

Усовершенствованные алгоритмы сортировки

Все рассматриваемые до сих пор алгоритмы сортировки обладают очень серьезным недостатком, а именно, время их выполнения пропорционально квадрату числа элементов. Для больших объемов данных эти сортировки будут медленными, а начиная с некоторой величины они будут слишком медленными, чтобы их можно было использовать на практике. Каждый программист слышал или рассказывал "страшные" истории о "сортировке, которая выполнялась три дня". К сожалению эти истории часто не являлись выдумкой.

Если сортировка выполняется слишком долго, то причиной этого может быть используемый алгоритм. Однако, первая реакция на такое поведение сортировки выражается словами: "Давай напишем программу на ассемблере". Хотя использование ассемблера почти всегда позволяет повысить быстродействие программы в некоторое число раз с постоянным коэффициентом, но если выбранный алгоритм не является достаточно хорошим, то сортировка вне зависимости от оптимальности кода по-прежнему будет выполняться долго. Следует помнить, что при квадратичной зависимости времени выполнения программы от числа элементов массива повышение оптимальности кода или повышение быстродействия ЭВМ приведет лишь к незначительному улучшению, поскольку время выполнения программы будет увеличиваться по экспоненте. /Кривая, показанная на рис.1 лишь слегка сместится вправо, однако ее вид не изменится/. Следует помнить, что если какая-либо программа, написанная на языке Турбо Паскаль, выполняется недостаточно быстро, то она не станет выполняться достаточно быстро, если ее написать на ассемблере. Решение лежит в использовании лучшего алгоритма сортировки.

В этой главе будут рассмотрены две очень хорошие сортировки. Первая носит название сортировки Шелла, а вторая называется быстрой сортировкой, алгоритм которой признан наилучшим. Эти сортировки выполняются очень быстро.

Быстрая сортировка

Быстрая сортировка ([англ. quicksort](#)), часто называемая qsort по имени реализации в стандартной библиотеке языка [Си](#) — широко известный [алгоритм сортировки](#), разработанный английским информатиком [Чарльзом Хоаром](#) в [1960 году](#). Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем $O(n \log n)$ обменов при упорядочении n элементов), хотя и имеющий ряд недостатков.

Краткое описание алгоритма

- выбрать элемент, называемый опорным.
- сравнить все остальные элементы с опорным, на основании сравнения разбить множество на три — «меньшие опорного», «равные» и «большие», расположить их в порядке меньшие-равные-большие.
- повторить рекурсивно для «меньших» и «больших».

Примечание: на практике обычно разделяют сортируемое множество не на три, а на две части: например, «меньшие опорного» и «равные и большие». Такой подход в общем случае оказывается эффективнее, так как для осуществления такого разделения достаточно одного прохода по сортируемому множеству и однократного обмена лишь некоторых выбранных элементов.

Алгоритм

Быстрая сортировка использует стратегию «[разделяй и властвуй](#)». Шаги алгоритма таковы:

1. Выбираем в массиве некоторый элемент, который будем называть *опорным элементом*. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна [медиана](#), но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом.
2. Операция *разделения* массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него. Обычный алгоритм операции:
 1. Два индекса — l и r , приравниваются к минимальному и максимальному индексу разделяемого массива соответственно.
 2. Вычисляется индекс опорного элемента m .
 3. Индекс l последовательно увеличивается до тех пор, пока l -й элемент не превысит опорный.
 4. Индекс r последовательно уменьшается до тех пор, пока r -й элемент не окажется меньше либо равен опорному.
 5. Если $r = l$ — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент.
 6. Если $l < r$ — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r , которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r -й или l -й элемент соответственно.
3. [Рекурсивно](#) упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.
4. Базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута всегда и обработка гарантированно завершится.

