

Оглавление

[Введение 1](#_Toc518239186)

[Теоретические основы 2](#_Toc518239187)

[Алгоритм сортировки слиянием 2](#_Toc518239188)

[Пример работы алгоритма сортировки слиянием: 3](#_Toc518239189)

[Алгоритм сортировки Timsort 4](#_Toc518239190)

[Пример работы алгоритма Timsort 8](#_Toc518239191)

[Оценка сложности алгоритмов: 9](#_Toc518239192)

[Описание компьютерной программы: 10](#_Toc518239193)

[Назначение 10](#_Toc518239194)

[Требования 10](#_Toc518239195)

[Требования к входным данным 10](#_Toc518239196)

[Требования к выходным данным 10](#_Toc518239197)

[Функциональные требования 11](#_Toc518239198)

[Архитектура 12](#_Toc518239199)

[Руководство пользователя 14](#_Toc518239200)

# Введение

С развитием технологий и появления все большего количества информации возникает необходимость отсортировать данную информацию. Зачастую стандартные алгоритмы сортировки не столь эффективны на определенных последовательностях элементов (частично упорядоченных например). Были разработаны так называемых гибридные алгоритмы сортировок, которые призваны сократить время упорядочивания данных.

Timsort — гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году Тимом Петерсом. В настоящее время Timsort является стандартным алгоритмом сортировки в Python, OpenJDK 7 и реализован в Android JDK 1.5. Основная идея алгоритма в том, что в реальном мире сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. На таких данных Timsort существенно быстрее многих алгоритмов сортировки.

Цель курсового проекта – исследовать алгоритмы сортировки timsort и его базовую составляющую - сортировку слиянием и выявить наиболее эффективное применение данных алгоритмов.

Задачи:

1. Изучить теоретические основы алгоритмов данных сортировок
2. Спроектировать и реализовать программу для экспериментов с разным количеством используемых наборов элементов
3. Провести ряд экспериментов

4. Проанализировать результаты

# Теоретические основы

## Алгоритм сортировки слиянием

Одним из методов сортировки слиянием называется простым слиянием, именно он и применяется в связке с алгоритмом Timsort.

1. Последовательность а разбивается на две половины b и c;
2. Последовательности b и c сливаются при помощи объединения отдельных элементов в упорядоченные пары;
3. Полученные последовательности присваивается имя a, и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в четверки.
4. Предыдущие шаги повторяются: четверки сливаются до тех пор, пока не будет упорядочена вся последовательность, ведь длины сливаемых последовательностей каждый раз удваиваются.

## Пример работы алгоритма сортировки слиянием:

44 55 12 42 94 18 06 67

На первом шаге делим последовательность на 2 подпоследовательности:

44 55 12 42  
94 18 06 67

На втором шаге делим каждую подпоследовательность на 2 подпоследовательности:

44 55  
12 42  
94 18  
06 67

На третьем шаге делим каждую подпоследовательность на 2 подпоследовательности. После останутся подпоследовательности длиной 1 элемент. Сливаем последовательности в обратном порядке:

12 42 44 45  
06 18 67 94

На четвертом шаге сливаем оставшиеся подпоследовательности:

06 12 18 42 44 55 67 94

## Алгоритм сортировки Timsort

Алгоритм Timsort ищет в массиве упорядоченные последовательности, называемые run, для ускорения поиска.

**N** — размер входного массива  
**run** — упорядоченный подмассив во входном массиве. Причём упорядоченный либо нестрого по возрастанию, либо строго по убыванию.  
**minrun** — как было сказано выше, на первом шаге алгоритма входной массив будет поделен на подмассивы. minrun — это минимальный размер такого подмассива. Это число рассчитывается по определённой логике из числа N.  
  
Шаг 0:

(1) Число minrun (минимальный размер упорядоченной последовательности) определяется на основе N исходя из следующих принципов: оно не должно быть слишком большим, поскольку к подмассиву размера minrun будет в дальнейшем применена сортировка вставками, а она эффективна только на небольших массивах.  
(2) Оно не должно быть слишком маленьким, поскольку чем меньше подмассив — тем больше итераций слияния подмассивов придётся выполнить на последнем шаге алгоритма. Оптимальная величина для N / minrun это степень числа 2 (или близким к нему). Это требование обусловлено тем, что алгоритм слияния подмассивов наиболее эффективно работает на подмассивах примерно равного размера.  
В этом месте автор алгоритма ссылается на собственные эксперименты, показавшие, что при minrun > 256 нарушается пункт (1), при minrun < 8 — пункт (2) и наиболее эффективно использовать значения из диапазона (32;65). Исключение — если N < 64, тогда minrun = N и timsort превращается в простую сортировку вставкой. В данный момент алгоритм расчёта minrun предельно прост: берутся старшие 6 бит из N и добавляется единица, если в оставшихся младших битах есть хотя бы один ненулевой.

