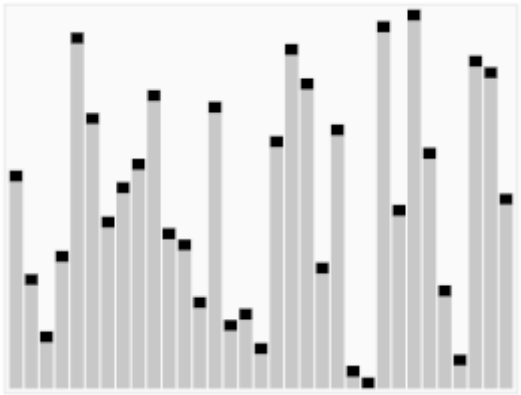
Быстрая сортировка

Быстрая сортировка, **сортировка Хоара** (англ. *quicksort*), часто называемая ***qsort*** (по имени в стандартной библиотеке языка Си) — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в МГУ в 1960 году.

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем $O(n \log n)$ обменов при упорядочении n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Быстрая сортировка



Визуализация алгоритма быстрой сортировки. Горизонтальные линии обозначают опорные элементы.

Предназначение	Алгоритм сортировки
Худшее время	O(n^2)
Лучшее время	O(n log n) (обычное разделение) или O(n) (разделение на 3 части)
Среднее время	O(n log n)
Затраты памяти	O(n) вспомогательных O(log n) вспомогательных (Седжвик 1978)

Содержание

Общее описание

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. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.

Общая идея алгоритма состоит в следующем:

- 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 (см.ниже).
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие»[1].
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения (см. ниже).

Хоар разработал этот метод применительно к машинному переводу; словарь хранился на магнитной ленте, и сортировка слов обрабатываемого текста позволяла получить их переводы за один прогон ленты, без перемотки её назад. Алгоритм был придуман Хоаром во время его пребывания в Советском Союзе, где он обучался в Московском университете компьютерному переводу и занимался разработкой русско-английского разговорника.

Алгоритм

Общий механизм сортировки

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

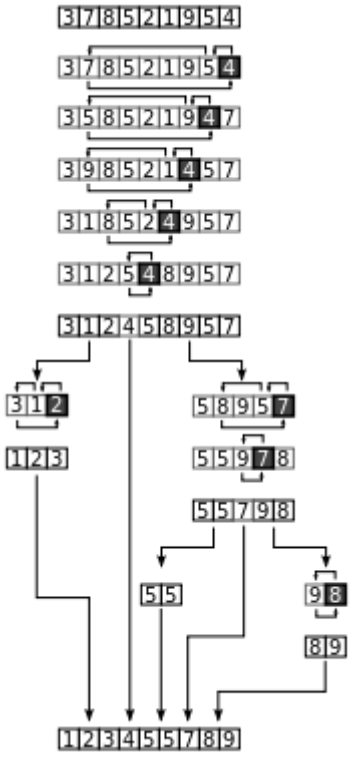
1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

В наиболее общем виде алгоритм на псевдокоде (где A — сортируемый массив, а lo и hi — соответственно, нижняя и верхняя границы сортируемого участка этого массива) выглядит следующим образом:.

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
       p := partition(A, lo, hi)
       quicksort(A, lo, p - 1)
       quicksort(A, p + 1, hi)
```

Здесь предполагается, что массив A передаётся по ссылке, то есть сортировка происходит «на том же месте», а неописанная функция partition возвращает значение опорного элемента.

Для выбора опорного элемента и операции разбиения существуют разные подходы, влияющие на производительность алгоритма.



Пример быстрой сортировки. Здесь опорным является последний элемент массива (ячейка чёрного цвета), что в отсортированных массивах может приводить к ухудшению производительности.

Возможна также следующая реализация быстрой сортировки:

