ПРЕЗЕНТАЦИЯ ПО ИНФОРМАТИКЕ

НА ТЕМУ

TIMSORT – АЛГОРИТМ СОРТИРОВКИ

Выполнили учащиеся

10ф и 10м классов

Шафир Александр и Мещерин Павел

Академическая гимназия имени Д. К. Фаддеева при СПбГУ

Введение

Timsort - гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием. 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.

Оценки алгоритма

Худшая производительность	$O(n \log n)$
Лучшая производительность	O(n)
Средняя производительность	$O(n \log n)$
Максимальный объем занимаемой памяти	O(n)

Базовый принцип

Сортировка посредством алгоритма timsort сочетает в себе 4 шага.

1. Высчитывание *minrun* - минимального размера подмассива
2. Разделение основного массива на подмассивы размера *minrun* или меньше
3. Сортировка полученных подмассивов вставками/пузырьком или другими видами сортировок
4. Сборка подмассивов обратно в цельный массив сортировкой слиянием.

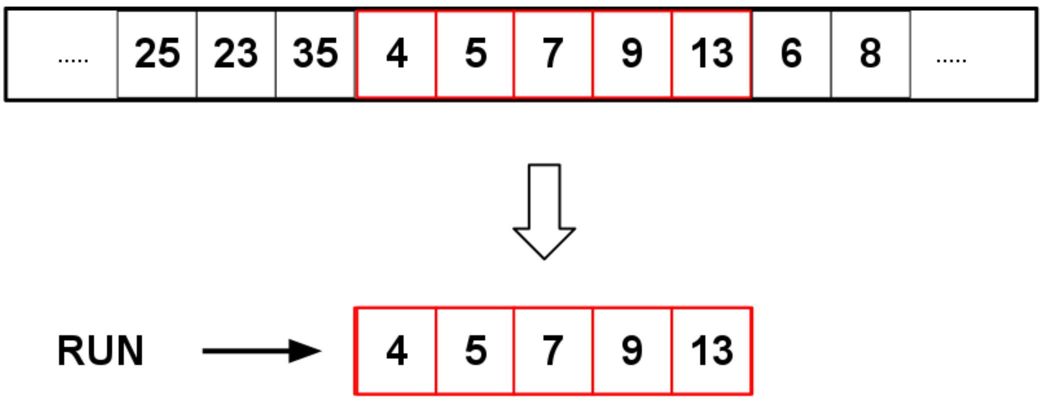
Как отмечалось ранее, в реальной жизни случайный набор данных может содержать в себе уже отсортированные массивы, на этом и заточен принцип сортировки timsort. Программа «вылавливает» уже отсортированные куски и добавляет к ним оставшиеся данные.

Шаг 0 - Вычисление minrun

Поиск упорядоченных элементов в массиве и составление из них меньших массивов, наиболее эффективно брать длину массива из диапазона 32 - 65

Шаг 1 - Разбиение на подмассивы и их сортировка:

1. Ставим указатель текущего элемента в начало входного массива.
2. Начиная с текущего элемента, ищем во входном массиве **run** (упорядоченный подмассив). По определению, в этот **run** однозначно войдет текущий элемент и следующий за ним, а вот дальше — уже как повезет. Если получившийся подмассив упорядочен по убыванию — переставляем элементы так, чтобы они шли по возрастанию (это простой линейный алгоритм, просто идём с обоих концов к середине, меняя элементы местами).
3. Если размер текущего **run**'а меньше чем **minrun** — берём следующие за найденным **run**-ом элементы в количестве **minrun — size(run)**. Таким образом, на выходе у нас получается подмассив размером **minrun** или больше, часть которого (а в идеале — он весь) упорядочена.
4. Применяем к данному подмассиву сортировку вставками. Так как размер подмассива невелик и часть его уже упорядочена — сортировка работает быстро и эффективно.
5. Ставим указатель текущего элемента на следующий за подмассивом элемент.
6. Если конец входного массива не достигнут — переход к пункту 2, иначе — конец данного шага.

