Санкт-Петербургский государственный университет
Факультет прикладной математики — процессов управления

"Алгоритмы и анализ сложности"

Алгоритм поразрядной MSD-сортировки

Работу выполнил:
Михайлов Данил Дмитриевич
Группа: 22.Б12-ПУ

Преподаватель:
Никифоров К.А.

Санкт-Петербург
2024 г.

Содержание

Описание алгоритма

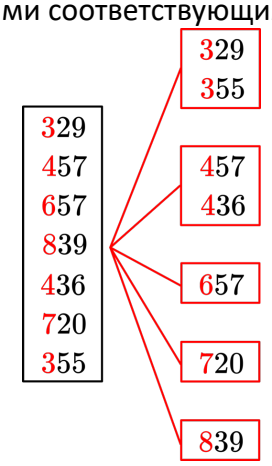
В общем случае алгоритм поразрядной MSD-сортировки — это упорядочивание множества объектов в лексикографическом порядке. Обычно в качестве таких объектов выступают числа или строки, которые тоже можно считать числами в 256-разрядной системе счисления.

История разработки

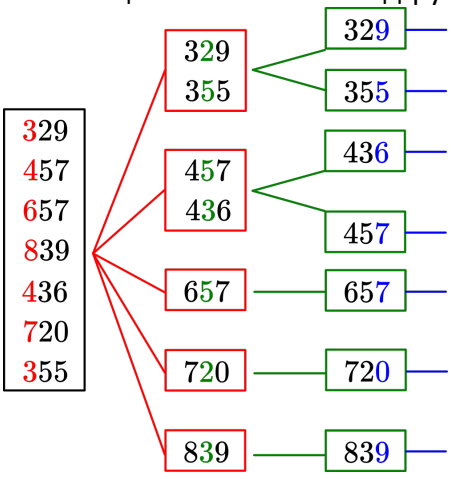
Первые идеи поразрядной сортировки восходят к концу 19 века. Герман Холлерит, американский изобретатель, использовал принципы поразрядной сортировки в своих табуляторах для обработки данных переписи населения США 1890 и 1990 годов [3].
В 1954 году Гарольд Сьюард (Harold H. Seward) формализовал и описал алгоритм поразрядной сортировки, включая варианты LSD- и MSD-сортировок [2].

Идея алгоритма

1. Имея набор чисел, которые нужно отсортировать, разбиваем его на подгруппы в зависимости от значения текущего разряда таким образом, чтобы в каждой группе оказались элементы с одинаковыми соответствующими значениями:



2. Далее каждую группу рассматриваем отдельно. Для конкретной взятой группы выполняем шаг 1, разбивая её на ещё более мелкие подгруппы.



Процесс продолжается до тех пор, пока не будут обработаны все разряды или пока все подгруппы не будут полностью отсортированы (если дошли до подгрупп размером 1, то каждую такую подгруппу можем считать отсортированной).

Области применения

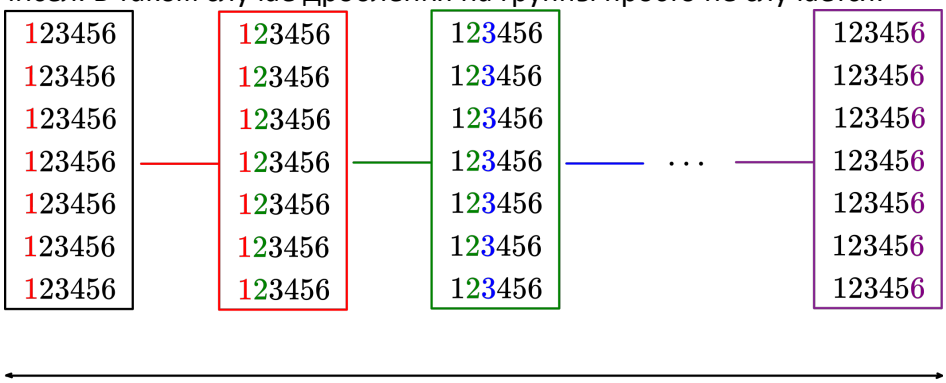
В любой области, где нужна эффективная сортировка строк произвольной длины, используют данный алгоритм сортировки. Например, в современном проводнике, в СУБД, в задачах обработки текстов и биоинформатике (цепи ДНК), в сетевых приложениях для сортировки IP-адресов, в компьютерной графике для сортировки объектов по их координатам и так далее.

Априорный анализ алгоритма

Пусть в сортируемом наборе чисел всего k разрядов. А значение каждого разряда не превосходит какое-то число d. Тогда сложность алгоритма будет принадлежать одному из трёх классов в зависимости от распределения входных данных.