```
algorithm quicksort(A) is
    if A is empty
        return A
    pivot := A.pop() (извлечь последний или первый элемент из массива)
    lA := A.filter(where e < pivot) (создать массив с элементами меньше опорного)
    rA := A.filter(where e > pivot) (создать массив с элементами больше опорного)
    return quicksort(lA) + [pivot] + quicksort(rA) (вернуть массив состоящий из отсортированной левой части,
опорного и отсортированной правой части)
```

На практике она не используется, а служит лишь в образовательных целях, так как использует дополнительную память, чего можно избежать.

Выбор опорного элемента

В ранних реализациях, как правило, опорным выбирался первый элемент, что снижало производительности на отсортированных массивах. Для улучшения эффективности может выбираться средний, случайный элемент или (для больших массивов) медиана первого, среднего и последнего элементов.[2] Медиана всей последовательности является оптимальным опорным элементом, но её вычисление слишком трудоёмко для использования в сортировке.

Выбор опорного элемента по медиане трёх для разбиения Ломуто:

```
mid := (lo + hi) / 2
if A[mid] < A[lo]
    swap A[lo] with A[mid]
if A[hi] < A[lo]
    swap A[lo] with A[hi]
if A[hi] < A[mid]
    swap A[hi] with A[mid]
swap A[hi] with A[mid]
pivot := A[hi]
```

Разбиение Ломуто

Данный алгоритм разбиения был предложен Нико Ломуто[3] и популяризован в книгах Бентли (Programming Pearls) и Кормена (Введение в алгоритмы).[4] В данном примере опорным выбирается последний элемент. Алгоритм хранит индекс в переменной `i`. Каждый раз, когда находится элемент, меньше или равный опорному, индекс увеличивается, и элемент вставляется перед опорным. Хоть эта схема разбиения проще и компактнее, чем схема Хоара, она менее эффективна и используется в обучающих материалах. Сложность данной быстрой сортировки падает до $O(n^2)$, когда массив уже отсортирован или все его элементы равны. Существуют различные методы оптимизации данной сортировки: алгоритмы выбора опорного элемента, использование сортировки вставками на маленьких массивах. В данном примере сортируются элементы массива A от `lo` до `hi` (включительно)[4]:

```
algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Сортировка всего массива может быть выполнена с помощью выполнения `quicksort(A, 1, length(A))`.

Разбиение Хоара

Данная схема использует два индекса (один в начале массива, другой в конце), которые приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после. Эти элементы меняются местами. Обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс.[5] Схема Хоара эффективнее схемы Ломуто, так как происходит в среднем в три раза меньше обменов (swap) элементов, и разбиение эффективнее, даже когда все элементы равны.[6] Подобно схеме Ломуто, данная схема также показывает эффективность в $O(n^2)$, когда входной массив уже отсортирован. Сортировка с использованием данной схемы нестабильна. Следует заметить, что конечная позиция опорного элемента необязательно совпадает с возвращённым индексом. Псевдокод[4]:

```
algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot

        do
            j := j - 1
        while A[j] > pivot

        if i >= j then
            return j

        swap A[i] with A[j]
```

