Быстрая сортировка

Быстрая сортировка (англ. *quick sort*, сортировка Хоара) — один из самых известных и широко используемых алгоритмов сортировки. Среднее время работы $O(n \log n)$, что является асимптотически оптимальным временем работы для алгоритма, основанного на сравнении. Хотя время работы алгоритма для массива из n элементов в худшем случае может составить $\Theta(n^2)$, на практике этот алгоритм является одним из самых быстрых.

Содержание

Алгоритм

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй".

- Массив $a[l \ldots r]$ типа T разбивается на два (возможно пустых) подмассива $a[l \ldots q]$ и $a[q+1 \ldots r]$, таких, что каждый элемент $a[l \ldots q]$ меньше или равен $a[q]$, который в свою очередь, не превышает любой элемент подмассива $a[q+1 \ldots r]$. Индекс вычисляется в ходе процедуры разбиения.
- Подмассивы $a[l \ldots q]$ и $a[q+1 \ldots r]$ сортируются с помощью рекурсивного вызова процедуры быстрой сортировки.
- Поскольку подмассивы сортируются на месте, для их объединения не требуются никакие действия: весь массив $a[l \ldots r]$ оказывается отсортированным.

Псевдокод

```
void quicksort(a: T[n], int l, int r)
   if l < r
      int q = partition(a, l, r)
      quicksort(a, l, q)
      quicksort(a, q + 1, r)
```

Для сортировки всего массива необходимо выполнить процедуру $\text{quicksort(a, 0, length[a} - 1)$.

Разбиение массива

Основной шаг алгоритма сортировки — процедура partition, которая переставляет элементы массива $a[l \ldots r]$ типа T нужным образом. Разбиение осуществляется с использованием следующей стратегии. Прежде всего, в качестве разделяющего элемента произвольно выбирается элемент $a[(l + r)/2]$. Далее начинается просмотр с левого конца массива, который продолжается до тех пор, пока не будет найден элемент, превосходящий по значению разделяющий элемент, затем выполняется просмотр, начиная с правого конца массива, который продолжается до тех пор, пока не отыскивается элемент, который по значению меньше разделяющего. Оба элемента, на которых просмотр был прерван, очевидно, находятся не на своих местах в разделенном массиве, и потому они меняются местами. Так продолжаем дальше, пока не убедимся в том, что слева от левого указателя не осталось ни одного элемента, который был бы больше по значению разделяющего, и ни одного элемента справа от правого указателя, которые были бы меньше по значению разделяющего элемента.

Переменная v сохраняет значение разделяющего элемента $a[(l + r)/2]$, а i и j представляет собой, соответственно, указатели левого и правого просмотра. Цикл разделения увеличивает значение i и уменьшает значение j на 1, причем условие, что ни один элемент слева от i не больше v и ни один элемент справа от j не меньше v, не нарушается. Как только значения указателей пересекаются, процедура разбиения завершается.

```
int partition(a: T[n], int l, int r)
   T v = a[(l + r) / 2]
   int i = l
   int j = r
   while (i ⩽ j)
      while (a[i] < v)
         i++
      while (a[j] > v)
         j--
      if (i ⩾ j)
         break
      swap(a[i++], a[j--])
   return j
```

Асимптотика

Худшее время работы

Предположим, что мы разбиваем массив так, что одна часть содержит $n - 1$ элементов, а вторая — 1. Поскольку процедура разбиения занимает время $\Theta(n)$, для времени работы $T(n)$ получаем соотношение:

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2).$$

Мы видим, что при максимально несбалансированном разбиении время работы составляет $\Theta(n^2)$. В частности, это происходит, если массив изначально отсортирован.

Способ построить массив с максимальным количеством сравнений при выборе среднего элемента в качестве опорного

В некоторых алгоритмах быстрой сортировки в качестве опорного выбирается элемент, который стоит в середине рассматриваемого массива. Рассмотрим массив, на котором быстрая сортировка с выбором среднего элемента в качестве опорного сделает $\Theta(n^2)$ сравнений. Очевидно, что это будет достигаться при худшем случае (когда при каждом разбиении в одном массиве будет оказываться 1, а в другом $n - 1$ элемент).

Заполним сначала массив a длины n элементами от 1 до n, затем применим следующий алгоритм (нумерация с нуля):

```
void antiQsort(a: T[n])
   for i = 0 to n - 1
      swap(a[i], a[i / 2])
```

Тогда на каждом шаге в качестве среднего элемента будет ставиться самый крупный элемент.

