test string = 0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10

Test 6:
Test string = -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10

Test 7:
Test string = -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1

Test 8:
Test string = 23 54 7 92 16 37 59 152 43 65 10
7 10 16 23 37 43 54 59 65 92 152

Test 9:
Test string = -23 56 -34 -78 45 23 -21 76 38
-78 -34 -23 -21 23 38 45 56 76

```

Рисунок 2 — Пример запуска программы через script1.

Пример запуска программы через исполняемый файл cw представлен на рисунке 3.

```
larzi@Larzi:~/Рабочий стол/CW$ ./cw  
-345 867 5785 248 3 -4 86 -6 -34 -65 1 0  
-345 -65 -34 -6 -4 0 1 3 86 248 867 5785
```

Рисунок 3 — Пример запуска программы через исполняемый файл cw.

## 4. ИССЛЕДОВАНИЕ

### 4.1. Исследование времени работы алгоритма сортировки слабой кучей

Сложность по времени зависит от просейки. Однократная просейка обходится в  $O(\log n)$ . Сначала мы для  $n$  элементов делаем просейку, чтобы из массива построить первоначальную кучу — этот этап занимает  $O(n \log n)$ . На втором этапе мы при вынесении  $n$  текущих максимумов из кучи делаем однократную просейку для оставшейся неотсортированной части, т.е. этот этап также стоит нам  $O(n \log n)$ .

Итоговая сложность по времени:  $O(n \log n) + O(n \log n) = O(n \log n)$ .

При этом у сортировки слабой кучей нет ни вырожденных ни лучших случаев.