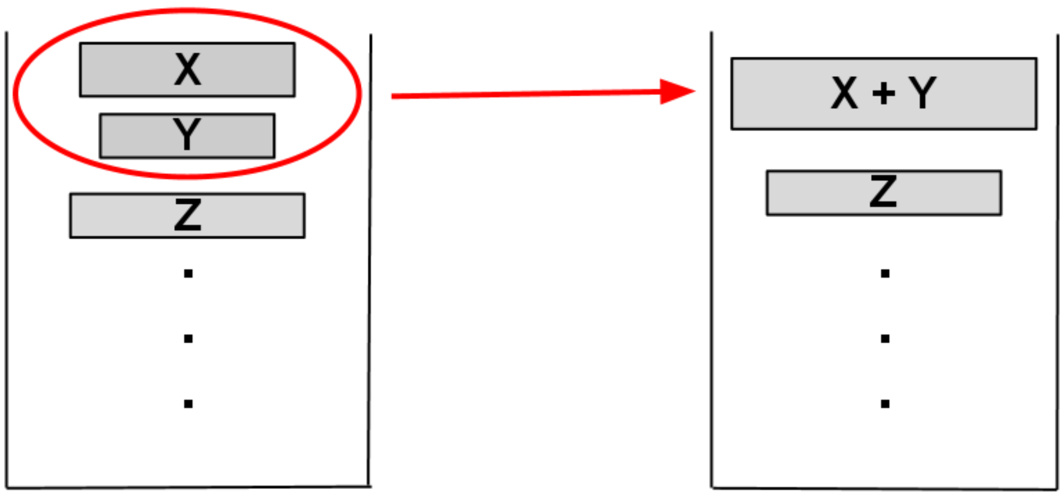


Шаг 2 - Слияние, условия

1. Объединять подмассивы примерно равного размера
2. Сохранить стабильность алгоритма