При выполнении partition делается $\Theta(n)$ сравнений из-за того, что с помощью индексов i и j мы проходим в лучшем случае $\Omega(n)$ элементов (если функция прекращает свою работу, как только индексы встречаются), в худшем случае $O(2n)$ элементов (если оба индекса полностью проходят массив). При каждом изменении индекса делается сравнение, значит, процедура partition делает $\Theta(n)$ сравнений с точностью до константы.

Рассмотрим, какой элемент будет выбираться опорным на каждом шаге. antiQsort на каждом шаге меняет местами последний и центральный элементы, поэтому в центре оказывается самый крупный элемент. А partition делает абсолютно симметричные этой процедуре операции, но в другую сторону: меняет местами центральный элемент с последним, так что самый крупный элемент становится последним, а затем выполняет на массиве длины на один меньшей ту же операцию. Получается, что опорным всегда будет выбираться самый крупный элемент, так как antiQsort на массиве любой длины будет выполнять операции, обратные partition. Фактически, partition — это antiQsort, запущенная в другую сторону. Также стоит отметить, что процедура разбиения будет делать на каждом шаге только одну смену элементов местами. Сначала i дойдет до середины массива,

до опорного элемента, j останется равным индексу последнего элемента. Затем произойдет swap и i снова начнет увеличиваться, пока не дойдет до последнего элемента, j опять не изменит свою позицию. Потом произойдет выход из while.

Разбиение массива будет произведено $\Theta(n)$ раз, потому что разбиение производится на массивы длины 1 и $n-1$ из-за того, что на каждом шаге разбиения в качестве опорного будет выбираться самый крупный элемент (оценка на худшее время работы доказана выше). Следовательно, на массиве, который строится описанным выше способом, выполняется $\Theta(n)$ partition и $\Theta(n)$ сравнений для каждого выполнения partition. Тогда быстрая сортировка выполнит $\Theta(n^2)$ сравнений для массива, построенного таким способом.

Способ построить массив с максимальным количеством сравнений при детерминированном выборе опорного элемента

Рассмотрим алгоритм построения массива, на котором быстрая сортировка с детерминированным выбором опорного элемента будет делать максимальное (в данном случае — $\Theta(n^2)$) количество сравнений. Такое число сравнений достигается при разбиении на массивы длиной 1 и $n-1$ на каждой итерации. Создадим массив a длины n, заполненный элементами типа $pair$. Такой элемент хранит пару значений (val, key), где val — элемент массива, а key — индекс. Изначально $a[i]$ элемент имеет вид $(0, i)$.

Далее, запустим для данного массива алгоритм быстрой сортировки. Сравниваем два элемента типа $pair$ по их значениям val. На каждом шаге будем выполнять следующие действия: при обращении к i-ому элементу в качестве опорного на шаге под номером k, присвоим $val = n - k + 1$ для элемента $a[i]$. Затем выполним шаг сортировки. После завершения работы алгоритма быстрой сортировки, дополнительно отсортируем получившиеся элементы $pair$ по значениям key. Искомым будет являться массив элементов val в соответствующей последовательности.

Пример для $n = 4$, при последовательном выборе опорных элементов $2, 2, 1, 1$.

Построение массива						
Шаг 1.0	**Шаг 1.1**	**Шаг 1.2**	**Шаг 2.0**	**Шаг 2.1**	**Шаг 2.2**	**Шаг 3.0**
1 2 3 4 0 **0** 0 0	1 2 3 4 0 **4** 0 0	1 4 3 2 0 0 0 **4**	1 4 3 2 0 **0** 0 4	1 4 3 2 0 **3** 0 4	1 3 4 2 0 0 **3** 4	1 3 4 2 **0** 0 3 4
Шаг 3.1	**Шаг 3.2**	**Шаг 4.0**	**Шаг 4.1**	**Шаг 4.2**	**Результат**	
1 3 4 2 **2** 0 3 4	3 1 4 2 0 **2** 3 4	3 1 4 2 **0** 2 3 4	3 1 4 2 **1** 2 3 4	3 1 4 2 **1** 2 3 4	**1 2 3 4** **2 4 1 3**	
Итоговый массив						
2 4 1 3						

Покажем, почему на данном массиве будет достигаться максимальное время работы быстрой сортировки. На этапе построения мы каждый раз присваивали опорному элементу минимальное значение. Следовательно, при выполнении quicksort алгоритм в качестве опорного всегда будет выбирать наибольший элемент массива (выборка будет производится в том же порядке ввиду детерминированности определения опорного элемента). Таким образом, так как каждый раз массив разбивается на две части — большие или равные опорному элементы и меньшие его — на каждом шаге имеем разбиение на массивы длины 1 и $n-1$, чего мы, собственно, и добивались. При таком выполнении алгоритма происходит $\Theta(n^2)$ разделений на два подмассива, и на каждом разделении выполняется $\Theta(n^2)$ сравнений. Следовательно, на данном массиве быстрая сортировка работает за $\Theta(n^2)$.