Сортировка кучей в среднем работает несколько медленнее чем быстрая сортировка. Но для quicksort можно подобрать такой массив, на котором компьютер зависнет, а вот для Weakheapsort — нет.

Сравнение по времени сортировки слабой кучей с быстрой сортировкой представлено в таблице 2.

Таблица 2 - сравнение сортировки слабой кучей с быстрой сортировкой по времени

	Сложность по времени		
	Худшая	Средняя	Лучшая
Сортировка	$O(n \log n)$		
Быстрая сортировка	$O(n^2)$	$O(n \log n)$	$O(n)$

### 4.2. Исследование времени работы быстрой сортировки

Общая сложность алгоритма определяется лишь количеством разделений, то есть глубиной рекурсии. Глубина рекурсии, в свою очередь,

зависит от сочетания входных данных и способа определения опорного элемента.

- Лучший случай:

В наиболее сбалансированном варианте при каждой операции разделения массив делится на две одинаковые (плюс-минус один элемент) части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит  $\log_2 n$ . В результате количество сравнений, совершаемых быстрой сортировкой, было бы равно значению рекурсивного выражения  $C_n = 2C_{n/2} + n$ , что даёт общую сложность алгоритма  $O(n \log_2 n)$ .

- Средний случай:

Среднюю сложность при случайном распределении входных данных можно оценить лишь вероятностно. Поскольку на каждом уровне рекурсии выполняется не более  $O(n)$  операций, средняя сложность составит  $O(n \log n)$ .

- Худший случай:

В самом несбалансированном варианте каждое разделение даёт два подмассива размерами 1 и  $n-1$ , то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. При простейшем выборе опорного элемента — первого или последнего в массиве, — такой эффект даст уже отсортированный (в прямом или обратном порядке) массив, для среднего или любого другого фиксированного элемента «массив худшего случая» также может быть специально подобран. В этом случае потребуется  $n-1$  операций разделения, а общее время работы составит  $O(n^2)$  операций, то есть сортировка будет выполняться за квадратичное время.



### **4.3. Исследование затрат памяти для сортировки слабой кучей и быстрой сортировкой**

- Для слабой кучи

Сложность по дополнительной памяти  $O(n)$  — требуется дополнительный массив, в котором для узлов с потомками (таковых в массиве примерно половина) зафиксирован порядок левого/правого поддеревьев.

Может становиться и  $O(1)$ . Для элемента нам нужен всего один дополнительный бит (0/1), чтобы указать порядок следования потомков.

Другой способ превратить  $O(n)$  в  $O(1)$  — хранить флаги в целом числе. Двоичное разложение числа — набор нулей и единиц, отвечающих за порядок поддеревьев всех элементов массива.  $i$ -й элемент массива соответствует  $i$ -му биту числа.

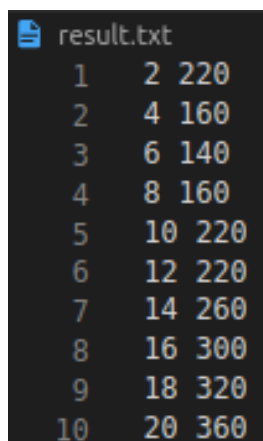
- Для быстрой сортировки

Требует лишь  $O(\log n)$  дополнительной памяти для своей работы. Не улучшенный рекурсивный алгоритм в худшем случае нуждается в  $O(n)$  памяти, что может привести к переполнению стека.