Повторяющиеся элементы

Для улучшения производительности при большом количестве одинаковых элементов в массиве может быть применена процедура разбиения массива на три группы: элементы меньшие опорного, равные ему и больше него. (Бентли и Макилрой называют это «толстым разбиением». Данное разбиение используется в функции `qsort` в седьмой версии Unix[7]). Псевдокод:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := pivot(A, lo, hi)
        left, right := partition(A, p, lo, hi)  // возвращается два значения
        quicksort(A, lo, left)
        quicksort(A, right, hi)
```

Оценка сложности алгоритма

Ясно, что операция разделения массива на две части относительно опорного элемента занимает время $O(\log_2 n)$. Поскольку все операции разделения, проделываемые на одной глубине рекурсии, обрабатывают разные части исходного массива, размер которого постоянен, суммарно на каждом уровне рекурсии потребуется также $O(n)$ операций. Следовательно, 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 $C_n = 2 \cdot C_{n/2} + n$, что даёт общую сложность алгоритма $O(n \cdot \log_2 n)$.

Среднее.

Среднюю сложность при случайном распределении входных данных можно оценить лишь вероятностно.

Прежде всего необходимо заметить, что в действительности необязательно, чтобы опорный элемент всякий раз делил массив на две *одинаковых* части. Например, если на каждом этапе будет происходить разделение на массивы длиной 75 % и 25 % от исходного, глубина рекурсии будет равна $\log_{4/3} n$, а это по-прежнему даёт сложность $O(n \log n)$. Вообще, при любом *фиксированном* соотношении между левой и правой частями разделения сложность алгоритма будет той же, только с разными константами.

Будем считать «удачным» разделением такое, при котором опорный элемент окажется среди центральных 50 % элементов разделяемой части массива; ясно, вероятность удачи при случайном распределении элементов составляет 0,5. При удачном разделении размеры выделенных подмассивов составят не менее 25 % и не более 75 % от исходного. Поскольку каждый выделенный подмассив также будет иметь случайное распределение, все эти рассуждения применимы к любому этапу сортировки и любому исходному фрагменту массива.
Удачное разделение даёт глубину рекурсии не более $\log_{4/3} n$. Поскольку вероятность удачи равна 0,5, для получения k удачных разделений в среднем потребуется $2 \cdot k$ рекурсивных вызовов, чтобы опорный элемент *k* раз оказался среди центральных 50 % массива. Применяя эти соображения, можно заключить, что в среднем глубина рекурсии не превысит $2 \cdot \log_{4/3} n$, что равно $O(\log n)$ А поскольку на каждом уровне рекурсии по-прежнему 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 $\sum_{i=0}^{n}(n - i) = O(n^2)$ операций, то есть сортировка будет выполняться за квадратичное время. Но количество обменов и, соответственно, время работы — это не самый большой его недостаток. Хуже то, что в таком случае глубина рекурсии при выполнении алгоритма достигнет n, что будет означать n-кратное сохранение адреса возврата и локальных переменных процедуры разделения массивов. Для больших значений n худший случай может привести к исчерпанию памяти (переполнению стека) во время работы программы.

Достоинства и недостатки

Достоинства:

- Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
- Прост в реализации.
- Требует лишь $O(\log n)$ дополнительной памяти для своей работы. (Не улучшенный рекурсивный алгоритм в худшем случае $O(n)$ памяти)
- Хорошо сочетается с механизмами кэширования и виртуальной памяти.
- Допускает естественное распараллеливание (сортировка выделенных подмассивов в параллельно выполняющихся подпроцессах).
- Допускает эффективную модификацию для сортировки по нескольким ключам (в частности — алгоритм Седжвика для сортировки строк): благодаря тому, что в процессе разделения автоматически выделяется отрезок элементов, равных опорному, этот отрезок можно сразу же сортировать по следующему ключу.
- Работает на связных списках и других структурах с последовательным доступом, допускающих эффективный проход как от начала к концу, так и от конца к началу.

Недостатки:

- Сильно деградирует по скорости (до $O(n^2)$) в худшем или близком к нему случае, что может случиться при неудачных входных данных.
- Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека, так как в худшем случае ей может потребоваться сделать $O(n)$ вложенных рекурсивных вызовов.

- Неустойчив.

Улучшения

Улучшения алгоритма направлены, в основном, на устранение или смягчение вышеупомянутых недостатков, вследствие чего все их можно разделить на три группы: придание алгоритму устойчивости, устранение деградации производительности специальным выбором опорного элемента, и защита от переполнения стека вызовов из-за большой глубины рекурсии при неудачных входных данных.

- Проблема неустойчивости решается путём расширения ключа исходным индексом элемента в массиве. В случае равенства основных ключей сравнение производится по индексу, исключая, таким образом, возможность изменения взаимного положения равных элементов. Эта модификация не бесплатна — она требует дополнительно O(n) памяти и одного полного прохода по массиву для сохранения исходных индексов.