Псевдокод:

int GetMinrun(int n)

{

int r = 0; /\* станет 1 если среди сдвинутых битов будет хотя бы 1 ненулевой \*/

while (n >= 64) {

r |= n & 1;

n >>= 1;

}

return n + r;

}

**Шаг 1. Разбиение на подмассивы и их сортировка.**

1. Указатель текущего элемента ставится в начало входного массива.
2. Начиная с текущего элемента, в этом массиве идёт поиск упорядоченного подмассива run. По определению, в run однозначно войдет текущий элемент и следующий за ним. Если получившийся подмассив упорядочен по убыванию — элементы переставляются так, чтобы они шли по возрастанию.
3. Если размер текущего run’а меньше, чем minrun — выбираются следующие за найденным run-ом элементы в количестве minrun-size(run). Таким образом, на выходе будет получен подмассив размером minrun или больше, часть которого (а в идеале — он весь) упорядочена.
4. К данному подмассиву применяется сортировка вставками. Так как размер подмассива невелик и часть его уже упорядочена — сортировка работает быстро и эффективно.
5. Указатель текущего элемента ставится на следующий за подмассивом элемент.
6. Если конец входного массива не достигнут — переход к пункту 2, иначе — конец данного шага.

**Шаг 2. Слияние**

Если данные входного массива были близки к случайным — размер упорядоченных подмассивов близок к minrun, если в данных были упорядоченные диапазоны — упорядоченные подмассивы имеют размер, превышающий minrun.

Нужно объединить эти подмассивы для получения результирующего, полностью упорядоченного массива. Для достижения эффективности Объединение должно удовлетворять двум требованиям:

1. Объединять подмассивы примерно равного размера
2. Сохранить стабильность алгоритма — то есть не делать бессмысленных перестановок.
3. Алгоритм:
4. Создается пустой стек пар <индекс начала подмассива>-<размер подмассива>. Берётся первый упорядоченный подмассив.
5. В стек добавляется пара данных <индекс начала>-<размер> для текущего подмассива.
6. Определяется, нужно ли выполнять процедуру слияния текущего подмассива с предыдущими. Для этого проверяется выполнение двух правил (пусть X, Y и Z — размеры трёх верхних в стеке подмассивов):

X > Y + Z

Y > Z

1. Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z. Повторяется до выполнения обоих правил или полного упорядочивания данных.
2. Если еще остались не рассмотренные подмассивы — берётся следующий и переходим к пункту 2. Иначе — конец.

Цель этой процедуры — сохранение баланса. Изменения будут выглядеть как на картинке справа, а значит, размеры подмассивов в стеке эффективны для дальнейшей сортировки слиянием. В идеальном случае: есть подмассивы размера 128, 64, 32, 16, 8, 4, 2, 2. В этом случае никакие слияния не выполнятся, пока не встретятся 2 последних подмассива, после чего будут выполнены 7 идеально сбалансированных слияний.

**Процедура слияния подмассивов**

1. Создаётся временный массив в размере меньшего из соединяемых подмассивов.
2. Меньший из подмассивов копируется во временный массив
3. Указатели текущей позиции ставятся на первые элементы большего и временного массива.
4. На каждом следующем шаге рассматривается значение текущих элементов в большем и временном массивах, берётся меньший из них и копируется в новый отсортированный массив. Указатель текущего элемента перемещается в массиве, из которого был взят элемент.
5. Пункт 4 повторяется, пока один из массивов не закончится.
6. Все элементы оставшегося массива добавляются в конец нового массива.

Модификация процедуры слияния подмассивов (Galloping mode)

Представим себе процедуру слияния следующих массивов:

A = {1, 2, 3,..., 9999, 10000}

B = { 20000, 20001, ...., 29999, 30000}

Вышеуказанная процедура для них сработает, но каждый раз на её четвёртом пункте нужно будет выполнить одно сравнение и одно копирование. В итоге 10000 сравнений и 10000 копирований. Алгоритм Timsort предлагает в этом месте модификацию, которую он называет «галоп». Алгоритм:

1. Начинается процедура слияния, как было показано выше.
2. На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминается, из какого именно подмассива был элемент.
3. Если уже некоторое количество элементов (в данной реализации алгоритма это число равно 7) было взято из одного и того же массива — предполагается, что и дальше нам придётся брать данные из него. Чтобы подтвердить эту идею, алгоритм переходит в режим «галопа», то есть перемещается по массиву-претенденту на поставку следующей большой порции данных бинарным поиском (массив упорядочен) текущего элемента из второго соединяемого массива.
4. В момент, когда данные из текущего массива-поставщика больше не подходят (или был достигнут конец массива), данные копируются целиком.