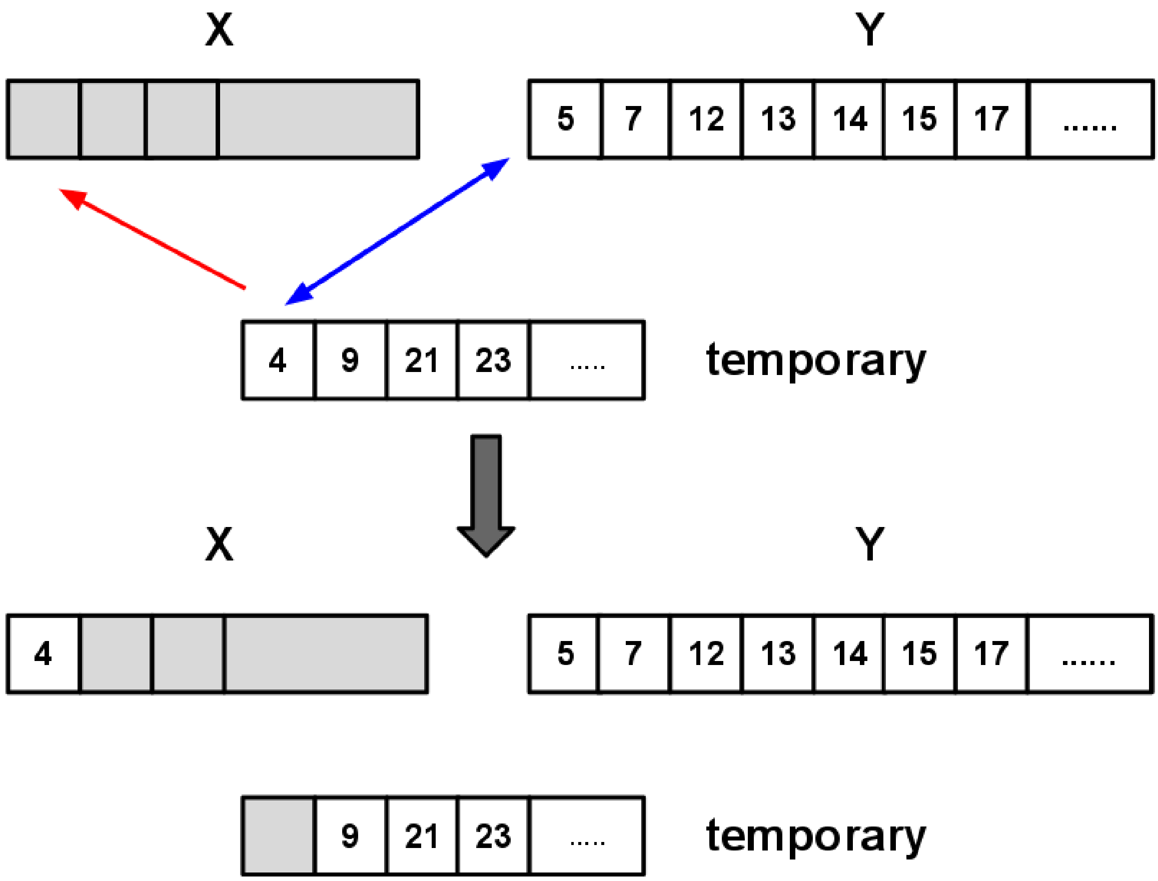
Алгоритм

1. Создается пустой стек пар <индекс начала подмассива> - <размер подмассива>
2. В стек добавляется пара данных <индекс начала>-<размер> для текущего подмассива
3. Определяется, нужно ли выполнять процедуру слияния текущего подмассива с предыдущими
4. Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z. Повторяется до выполнения обоих правил или полного упорядочивания данных
5. Если еще остались не рассмотренные подмассивы — берётся следующий и переходим к пункту 2. Иначе — конец



Процедура сливания массивов

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



Пример алгоритма

Пусть у нас есть массив [2, 5, 7, 4, 6, 3, 9, 8]

Пусть minrun - 4

1)Происходит разбитие на подмассивы [2, 5, 7, 4] и [3, 6, 9, 8]

2)Сортировка вставками в подмассивах [2, 4, 5, 7] и [3, 6, 8, 9]

3)Сортировка слиянием подмассивов [2, 3, 4, 5, 6, 7, 8, 9]

Доказательство времени работы алгоритма

Не сложно заметить, что чем меньше массивов, тем меньше произойдёт операций слияния, но чем их длины больше, тем дольше эти слияния будут происходить. Пусть k — число кусков, на которые разбился наш исходный массив, очевидно k = $\left\lceil \frac{\mathtt{n}}{\mathtt{minrun}} \right\rceil$.

Главный факт, который нам понадобится для доказательства нужной оценки времени работы в $O(n \log n)$ — эт

о то, что сливаемые массивы **всегда** имеют примерно одинаковую длину. Можно сказать больше: пока $k > 3$ сливаемые подмассивы будут именно одинаковой. Безусловно, после разбиения массива на блоки длиной $\mathtt{minrun}$ последний блок может быть отличен от данного значения, но число элементов в нём не превосходит константы $\mathtt{minrun}$.

При слиянии, длинна образовавшегося слитого массива увеличивается в ≈ 2 раза. Таким образом, каждый подмассив $\mathtt{run_i}$ может участвовать в не более $O(\log n)$ операций слияния, а значит и каждый элемент будет задействован в сравнениях не более $O(\log n)$ раз. Элементов n, откуда получаем оценку в $O(n \log n)$.