- Деградация по скорости в случае неудачного набора входных данных решается по двум разным направлениям: снижение вероятности возникновения худшего случая путём специального выбора опорного элемента и применение различных технических приёмов, обеспечивающих устойчивую работу на неудачных входных данных. Для первого направления:
  - Выбор среднего элемента. Устраняет деградацию для предварительно отсортированных данных, но оставляет возможность случайного появления или намеренного подбора «плохого» массива.
  - Выбор медианы из трёх элементов: первого, среднего и последнего. Снижает вероятность возникновения худшего случая, по сравнению с выбором среднего элемента.
  - Случайный выбор. Вероятность случайного возникновения худшего случая становится исчезающе малой, а намеренный подбор — практически неосуществимым. Ожидаемое время выполнения алгоритма сортировки составляет O($n \lg n$).

  Недостаток всех усложнённых методов выбора опорного элемента — дополнительные накладные расходы; впрочем, они не так велики.
- Во избежание отказа программы из-за большой глубины рекурсии могут применяться следующие методы:
  - При достижении нежелательной глубины рекурсии переходить на сортировку другими методами, не требующими рекурсии. Примером такого подхода является алгоритм Introsort или некоторые реализации быстрой сортировки в библиотеке STL. Можно заметить, что алгоритм очень хорошо подходит для такого рода модификаций, так как на каждом этапе позволяет выделить непрерывный отрезок исходного массива, предназначенный для сортировки, и то, каким методом будет отсортирован этот отрезок, никак не влияет на обработку остальных частей массива.
  - Модификация алгоритма, устраняющая одну ветвь рекурсии: вместо того, чтобы после разделения массива вызывать рекурсивно процедуру разделения для обоих найденных подмассивов, рекурсивный вызов делается только для меньшего подмассива, а больший обрабатывается в цикле в пределах этого же вызова процедуры. С точки зрения эффективности в среднем случае разницы практически нет: накладные расходы на дополнительный рекурсивный вызов и на организацию сравнения длин подмассивов и цикла — примерно одного порядка. Зато глубина рекурсии ни при каких обстоятельствах не превысит $\log_2 n$, а в худшем случае вырожденного разделения она вообще будет не более 2 — вся обработка пройдёт в цикле первого уровня рекурсии. Применение этого метода не спасёт от катастрофического падения производительности, но переполнения стека не будет.
  - Разбивать массив не на две, а на три части[8].

См. также

- Список алгоритмов сортировки

Примечания

1. Очевидно, что после такой перестановки для получения отсортированного массива не понадобится перемещать ни один из элементов между получившимися отрезками, то есть достаточно будет произвести сортировку «меньшего» и «большего» отрезков как самостоятельных массивов.

2. *Sedgewick, Robert.* Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4 (https://books.google.com/books?id=ylAETlep0CwC). — 3. — Pearson Education, 1 September 1998. — ISBN 978-81-317-1291-7.
3. *Jon Bentley.* Programming Pearls. — Addison-Wesley Professional, 1999.
4. *Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.* Quicksort // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 3-е изд. — М.: Вильямс, 2013. — С. 170–190. — ISBN 5-8459-1794-8.
5. Hoare, C. a. R. (1962-01-01). “Quicksort” (http://comjnl.oxfordjournals.org/content/5/1/10). *The Computer Journal*. **5** (1): 10—16. DOI:10.1093/comjnl/5.1.10 (https://doi.org/10.1093%2Fcomjnl%2F5.1.10). ISSN 0010-4620 (https://www.worldcat.org/issn/0010-4620).
6. Quicksort Partitioning: Hoare vs. Lomuto (http://cs.stackexchange.com/a/11550/4201). *cs.stackexchange.com*. Проверено 3 августа 2015.
7. Bentley, Jon L.; McIlroy, M. Douglas (1993). “Engineering a sort function” (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8162). *Software—Practice and Experience*. **23** (11): 1249—1265. DOI:10.1002/spe.4380231105 (https://doi.org/10.1002%2Fspe.4380231105).
8. Dual Pivot Quicksort (http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf)

Литература

- *Левитин А. В.* Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 174–179. — 576 с. — ISBN 978-5-8459-0987-9
- *Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.* Глава 7. Быстрая сортировка // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — С. 198-219. — ISBN 5-8459-0857-4.

Ссылки

- Анимированное сравнение алгоритмов сортировки (http://www.sorting-algorithms.com/quick-sort)
- Визуализаторы: [1] (http://rain.ifmo.ru/cat/view.php/vis/sorts/quicksort-2004), [2] (http://rain.ifmo.ru/cat/view.php/vis/sorts/quicksort-2000), [3] (http://prototype-nk.ru/quicksort.html)
- Динамическая визуализация 7 алгоритмов сортировки с открытым исходным кодом (https://airtucha.github.io/SortVis/)