### **4.4. План исследования**

Для подтверждения теоретической оценки при запуске программы генерируется такое количество тестов, которое пользователь введёт вручную. Эти тесты заполнены случайными значениями, которые подаются программе на вход. Каждая последовательность проходит сортировку слабой кучей и быстрой сортировкой. Во время исследования фиксируется время, затраченное на выполнение алгоритма. Результаты записываются в файлы result.txt и result2.txt для сортировки слабой кучей и быстрой сортировки соответственно. Вывод записывается в виде паросочетаний «количество элементов — время сортировки».

Пример вывода подсчётов для сортировки слабой кучей для 10 тестов (в каждом тесте количество элементов увеличивается на 2) в файл представлен на рисунке 4.



1	2	220
2	4	160
3	6	140
4	8	160
5	10	220
6	12	220
7	14	260
8	16	300
9	18	320
10	20	360

Рисунок 4 — Пример вывода исследования времени для сортировки слабой кучей для 10 тестов

#### 4.5. Результаты исследования

На основе числовых метрик, полученных при запуске программы и записанных в файлы result.txt и result2.txt были построены графики. Ниже приведены 4 графика, иллюстрирующих время, затраченное на сортировки слабой кучей и быстрой сортировки в среднем и лучшем случаях.

На всех графиках оранжевой линией построен график логарифма от количества элементов в массиве. Этот график помогает наглядно показать схожесть формы графиков. Синей кривой построен график, отображающий время в микросекундах, затраченное на выполнение того или иного алгоритма. Наглядные результаты исследования продемонстрированы с помощью графиков, построенных программой plot.py, написанной на языке программирования Python. Исходный код представлен в приложении Е. На рисунках 5 – 8 представлены графики необходимого времени для сортировки слабой кучей и быстрой сортировки в среднем и лучшем случае.

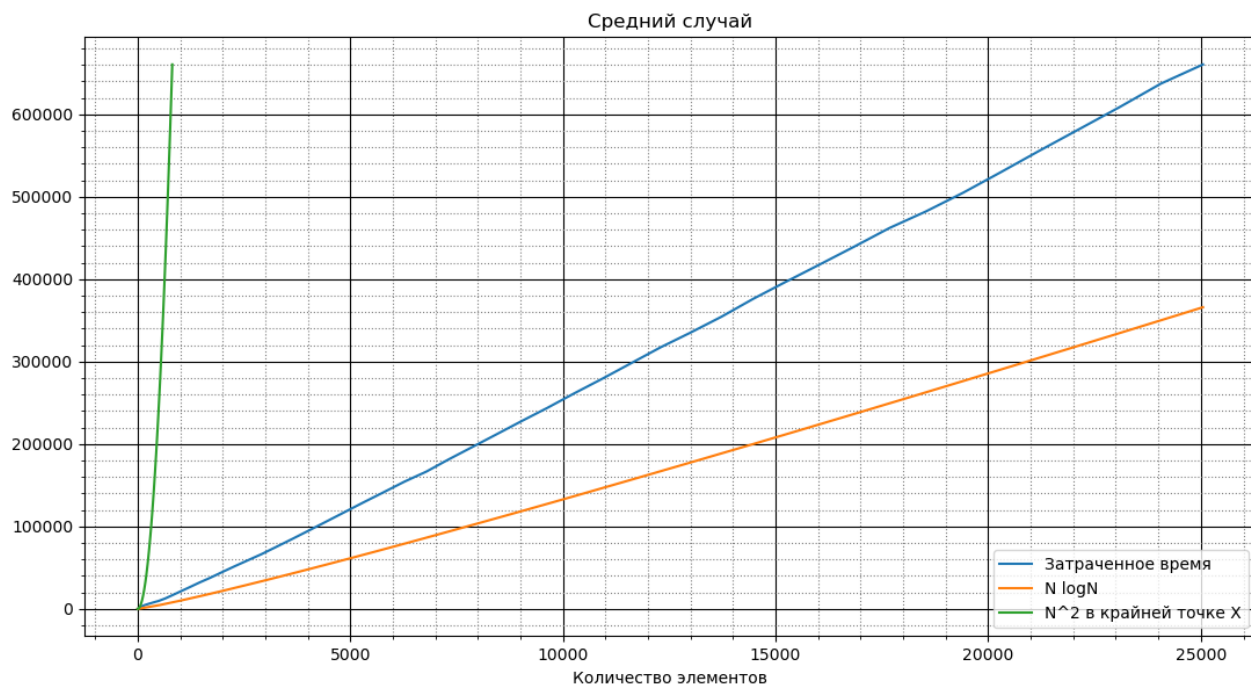


Рисунок 5 — Средний случай для сортировки слабой кучей

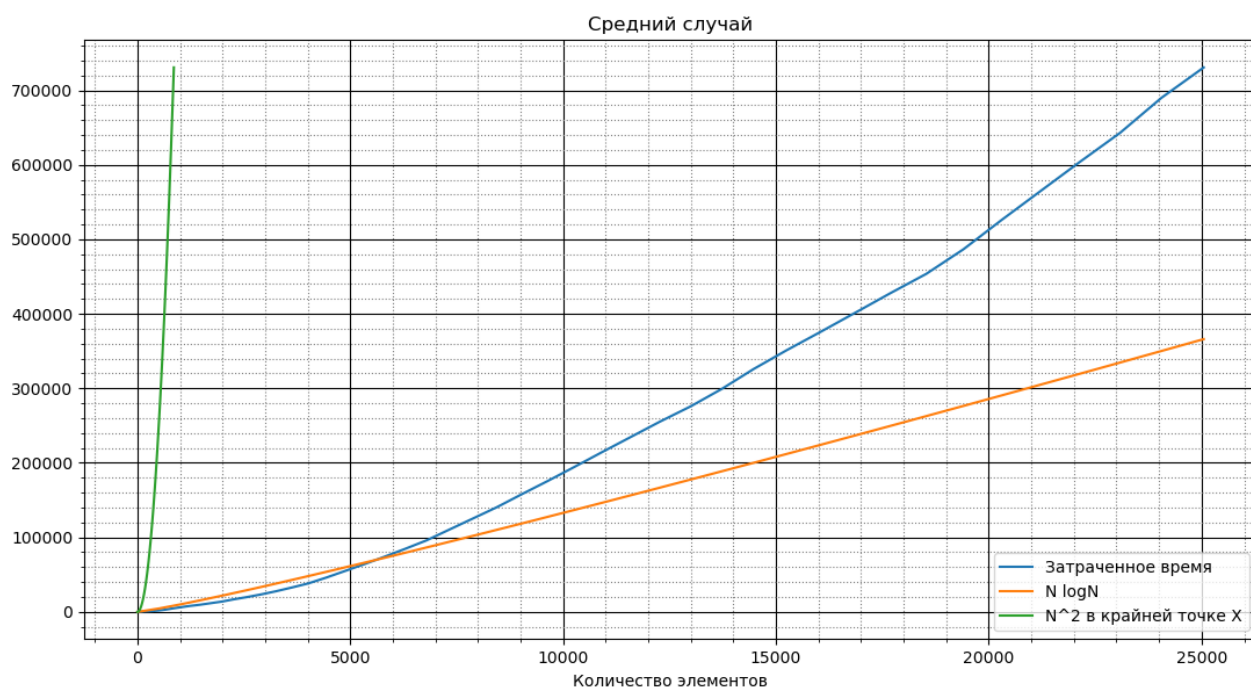


Рисунок 6 — Средний случай для быстрой сортировки

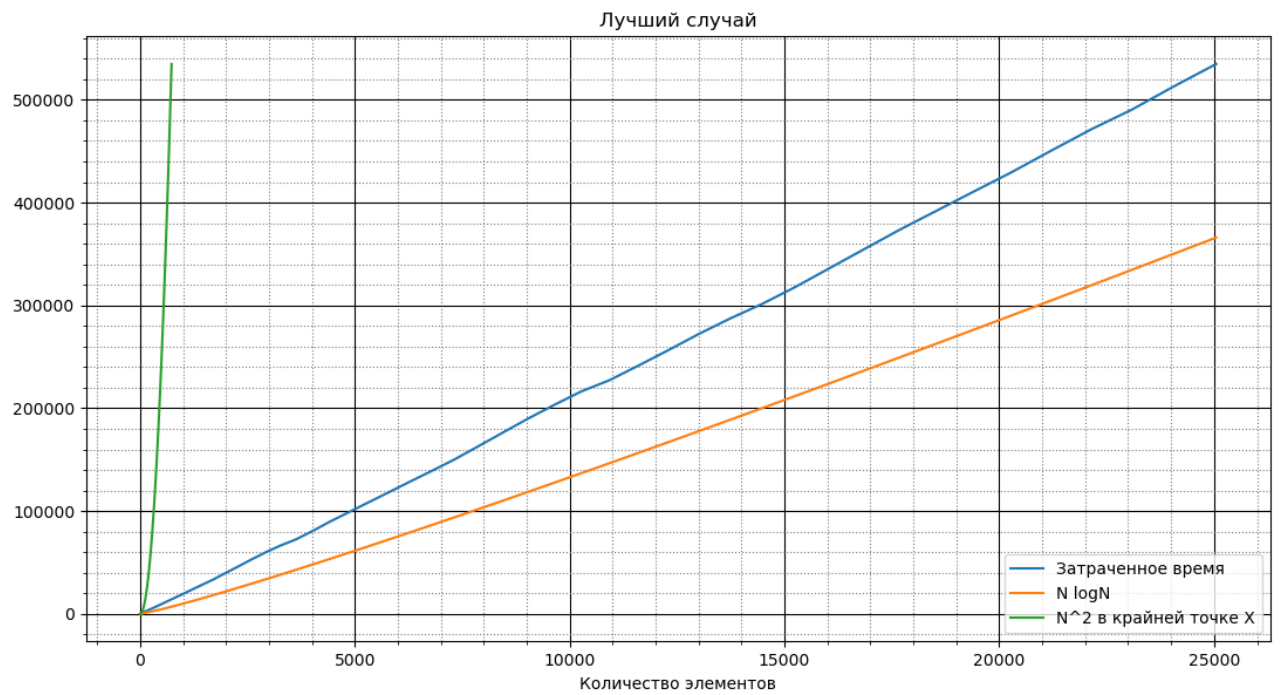


Рисунок 7 — Лучший случай для сортировки слабой кучей

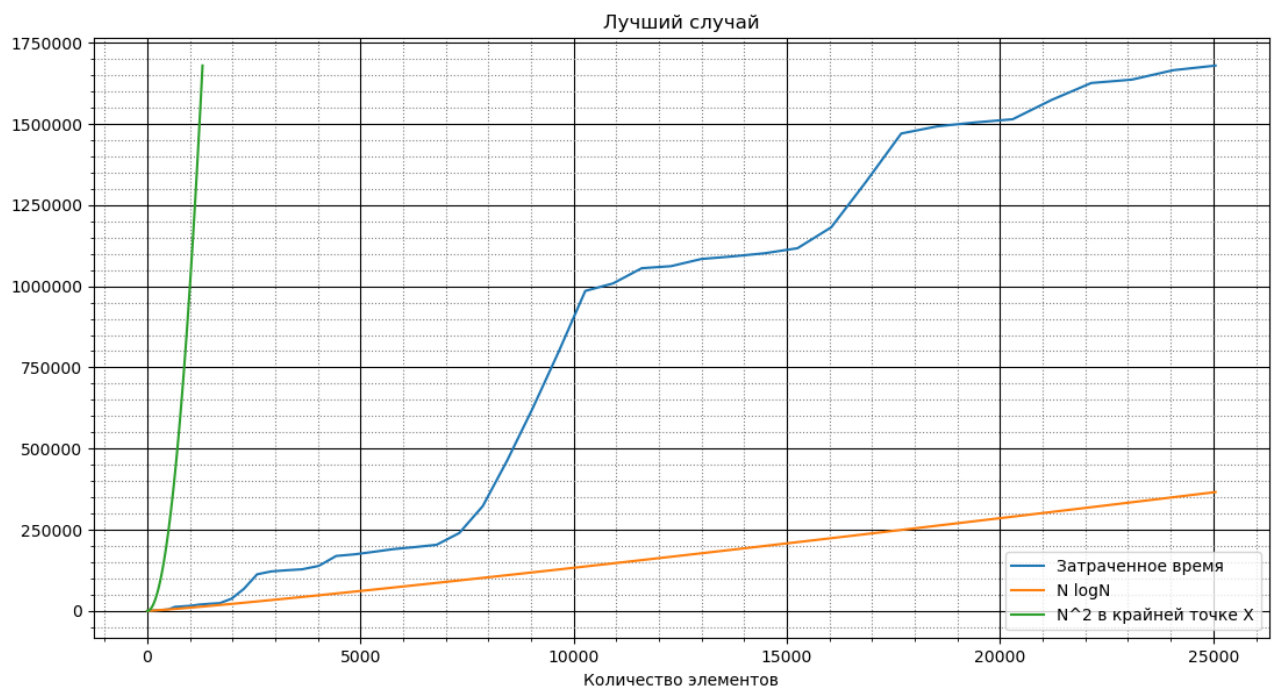


Рисунок 8 — Лучший случай для быстрой сортировки

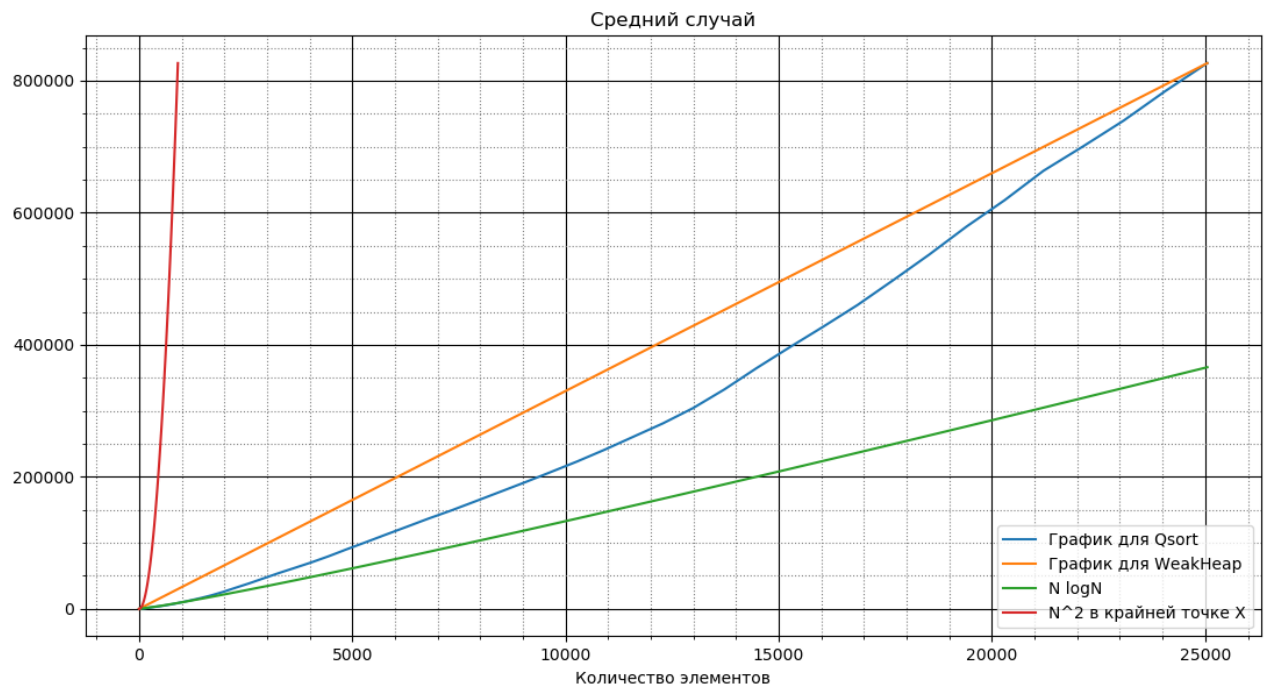


Рисунок 9 — Средний случай на одном графике

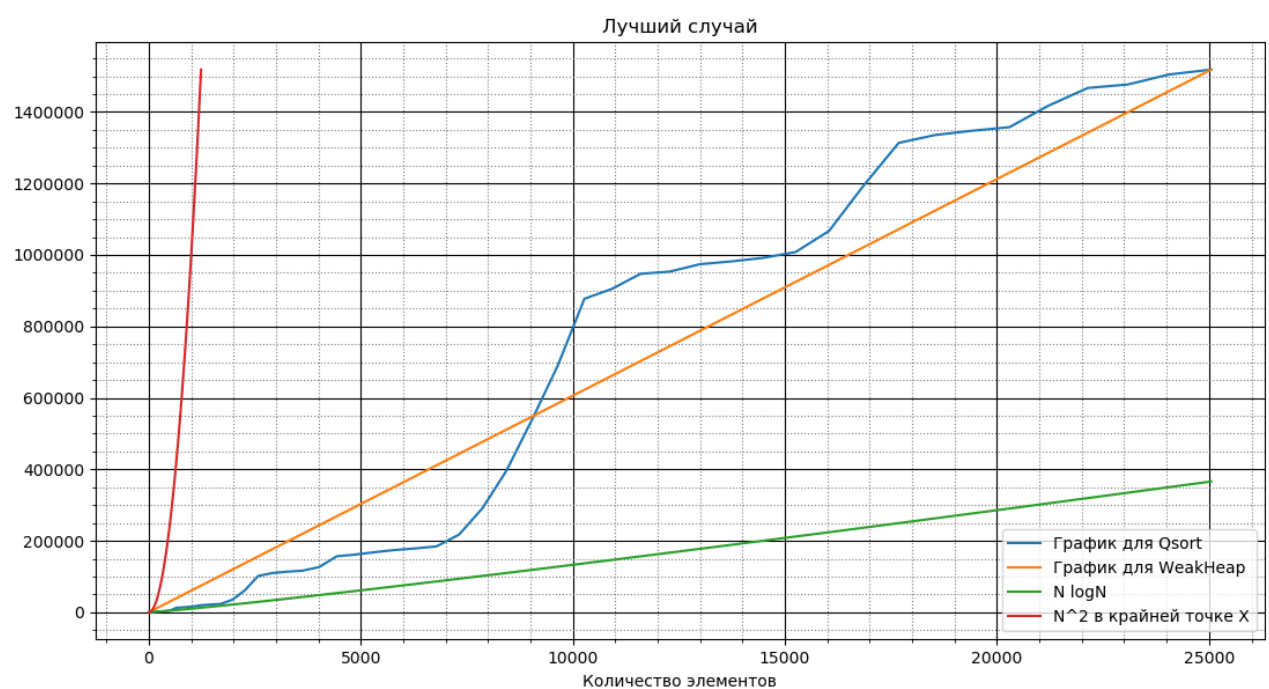


Рисунок 10 — Лучший случай на одном графике

Опираясь на графики можно сделать следующий вывод:  
 сортировка слабой кучей работает несколько медленнее чем быстрая сортировка. Однако последняя имеет тенденцию стремиться к  $O(n^2)$ , тогда как сортировка слабой кучей будет варьироваться в пределах  $O(n \log n)$ .