Наихудший случай

Самым плохим вариантом исходных данных для MSD-сортировки является набор из одинаковых чисел. В таком случае дробления на группы просто не случается:

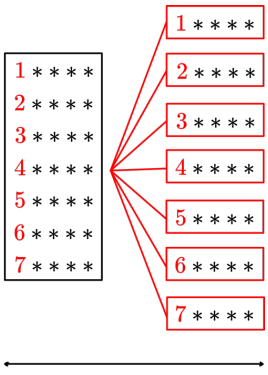


Тогда для каждого разряда попытка разбиения будет неудачной. Так как всего разрядов k штук, а сложность разбиения на подгруппы считаем линейной, то получаем сложность самого худшего случая:

$$O(n \cdot k)$$

Наилучший случай

По аналогии с тем, что худший случай заключается в том, что подгрупп больше не становится, то есть не происходит никакого разбиения задачи на подзадачи, лучший случай будет достигаться тогда, когда это разбиение наибольшее. То есть когда все элементы различаются уже в самом первом разряде:



$$O(n \cdot 1) = O(\mathrm{n}), n \leq d$$

Проблема заключается в том, что тогда размер массива должен быть не больше, чем максимальное значение разряда: если в массиве больше 10-ти элементов, то как минимум у каких-то двух первый разряд совпадёт, то есть они попадут в одну группу, значит, одним разбиением уже не обойтись.

Средний случай

На случайных данных чаще всего будет происходить деление на группы не единичного размера. Если предположить, что на каждом шаге происходит деление на d групп, то сложность станет вот такой:

$$\Omega(n \cdot log_{\mathrm{d}}(n))$$

Характеристики входных данных и единицы измерения трудоёмкости

Диапазон, размер и тип входных данных

Тип данных: целые числа без знака (unsigned integers).
Диапазон значений: от 0 до 2^{32} - 1.
Размер: от 10,000 до 1,000,000 с шагом в 10,000.

Программная реализация на языке Python

Мы будем создавать несколько массивов заданных размеров, заполненных “случайными” (насколько позволяет метод random.randint) числами при помощи класса SimpleInputGenerator. Сортироваться массив будет при помощи msd_radix_sort(). Для измерения трудоёмкости будем использовать 2 функции:

1. Функция, которая проводит эксперименты на *массивах разных размеров* (задаются размер самого маленького массива, самого большого и шаг изменения размера). Она возвращает подробную информацию для каждого размера: каково было худшее, среднее и лучшее время работы, аналогично — с числом базовых операций.
2. Функция, которая проводит эксперименты на *разных массивах одного и того же размера*. После сортировки каждого такого массива запоминается, сколько операций и времени было потрачено.

Для анализа экспериментов над массивами разной длины построим график зависимости между временем/количеством операций и размером массива. Все значения времени/количества операций разделим на первые числа (пронормируем), чтобы перейти от абсолютных значений к относительным. Шкалы осей будут логарифмическими, чтобы все кривые поместились на одном графике. Для оценки класса сложности трудоёмкости на том же графике добавим функции квадратичной и кубической сложности.

Чтобы проанализировать результаты экспериментов над массивами одной длины построим гистограмму: по оси абсцисс будут находиться количества потраченных операций, а по оси ординат — частоты (”Сколько массивов из 10,000 отсортировались за x операций?”). Анализировать будем именно число операций, а не время выполнения, так как число операций измеряется в тысячах, а время выполнения — в десятых и сотых долях, что больше подвержено арифмитическим погрешностям компьютера.

Характеристики использованной вычислительной среды и оборудования

Имя ОС	macOS Sequoia 15.0
Изготовитель ОС	Apple Inc.
Изготовитель ноутбука	Apple Inc.
Процессор	Apple M1
Версия BIOS	Н/Д
Установленная оперативная память (RAM)	8 ГБ
Число ядер	8
Число логических процессоров	8
Версия интепритатора	Python 3.13.0

Вычислительный эксперимент

Для массивов размерами от 10,000 до 1,000,000 с шагом 10,000, на каждом из которых алгоритм работал по 10 раз, получились следующие результаты:

input size	*worst time*	*avg time*	*best time*	*worst ops*	*avg ops*	*best ops*
10,000	0.011726	0.011324	0.011159	65,909	65,792.6	65,545
20,000	0.023114	0.022982	0.022917	136,834	136,728.1	136,621
30,000	0.035464	0.035055	0.034755	210,153	209,806.9	209,355
40,000	0.047496	0.047383	0.047327	285,538	285,265.6	284,942
…						
80,000	0.099820	0.099017	0.098534	599,174	598,864.9	598,489
…						
990,000	1.468641	1.450700	1.441124	8,497,415	8,495,496.6	8,493,866
1,000,000	1.474814	1.463953	1.454506	8,587,507	8,586,332.4	8,585,194