Среднее время работы

Лемма:
Время работы алгоритма быстрой сортировки равно $O(n \log n)$.
Доказательство:
▷

Пусть X — полное количество сравнений элементов с опорным за время работы сортировки. Нам необходимо вычислить полное количество сравнений. Переименуем элементы массива как $z_1 \ldots z_n$, где z_i наименьший по порядку элемент. Также введем множество $Z_{ij} = \{z_i, z_{i+1} \ldots z_j\}$.

Заметим, что сравнение каждой пары элементов происходит не больше одного раза, так как элемент сравнивается с опорным, а опорный элемент после разбиения больше не будет участвовать в сравнении.

Поскольку каждая пара элементов сравнивается не более одного раза, полное количество сравнений выражается как

$X = \sum\limits_{i=1}^{n-1} \sum\limits_{j=i+1}^{n} X_{ij}$, где $X_{ij} = 1$ если произошло сравнение z_i и z_j и $X_{ij} = 0$, если сравнения не произошло.

Применим к обеим частям равенства операцию вычисления матожидания и воспользовавшись ее линейностью получим

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ сравнивается с } z_j\}$$

Осталось вычислить величину $Pr\{z_i \text{ сравнивается с } z_j\}$ — вероятность того, что z_i сравнивается с z_j. Поскольку предполагается, что все элементы в массиве различны, то при выборе x в качестве опорного элемента впоследствии не будут сравниваться никакие z_i и z_j для которых $z_i < x < z_j$.

С другой стороны, если z_i выбран в качестве опорного, то он будет сравниваться с каждым элементом Z_{ij} кроме себя самого. Таким образом элементы z_i и z_j сравниваются тогда и только тогда когда первым в множестве Z_{ij} опорным элементом был выбран один из них.

$Pr\{z_i \text{ сравнивается с } z_j\} = Pr\{\text{первым опорным элементом был } z_i \text{ или } z_j\} = Pr\{\text{первым опорным элементом был } z_i\} + Pr\{\text{первым опорным элементом был } z_j\} =$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

$\triangleleft$

Матожидание времени работы быстрой сортировки будет $O(n \log n)$.

Модификации

Нерекурсивная реализация быстрой сортировки

Для выполнения быстрой сортировки можно воспользоваться стеком, в котором в виде сортируемых подмассивов содержится перечень действий, которые предстоит выполнить. Каждый раз когда возникает необходимость в обработке подмассива, он выталкивается из стека. После разделения массива получаются два подмассива, требующих дальнейшей обработки, которые и заталкиваются в стек. Представленная ниже нерекурсивная реализация использует стек, заменяя рекурсивные вызовы помещением в стек параметров функции, а вызовы процедур и выходы из них — циклом, который осуществляет выборку параметров из стека и их обработку, пока стек не пуст. Мы помещаем больший из двух подмассивов в стек первым с тем, чтобы максимальная глубина стека при сортировке N элементов не превосходила величины $\log n$.

```
void quicksort(a: T[n], int l, int r)
   stack< pair<int,int> > s
   s.push(l, r)
   while (s.isNotEmpty)
      (l, r) = s.pop()
      if (r ⩽ l)
         continue
      int i = partition(a, l, r)
      if (i - l > r - i)
         s.push(l, i - 1)
         s.push(i + 1, r)
      else
         s.push(i + 1, r)
         s.push(l, i - 1)
```

В качестве альтернативного варианта можно использовать обычную рекурсивную версию, в которой 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 $\log n$, а в худшем случае вырожденного разделения она вообще будет не более 1 — вся обработка пройдёт в цикле первого уровня рекурсии.

Улучшенная быстрая сортировка

Выбор медианы из первого, среднего и последнего элементов в качестве разделяющего элемента и отсечение рекурсии меньших подмассивов может привести к существенному повышению эффективности быстрой сортировки. Функция median возвращает индекс элемента, являющегося медианой трех элементов. После этого он и средний элемент массива меняются местами, при этом медиана становится разделяющим элементом. Массивы небольшого размера (длиной $M = 11$ и меньше) в процессе разделения игнорируются, затем для окончания сортировки используется сортировка вставками.

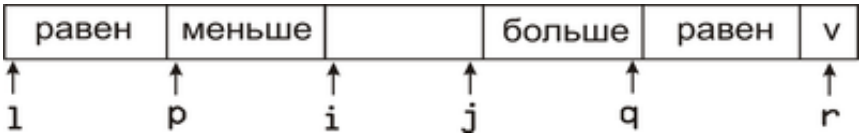
```
const int M = 10
void quicksort(a: T[n], int l, int r)
   if (r - l ⩽ M)
      insertion(a, l, r)
      return
   int med = median(a[l], a[(l + r) / 2], a[r])
   swap(a[med], a[(l + r) / 2])
   int i = partition(a, l, r)
   quicksort(a, l, i)
   quicksort(a, i + 1, r)
```

Вообще, можно применять любые эвристики по выбору опорного элемента. Например, в стандартной реализации в Java в качестве разделяющего выбирается средний из 7 элементов, равномерно распределённых по массиву.