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсовой работы была реализована структура Слабой кучи, а также сортировка на её основе. На базе числовых метрик, полученных в результате исследования работы алгоритмов, были построены графики зависимости времени выполнения алгоритма от количества элементов в массиве.

Также был написан Makefile, собирающий программу в исполняемый файл sw, и скрипт для тестирования. Была реализована программа на языке Python для построения графиков по измеренным в ходе работы программы sw значениям.

Полученные практические результаты сравнили с теоритическими оценками, в результате чего выяснили, что сортировка слабой кучей справляется несколько медленне быстрой сортировки. Однако последняя может сильно деградировать по скорости при неудачно подобранном массиве, в то время как сортировка слабой кучей справиться с такой задачей.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. <https://habr.com/ru/company/edison/blog/499786/>
2. <https://habr.com/ru/company/edison/blog/495420/>
3. [https://ru.qaz.wiki/wiki/Weak\\_heap#Weak-heap\\_sort](https://ru.qaz.wiki/wiki/Weak_heap#Weak-heap_sort)
4. [https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F\\_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)
5. <https://habr.com/ru/sandbox/29775/>
6. <https://itnan.ru/post.php?c=1&p=499786>
7. <https://compsciclub.ru/courses/advanced-algo/2014-autumn/classes/875/>

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД WEAKHEAP

#### Файл WeakHeap.h