Для массивов фиксированной длины (10,000 элементов) результаты получились уже вот такие:

time	**ops**
0.015845	66052
0.014877	65876
0.017486	65775
…	
0.031251	65988
0.047319	65844
0.046887	65890

Анализ полученных результатов

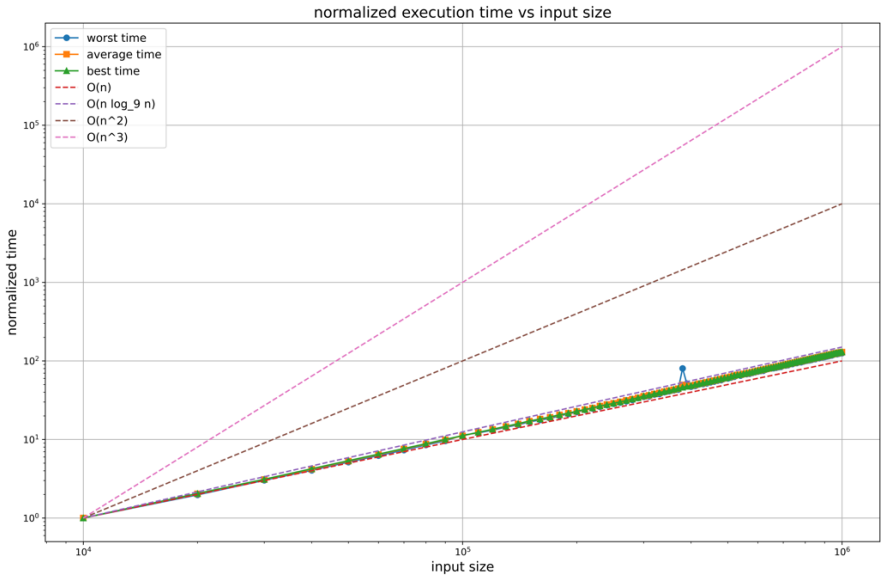
Подтверждение априорной оценки о классе сложности

Если смотреть на “сырые” данные, то уже виден почти линейный класс трудоёмкости:

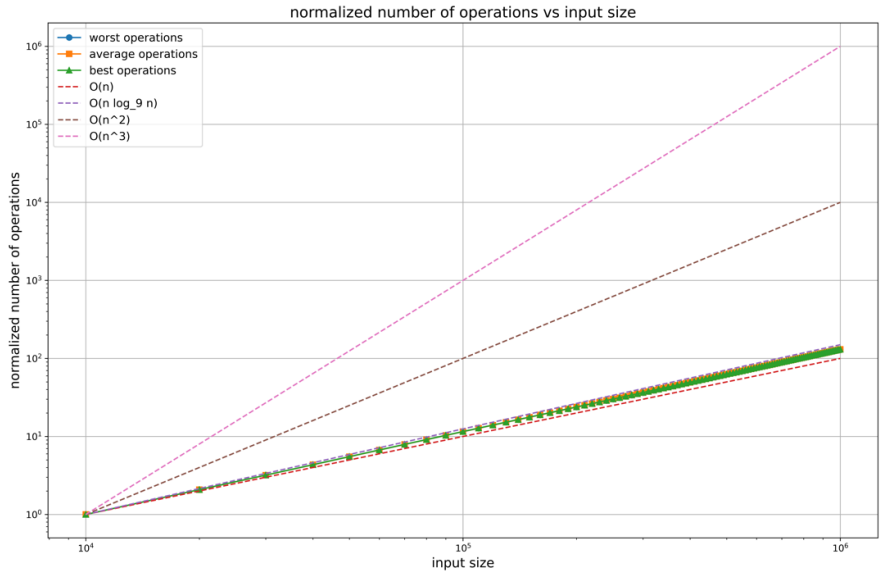
- при увеличении размера задачи с 10,000 до 20,000 (в 2 раза) среднее время увеличилось в ~2.03 раза, а среднее число операций — в 2.08 раз
- при увеличении размера задачи с 20,000 до 40,000 (в 2 раза) среднее время увеличилось в ~2.06 раз, а среднее число операций — в ~2.09 раз
- при увеличении размера задачи с 40,000 до 80,000 (в 2 раза) среднее время увеличилось в ~2.09 раз, а среднее число операций — в ~2.09 раз

То есть во всех случаях, не учитывая погрешность в полученных значениях времени/числа операций, происходит практически линейный рост: увеличили входные данные в 2 раза — трудоёмкость выросла в 2 раза с поправкой на константу.