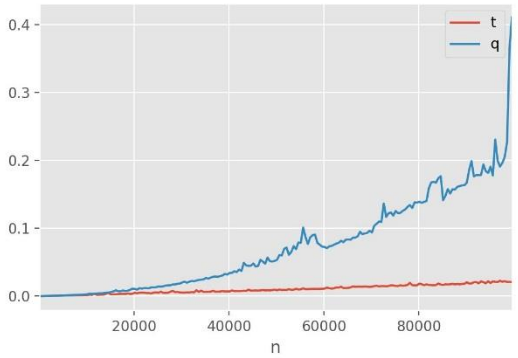
Также нужно сказать про сортировку вставками, которая используется для сортировки подмассивов $\mathrm{run_i}$: в нашем случае, алгоритм работает за $O(\mathtt{minrun} + \mathtt{inv})$, где $\mathtt{inv}$ — число обменов элементов входного массива, равное числу инверсий. С учетом значения k получим, что сортировка всех блоков может занять $O(\mathtt{minrun} + \mathtt{inv}) \cdot k = O(\mathtt{minrun} + \mathtt{inv}) \cdot \left\lceil \frac{\mathtt{n}}{\mathtt{minrun}} \right\rceil$. Что в худшем случае $(\mathtt{inv} = \frac{\mathtt{minrun}(\mathtt{minrun} - 1)}{2})$ может занимать $O(\mathtt{n} \cdot \mathtt{minrun})$ времени. Откуда видно, что константа $\mathtt{minrun}$ играет немалое значение: при большом $\mathtt{minrun}$ слияний будет меньше, а сортировки вставками будут выполняться долго. Причём эти функции растут с разной скоростью, поэтому ещё после экспериментов на различных значениях и был выбран оптимальный диапазон — от 32 до 64.

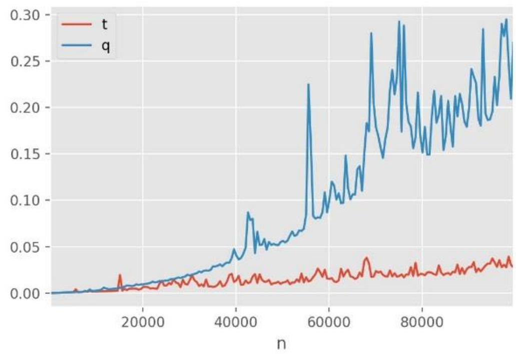
Доказательство корректности метода

//пока ничего нет

Сравнение с QSORT

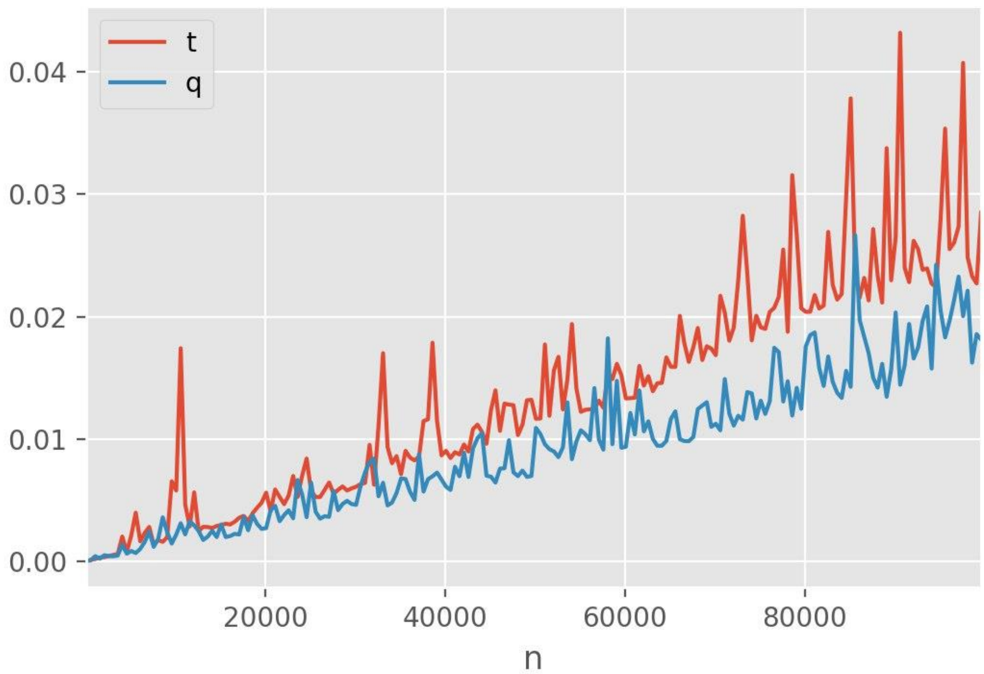
Построим несколько графиков зависимости времени *t* от количества элементов *n*.