Быстрая сортировка с разделением на три части

Когда в сортируемом массиве имеется множество повторяющихся ключей предыдущие реализации быстрой сортировки можно существенно улучшить. Например массив, который состоит из равных ключей, вовсе не нуждается в дальнейшей сортировке, однако предыдущие реализации продолжают процесс разделения, подвергая обработке все более мелкие подмассивы, независимо от того, насколько большим является исходный файл.

В основу программы положено разделение массива на три части: на элементы,меньшие разделяющего элемента $a[l] \ldots a[i]$, элементы, равные разделяющему элементу $a[i + 1] \ldots a[j - 1]$, и элементы большие разделяющего элемента $a[j] \ldots a[r]$. После этого сортировка завершается двумя рекурсивными вызовами.



Разделение массива a

Элементы массива равные разделяющему элементу находятся между l и p и между q и r. В разделяющем цикле, когда указатели просмотра перестают изменяться и выполняется обмен значениями i и j, каждый из этих элементов проверяется на предмет равенства разделяющему элементу. Если элемент, который сейчас находится слева, равен разделяющему элементу, то при помощи операции обмена он помещается в левую часть массива, если элемент, который сейчас находится справа, равен разделяющему элементу, то в результате операции обмена он помещается в правую часть массива. После того как указатели пересекутся, элементы, равные разделяющему элементу и находящиеся на разных концах массива, после операции обмена попадают в свои окончательные позиции. После этого указанные ключи могут быть исключены из подмассивов, для которых выполняются последующие рекурсивные вызовы.

```
void quicksort(a: T[n], int l, int r)
   T v = a[r]
   if (r ⩽ l)
      return
   int i = l
   int j = r - 1
   int p = l - 1
   int q = r
   while (i ⩽ j)
      while (a[i] < v)
         i++
      while (a[j] > v)
         j--
      if (i ⩾ j)
         break
      swap(a[i], a[j])
      if (a[i] == v)
         p++
         swap(a[p], a[i])
      i++
      if (a[j] == v)
         q--
         swap(a[q], a[j])
      j--
   swap(a[i], a[r])
   j = i - 1
   i++
   for (int k = l; k ⩽ p; k++, j--)
      swap(a[k], a[j])
   for (int k = r - 1; k ⩾ q; k--, i++)
      swap(a[k], a[i])
   quicksort(a, l, j)
   quicksort(a, i, r)
```

Параллельная сортировка

Еще одной оптимизацией является параллельная сортировка на основе быстрой. Пусть, исходный набор данных расположен на первом процессоре, с него начинается работа алгоритма. Затем исходный массив окажется разделенным на две части, меньшая из которых передастся другому свободному процессору, большая останется на исходном для дальнейшей обработки. Далее обе части

опять будут разделены и опять на двух исходных останутся большие части, а меньшие отправятся другим процессорам. В этом заключается ускорение алгоритма. При задействовании всех процессоров, все части параллельно будут сортироваться последовательным алгоритмом.

Introsort

Для предотвращения ухудшения времени работы быстрой сортировки до $O(n^2)$ при неудачных входных данных, также можно использовать алгоритм сортировки Introsort. Он использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов). Так как после нескольких итераций быстрой сортировки с применением разных эвристик массив с большей вероятностью окажется «почти отсортированным», то пирамидальная сортировка может довольно быстро закончить дело. Также, пирамидальная сортировка хороша тем, что требует $O(1)$ дополнительной памяти, в отличие от, например, сортировки слиянием, где потребуется $O(n)$ дополнительной памяти.

См. также

- Сортировка Шелла
- Сортировка кучей
- Сортировка слиянием
- Timsort
- Smoothsort
- PSRS-сортировка

Источники информации

- Википедия — Быстрая сортировка
- Wikipedia — Quicksort
- Wikipedia — Introsort
- *Т. Кормен, Ч. Лейзерсон, Р. Ривест*: Алгоритмы: построение и анализ глава 7
- *Р. Седжвик*: Фундаментальные алгоритмы на C++ части 1 - 4

Источник — «http://neerc.ifmo.ru/wiki/index.php?title=Быстрая_сортировка&oldid=68292»

- Эта страница последний раз была отредактирована 11 января 2019 в 03:33.