В примере первые 7 итераций сравниваются числа 1, 2, 3, 4, 5, 6 и 7 из массива A с числом 20000, так как 20000 больше — элементы массива A копируются в результирующий. Начиная со следующей итерации алгоритм переходит в режим «галопа»: сравнивает с числом 20000 последовательно элементы 8, 10, 14, 22, 38, n+2^i, …, 10000 массива A. (~log2 N сравнений). После того как конец массива A достигнут и известно, что он весь меньше B, нужные данные из массива A копируются в результирующий.

## Пример работы алгоритма Timsort

(условный пример, т.к. в реальности Timsort применяют для массивов размерности больше 64. Т.к. до 64 элементов Timsort упорядочивает массив сортировкой вставками.)

44 55 12 42 94 18 06 67 44 55 12 42 94 18 06 67 44 55 12 42 94 18 06 67

На нулевом шаге вычислим minrun (4, например).  
На первом шаге найдем в массиве подпоследовательности и отсортируем их вставками, параллельно добавляя в стек индексы вхождений подпоследовательности в исходный массив и их размерность:

|  |  |
| --- | --- |
| 6 | 12 42 44 55 |
| 5 | 06 18 67 94 |
| 4 | 12 42 44 55 |
| 3 | 06 18 67 94 |
| 2 | 12 42 44 55 |
| 1 | 06 18 67 94 |

Второй шаг – слияние:  
На данный момент размер стек - 6. Достаём из верхушки стека первые три элемента. Условие нарушается, следовательно, сливаем верхний (первый) элемент стека с минимальным из второго и третьего.

|  |  |
| --- | --- |
| 5 | 06 12 18 42 44 55 67 94 |
| 4 | 12 42 44 55 |
| 3 | 06 18 67 94 |
| 2 | 12 42 44 55 |
| 1 | 06 18 67 94 |

|  |  |
| --- | --- |
| 4 | 06 12 12 18 42 42 44 44 55 55 67 94 |
| 3 | 06 18 67 94 |
| 2 | 12 42 44 55 |
| 1 | 06 18 67 94 |

|  |  |
| --- | --- |
| 3 | 06 12 12 18 42 42 44 44 55 55 67 94 |
| 2 | 06 12 18 42 44 55 67 94 |
| 1 | 06 18 67 94 |

|  |  |
| --- | --- |
| 2 | 06 06 12 12 18 18 42 42 44 44 55 55 67 67 94 94 |
| 1 | 06 12 18 42 44 55 67 94 |

|  |  |
| --- | --- |
| 1 | 06 06 06 12 12 12 18 18 18 42 42 42 44 44 44 55 55 55 67 67 67 94 94 94 |

## Оценка сложности алгоритмов:

Анализ сортировки слиянием[[1]](#endnote-1):  
Поскольку на каждом шагу p (количество подпоследовательностей увеличивается) и сортировка заканчивается, как только p > n, она требует   
M = n \* log n пересылок; число С сравнений еще меньше чем M, так как при копировании остатка последовательности сравнения не производится.

|  |  |  |
| --- | --- | --- |
| Сложность по времени | Лучший случай  Средний случай  Худший случай | O(*n* log *n*) |
| Сложность по памяти | Доп.память | O(*n)* |

Анализ сортировки слиянием[[2]](#endnote-2):  
Чем меньше массивов, тем меньше произойдёт операций слияния, но чем их длины больше, тем дольше эти слияния будут происходить. На малом количестве длинных массивов хорошо помогает вышеописанный метод Galloping Mode.

|  |  |  |
| --- | --- | --- |
| Сложность по времени | Лучший случай  Средний случай  Худший случай | O(*n)*  O(*n* log *n*)  O(*n* log *n*) |
| Сложность по памяти | Доп.память | O(*n)* |

# Описание компьютерной программы:

## Назначение

1. Генерировать файлы, содержащие последовательность с определёнными свойствами (количество элементов, порядок).
2. Провести эксперимент по сравнению TimSort с MergeSort к каждому файлу и вывести столбчатые диаграммы, показывающие время выполнения, количества обменов и сравнений сортировок.