Построим график для времени:



… и для числа операций:

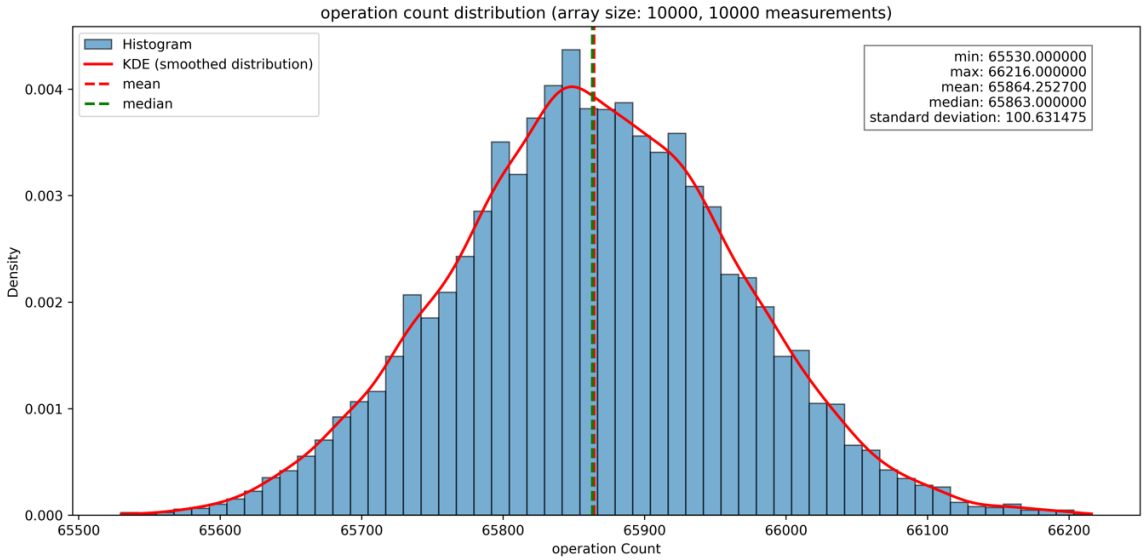


Как видно из графика, и затраченное время, и для число потраченных базовых операций лежат очень близко к графику функции $n \cdot \log_9(n)$, что подтверждает априорное утверждение о принадлежности функции трудоёмкости классу $\Omega(n \cdot \log_d(n))$.

Подтверждение априорной оценки о худшем, среднем и лучшем случаях

Так как у алгоритма есть худший, средний и лучший случаи, то можем проанализировать, как трудоёмкость алгоритма зависит от набора данных. Для этого закрепим размер массива = 10,000. И проведём 10,000 экспериментов, в каждом из которых сгенерируем свой рандомный массив и посчитаем время и число операций, ушедшее на его сортировку. Логично предположить, что редко будут реализоваться сценарии худшего случая, ведь для этого нужно, чтобы метод рандома сгенерировал много одних и тех же чисел. Так же редки будут случаи, когда у многих чисел разные разряды. В среднем в каких-то небольших выборках будут попадаться как и “удобные” для алгоритма — с разными первыми разрядами — числа, так и “неудобные” — аналогично. То есть должен получиться график, схожий с нормальным распределением.

Вот, что получилось при построении описанной гистограммы:



Как видно из графика, гипотеза о нормальном распределении функции трудоёмкости подтвердилась.

Заключение

Алгоритм поразрядной MSD сортировки является очень ценным и часто используемым инструментом для упорядочивания любых данных, где можно ввести понятие “разряда”, будь то числа, строки, даты и т.д. Проведённый априорный анализ сложности алгоритма показал, что в зависимости от структуры входных данных трудоёмкость данного алгоритма может иметь линейную, логарифмическую или комбинированную сложность, где последний вариант наиболее вероятен. Экспериментальные результаты на больших равномерно распределённых данных подтвердили теоретические оценки. Таким образом, поразрядная сортировка MSD остается эффективным решением для задач сортировки в самых различных областях — от обработки текстов до биоинформатики.

Список использованных литературных источников

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: “Introduction to Algorithms, Fourth Edition 2022”.
2. Donald E. Knuth: “The Art of Computer Pogramming, VOLUME 3, Second Edition”
3. Статья “Herman Hollerith” на сайте колумбийского университета – URL.