```
#pragma once

#include <vector>
#include <memory>

class WeakHeap{
    bool isWeakHeap();
    void createWeakHeap();
    std::vector<bool> BIN;
    int binSize;
public:
    WeakHeap();
    void sortArr();
    std::vector<int> arr;
};
```

#### Файл WeakHeap.cpp

```
#include <iostream>
#include <ctime>
#include <fstream>

#include <chrono>

#include "WeakHeap.h"

WeakHeap::WeakHeap(){

}

bool WeakHeap::isWeakHeap(){

    size_t position = 0;
    bool flag = 0;
```



```

        for(size_t i = 0; i < arr.size(); i++){           //проверка на свойство
салобой кучи:
        position = i * 2 + 1;                             //правый потомок меньше
родителя
        if(arr[i] < arr[position] && position < arr.size()){
            flag = 1;
            break;
        }
    }

    if(arr.size() % 2){                                   //создаётся битовый массив
для
        binSize = (arr.size() / 2) + 1;                 //элементов с потомками
    }else{
        binSize = (arr.size() / 2);
    }

    BIN.resize(arr.size());

    if(flag){
        return false;
    }else{
        return true;
    }
}

```

```

void WeakHeap::createWeakHeap(){

    int digit = 0;

    for(int i = arr.size() - 1; i >= 0; i--){

        digit = i;

        while(digit){
            if((digit % 2){
                digit = digit / 2;
                break;
            }else{
                digit = digit / 2;
            }
        }
    }
}

```

```

        if(arr[i] > arr[digit]){
            if(i < binSize){
                BIN[i] = !BIN[i];
            }
            std::swap(arr[i], arr[digit]);
        }
    }
}

```

```

void WeakHeap::sortArr(){

    if(arr.size() < 2){
        exit(1);
    }

    clock_t t = clock();

    if(!isWeakHeap()){
        createWeakHeap();
    }

    size_t countNotSortArr = arr.size() - 1;
    size_t position = 1;

    std::vector<int> forLambdaArr = arr;
    std::vector<bool> forLambdaBIN = BIN;

    while(countNotSortArr){
        std::swap(forLambdaArr[0], forLambdaArr[countNotSortArr]);
        countNotSortArr--;

        auto PR = [&forLambdaArr, &forLambdaBIN, &countNotSortArr](size_t
position, auto &&PR){ //Просейка массива после
                                                                    //перестановки
элементов
            if(position > countNotSortArr){
                return;
            }
            size_t newPosition = (position * 2) + forLambdaBIN[position];

            PR(newPosition, PR);

```

```

        if(forLambdaArr[position] > forLambdaArr[0]){
            std::swap(forLambdaArr[position], forLambdaArr[0]);
            forLambdaBIN[position] = !forLambdaBIN[position];
        }
    };

    PR(position, PR);
}

arr = forLambdaArr;

double a = difftime(clock(), t);

std::ofstream ft("result.txt", std::ios_base::app);

ft << arr.size() << ' ';
ft << a * 20 << '\n';

ft.close();
}

```

## ПРИЛОЖЕНИЕ Б

### ИСХОДНЫЙ КОД CUSTOMSORT

#### Файл customQsort.h

```
#pragma once

#include <vector>
#include <string>

void customQsort(std::vector<int> arrDigit);
std::vector<int> convert(std::string &strValue);
```

#### Файл customQsort.cpp

```
#include <iostream>
#include <ctime>
#include <fstream>
#include <algorithm>

#include "customQsort.h"

std::vector<int> convert(std::string &strValue){
    std::vector<int> arrDigit {};
    strValue.c_str();
    bool flag1 = 0; //флаги для отслеживания отрицательных
    значений
    bool flag2 = 0;
    for(size_t i = 0; strValue[i] != '\0'; i++){
        if(isdigit(strValue[i])){
            if(flag1){
                continue;
            }else{
                arrDigit.emplace_back(atoi(&strValue[i]));
                flag1 = 1;
            }
        }else if(strValue[i] == '-'){
            flag2 = 1;
        }else{
            flag1 = 0;
            if(flag2){
                arrDigit[arrDigit.size() - 1] -= (arrDigit.back() * 2);
                flag2 = !flag2;
            }
        }
    }
}
```

```

    }
    if(!arrDigit.size()){
        arrDigit.clear();
    }

    return arrDigit;
}

```

```

void customQsort(std::vector<int> arrDigit){

    clock_t t = clock();

    size_t l = 0;           // left index
    size_t r = arrDigit.size() - 1; // right index

    auto PR = [&arrDigit](int l, int r, auto&& PR){

        int left = l;
        int right = r;
        int base = arrDigit[(l + r) / 2];

        if(l >= r){
            return;
        }

        while(left < right){

            while(arrDigit[left] < base){
                ++left;
            }
            while(arrDigit[right] > base){
                --right;
            }
            if(left <= right){
                if(left != right){
                    std::swap(arrDigit[left], arrDigit[right]);
                }
                ++left;
                --right;
            }
        }
    }
}

```