## Требования

### **Требования к входным данным**

**Для генерации:**

1. Тип последовательности из множества SequenceType.
2. Число в диапазоне [, ], обозначающее количество элементов в генерируемой последовательности.

Для проведения эксперимента:

1. Непустой список файлов, каждый файл из данного списка является файлом типа из множества SequenceType.

### **Требования к выходным данным**

**Для генерации:**

Файл формата из множества SequenceType = {возрастающая (up), убывающая (dec), повторяющаяся (rep), случайная (rand)}, название которого строиться по схеме:

AmountOfElements.SequenceCopyNumber

AmountOfElements – число, записанное в десятичной системе счисления. Означает количество элементов, содержащихся в выходном файле.  
Sequence – элемент из SequenceType (сокращённое название в круглых скобках).

CopyNumber – номер копии, если указанная последовательность уже существует.

**Для проведения эксперимента:**

Столбчатые диаграммы, показывающие время выполнения, количество обменов и сравнений на трёх вкладках соответственно, произведенных Timsort и сортировкой слиянием при сортировке элементов выбранных файлов.

### **Функциональные требования**

**Для генерации:**

1. Программа не должна позволять клиенту ввести некорректное значение последовательности или количества элементов.
2. Программа должна уметь считать число количества элементов для генерации последовательности в диапазоне от 32 до 1000000.
3. Программа должна уметь создавать файл с последовательностью выбранного типа введенной длины с корректным именем.

**Для проведения эксперимента:**

1. Программа должна предоставить возможность выбора списка файлов для проведения эксперимента и проверить количество выбранных файлов, которое должно быть не меньше одного.
2. Программа должна произвести Timsort и сортировку слиянием для каждого файла и сохранить затраченное на сортировку время, количество обменов и сравнений совершенных каждой из сортировок.
3. Программа должна вывести столбчатую диаграмму, показывающую зависимость количества обменов и сравнений от типа последовательности и количества элементов во входных файлах.

## Архитектура

Архитектура компьютерной программы представлена на рисунке 1. Пунктирными стрелками отмечен обмен данными между блоками КП (Компьютерной программы, далее КП). Прочие линии показывают возможность перехода от одного модуля к другому. Подчёркиванием обозначены модули КП, с которыми пользователь может взаимодействовать. Прямоугольными скобками обозначены модули, связанные с внутренним устройством. Фигурными скобками обозначены папки и списки для хранения последовательностей

Main menu

[Sequence Generator]

*{File list}*

Рисунок 1 - Архитектура компьютерной программы

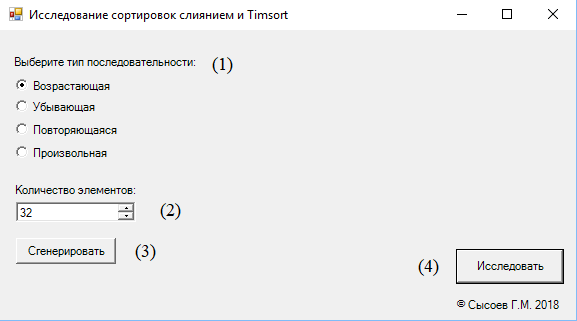
Experiment form

На рисунке 1. представлены:

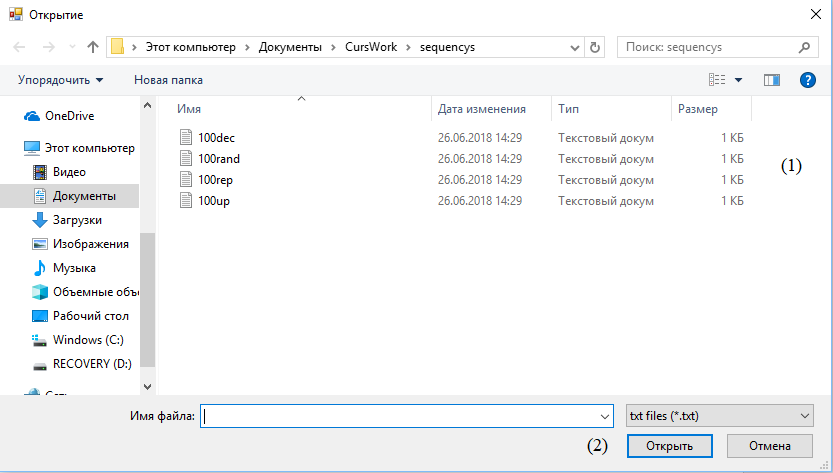
1. Main menu — модуль КП (Далее МКП), предоставляет возможность выбрать тип последовательности и количество элементов для генерации. Обеспечивает доступ к File list.
2. Sequence Generator — МКП, который выполняет генерацию необходимых пользователю последовательностей, сохраняющихся в File list.
3. Files List — список сгенерированных пользователем файлов для эксперимента. Из этого списка пользователь выбирает последовательности для проведения эксперимента.
4. Experiment Form — МКП, окно, на которое выводятся выходные данные эксперимента.