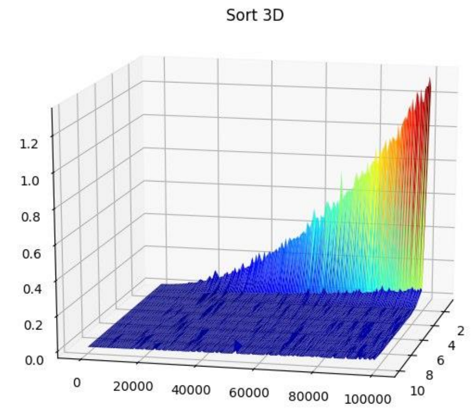


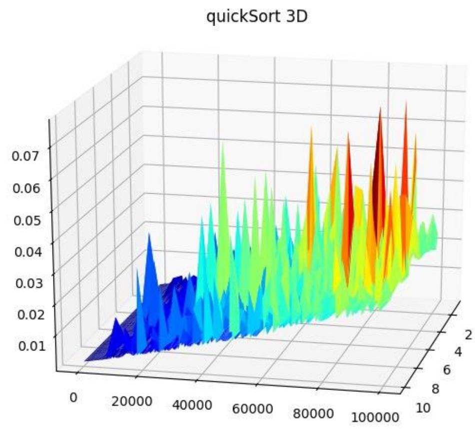
Заметим, что график TimSort почти линеен и сольно выигрывает по сравнению с quickSort. В ходе разбирательств в коде понимаем, что рандомное число лежало в пределах от -100 до 100. Увеличив этот диапазон в 100 раз, построим еще один график:

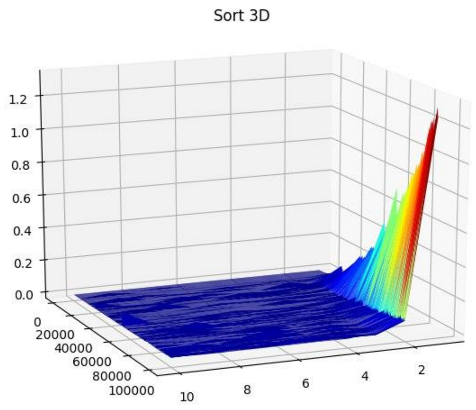
Делаем вывод что алгоритм сильно зависит от ширины выборки, поэтому вводим метрику r, такое что $(-10r;10r)$ - диапазон доступных чисел из которых

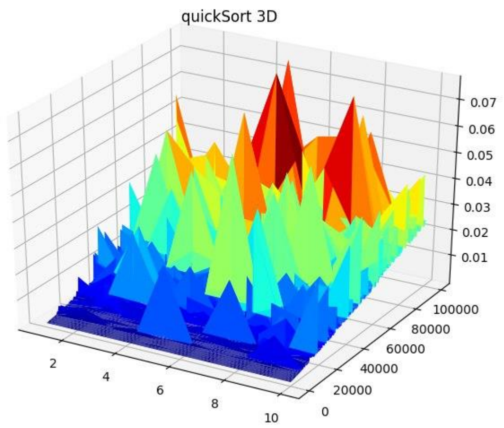


Делаем вывод, что TimSort эффективнее QuickSort только до определенного момента. выбирается рандомное. Получается 3D график:









Делаем вывод, что TimSort эффективнее QuickSort только до определенного момента.

Примечания:

Исходники с сырыми данными и средствами их визуализации лежат тут.

Красивые визуализации можно найти:

- youtube (самый сок это аудио)
- GIF

Литература:

- http://bugs.python.org/file4451/timsort.txt
- https://en.wikipedia.org/wiki/Timsort

P.S.: [some graphs]