```

        PR(l, right, PR);
        PR(left, r, PR);
    }
};

PR(l, r, PR);

double a = difftime(clock(), t);

std::ofstream ft("result2.txt", std::ios_base::app);

ft << arrDigit.size() << ' ';
ft << a * 20 << "\n";

ft.close();
}

```

## ПРИЛОЖЕНИЕ В

### ИСХОДНЫЙ КОД MAIN

#### Файл main.cpp

```
#include <iostream>
#include <sstream>
#include <vector>
#include <iterator>
#include <memory>

#include "WeakHeap.h"
#include "customQsort.h"

constexpr auto intervalElement = 2; // Величина для определения интервала

void generator(const size_t count, std::vector<int>& arr){ //Данная функция
генерирует рандомные значения для вектора arr.
    srand(time(0));
    for (size_t i = 0; i < count; i++){
        arr.push_back(rand() % 1000001);
    }
    std::sort(arr.begin(), arr.end());
}

int main(int argc, char* argv[]){

    std::string inputString {};
    getline(std::cin, inputString);
    std::stringstream ss(inputString);
    std::vector<int> inputVec {};
    std::copy(std::istream_iterator<int>(ss), {}, back_inserter(inputVec));

    auto test = std::make_shared<WeakHeap>();

    /*
    Если в массиве > одного элемента, то происходит обычная сортировка.
    Если же кол-во элементов == 1, то данное значение используется для
    формирования набора рандомных тестов.
    */

    if(inputVec.size() > 1){
```

```

    test->arr = inputVec;
    test->sortArr();
    for(const auto &a : test->arr){
        std::cout << a << ' ';
    }
    std::cout << std::endl;
} else if(inputVec.size() == 1){
    if(inputVec[0] <= 0){
        std::cout << "Кол-во тестов не может быть <= 0\n";
    } else{
        for(int i = 1; i <= inputVec[0]; i++){
            generator(intervalElement * i, test->arr);
            test->sortArr();
            customQsort(test->arr);
            test->arr.clear();
        }
    }
}

return 0;
}

```



## ПРИЛОЖЕНИЕ Г

### ИСХОДНЫЙ КОД SCRIPT1

#### Файл script1

```
#!/bin/bash

for n in {1..9}
do
    arg=$(cat Tests/test$n.txt)
    echo -e "\nTest $n:"
    echo "Test string = $arg"
    ./cw < Tests/test$n.txt
done
```

## ПРИЛОЖЕНИЕ Д

### ИСХОДНЫЙ КОД MAKEFILE

#### Файл Makefile

CC = g++

CCFLAGS = -std=c++2a -Wall

DIR = ./Source/

.PHONY: all clean

all: cw

cw: \$(DIR)main.cpp \$(DIR)WeakHeap.cpp \$(DIR)customQsort.cpp  
\$(CC) \$(CCFLAGS) \$^ -o \$@

clean:

rm \*.o lab\*

## ПРИЛОЖЕНИЕ Е

### ИСХОДНЫЙ КОД PLOT

#### Файл Plot.py

```
import sys
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import math
import numpy as np
```

```
file = sys.argv[1]
```

```
fs = file
```

```
f = open(fs, 'r')
```

```
x, y = [0], [0]
l = 0
```

```
m,n = 0,0
```

```
p = 0
```

```
for l in f:
    row = l.split()
    m += float(row[0])
    n += float(row[1])
    p += 1
    if p % 10 == 0:
        x.append(m / 10)
        y.append(n / 10)
```

```
f.close()
```

```
fig, ax1 = plt.subplots(
    nrows = 1, ncols = 1,
    figsize = (12, 12)
)
```

```
t = np.linspace(0.1, max(x))
```

```
a = np.log2(t) * t
```

```

v = t ** 2

xmax = math.sqrt(y[len(y) - 1])

nfg = np.linspace(0,xmax)

gfn = nfg * nfg

x.pop(0)
y.pop(0)
ax1.plot(x, y, label = 'Затраченное время')
ax1.plot(t, a, label = 'N logN')

ax1.plot(nfg, gfn, label = 'N^2 в крайней точке X')

ax1.grid(which = 'major',
         color = 'k')

ax1.minorticks_on()

ax1.grid(which = 'minor',
         color = 'gray',
         linestyle = ':')

ax1.legend()

ax1.set_xlabel('Количество элементов')

plt.title("Лучший случай")
plt.show()

```