## **Руководство пользователя**

Пользователь запускает программу kursovaya, после чего открывается главное окно программы с открытой вкладкой «Исследование сортировок слиянием и Timsort» (рисунок 2 – главное меню программы).

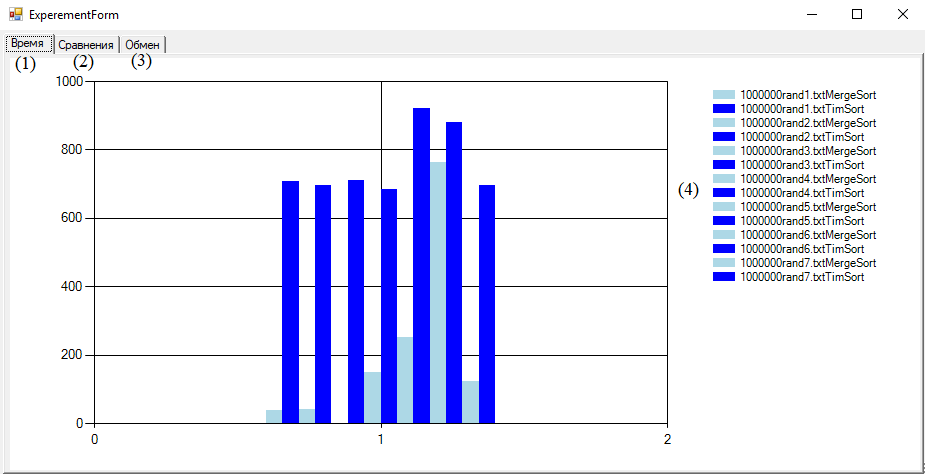
Рисунок 2 – главное меню программы.

(1) Кнопки выбора последовательности. Может быть выбран только один тип последовательности. (Изначально выбрана кнопка «Возрастающая»)  
(2) Текстовое поле для ввода количества элементов в генерируемой последовательности.  
(3) Кнопка, при нажатии на которую произойдёт генерация файла с последовательностью выбранного пользователем типа из SequenceType и количеством элементов.  
(4) Кнопка, при нажатии на которую программа открывает окно выбора файлов для эксперимента (Рисунок 3 – окно выбора файлов) и блокирует окно «Исследование сортировок слиянием и Timsort».

Рисунок 3 – окно выбора файлов.

(1) Список сгенерированных текстовых файлов с последовательностями типа SequenceType. Пользователь может выбрать несколько файлов.

(2) При нажатии на кнопку открыть (после того как пользователь выбрал текстовые файлы) программа закроет окно выбора файлов и откроет окно проведения эксперимента (Рисунок 4). Если ни одного файла не было выбрано и пользователь закроет окно (нажмёт кнопку «отмена»), программа разблокирует окно «Исследование сортировок слиянием и Timsort».

Рисунок 4 – окно результатов эксперимента

(1) Вкладка на которой представлен график сравнения времени выполнения сортировок на выбранных последовательностях.  
(2) Вкладка на которой представлен график сравнения количества сравнений сортировок на выбранных последовательностях.  
(3) Вкладка на которой представлен график сравнения количества обменов на выбранных последовательностях.  
(4) Список выбранных последовательностей, которые отображаются на графике слева на право (сверху вниз в списке).

Эксперименты

Входные данные:  
Возрастающие последовательности на 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

Цель:  
Проверить поведение сортировок на последовательностях одного типа (возрастающая последовательность) разной длины.

Выходные данные: приведены на рисунке 5.  
Обоснование:  
Согласно рисунку 5,

1. Н.Вирт АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ М.: Мир, 1989 [↑](#endnote-ref-1)
2. Peter McIlroy "Optimistic Sorting and Information Theoretic Complexity", Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, ISBN 0-89871-313-7, Chapter 53, pp 467-474, January 1993. [↑](#endnote-ref-2)