

Оглавление

[1. Введение 3](#_Toc516176632)

[2. Теоретические основы 4](#_Toc516176633)

[2.1. Определения 4](#_Toc516176634)

[2.2. Алгоритм сортировки слиянием 4](#_Toc516176635)

[2.3. Алгоритм сортировки timsort 5](#_Toc516176637)

3. [Анализ алгоритма 8](#_Toc516176647)

4. [Литература 9](#_Toc516176648)

# 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). Поэтому 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 программу для экспериментов с разным количеством используемых наборов элементов
3. Провести ряд экспериментов

4. Проанализировать результаты

# Теоретические основы

## Определения

Упорядоченный по возрастанию массив – массив размерности от 1 до N, в котором каждый последующий (i + 1) элемент больше или равен предыдущему (i). i E (1..N).

Упорядоченный строго по убыванию массив – массив размерности от 1 до N, в котором каждый последующий (i + 1) элемент меньше предыдущего (i). i E (1..N).

* 1. **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:

44 55 12 42 94 18 06 67

На первом шаге делим данную последовательность на 2 подпоследовательности:

44 55 12 42   
94 18 06 67

Слияние отдельных компонент (которые являются упорядоченными последовательностями длины 1) в упорядоченные пары даёт

06 12 18 42 44 55 67 94

Стр 110 вирт.

* 1. **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. Вычисление minrun.**

(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.**

* Указатель текущего элемента ставится в начало входного массива.
* 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.
* 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.
* К данному подмассиву применяется сортировка вставками. Так как размер подмассива невелик и часть его уже упорядочена — сортировка работает быстро и эффективно.
* Указатель текущего элемента ставится на следующий за подмассивом элемент.
* 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:

* Объединять подмассивы примерно равного размера
* Сохранить стабильность алгоритма — то есть не делать бессмысленных перестановок.
* Алгоритм:
* Создается пустой стек пар <индекс начала подмассива>-<размер подмассива>. Берётся первый упорядоченный подмассив.
* В стек добавляется пара данных <индекс начала>-<размер> для текущего подмассива.
* 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):
* Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z. Повторяется до выполнения обоих правил или полного упорядочивания данных.
* 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:

* Начинается процедура слияния, как было показано выше.
* На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминается, из какого именно подмассива был элемент.
* 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.
* В момент, когда данные из текущего массива-поставщика больше не подходят (или был достигнут конец массива), данные копируются целиком.

Режим галопа на примере:

Исходные массивы:

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

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

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.

# 3.Анализ алгоритма

Анализ сортировки слиянием:

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.

# Литература

* 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
* *Magnus Lie Hetland.* Python Algorithms: Mastering Basic Algorithms in the Python Language. — Apress, 2010.
* Н.Вирт АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ М.: Мир, 1989