Интересно, что Хоар разработал этот метод применительно к [машинному переводу](#): дело в том, что в то время словарь хранился на [магнитной ленте](#), и если упорядочить все слова в тексте, их переводы можно получить за один прогон ленты. 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. [\[1\]](#)

```
//алгоритм на языке java
public static void qSort(int[] A, int low, int high) {
    int i = low;
    int j = high;
    int x = A[(low+high)/2];
    do {
```

```

        while(A[i] < x) ++i;
        while(A[j] > x) --j;
        if(i <= j){
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
            i++; j--;
        }
    } while(i <= j);

    if(low < j) qSort(A, low, j);
    if(i < high) qSort(A, i, high);
}

```

```

//алгоритм на языке pascal
//при первом вызове 2-ой аргумент должен быть равен 0
//3-ий аргумент должен быть равен числу элементов массива минус 1
procedure qSort(var ar:array of real; low,high:integer);
var i,j:integer;
    m,wsp:real;
begin
    i:=low;
    j:=high;
    m:=ar[trunc((i+j)/2)];
    repeat
        while(ar[i]<m) do i:=i+1;
        while(ar[j]>m) do j:=j-1;
        if(i<=j) then begin
            wsp:=ar[i];
            ar[i]:=ar[j];
            ar[j]:=wsp;
            i:=i+1;
            j:=j-1;
        end;
    until (i>j);
    if(low<j) then qSort(ar,low,j);
    if(i<high) then qSort(ar,i,high);
end;

```

Оценка эффективности

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 после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате эффективный улучшенный метод.

- **Лучший случай.** Для этого алгоритма самый лучший случай — если в каждой итерации каждый из подмассивов делился бы на два равных по величине массива. В результате количество сравнений, делаемых быстрой сортировкой, было бы равно значению рекурсивного выражения $C_N = 2C_{N/2} + N$, что в явном выражении дает примерно $N \lg N$ сравнений. Это дало бы наименьшее время сортировки.
- **Среднее.** Дает в среднем $O(n \log n)$ обменов при упорядочении n элементов. В реальности именно такая ситуация обычно имеет место при случайном порядке элементов и выборе опорного элемента из середины массива либо случайно. На практике (в случае, когда обмены являются более затратной операцией, чем

сравнения) быстрая сортировка значительно быстрее, чем другие алгоритмы с оценкой $O(n \lg n)$, по причине того, что внутренний цикл алгоритма может быть эффективно реализован почти на любой архитектуре. $2C_{N/2}$ покрывает расходы по сортировке двух полученных подмассивов; N — это стоимость обработки каждого элемента, используя один или другой указатель. Известно также, что примерное значение этого выражения равно $C_N = N \lg N$.

- **Худший случай.** Худшим случаем, очевидно, будет такой, при котором на каждом этапе массив будет разделяться на вырожденный подмассив из одного опорного элемента и на подмассив из всех остальных элементов. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых.

Худший случай даёт $O(n^2)$ обменов. 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 во время работы алгоритма. Впрочем, на большинстве реальных данных можно найти решения, которые минимизируют вероятность того, что понадобится квадратичное время.

Достоинства и недостатки

Достоинства:

- Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
- Прост в реализации.
- Требуется лишь $O(\lg n)$ дополнительной памяти для своей работы. (Не улучшенный рекурсивный алгоритм в худшем случае $O(n)$ памяти)
- Хорошо сочетается с механизмами [кэширования](#) и [виртуальной памяти](#).
- Существует эффективная модификация ([алгоритм Седжвика](#)) для сортировки строк — сначала сравнение с опорным элементом только по нулевому символу строки, далее применение аналогичной сортировки для «большого» и «меньшего» массивов тоже по нулевому символу, и для «равного» массива по уже первому символу.

Недостатки:

- Сильно деградирует по скорости (до $\Theta(n^2)$) при неудачных выборах опорных элементов, что может случиться при неудачных входных данных. Этого можно избежать, используя такие модификации алгоритма, как [Introsort](#), или вероятностно, выбирая опорный элемент случайно, а не фиксированным образом.
- Наивная реализация алгоритма может привести к ошибке переполнения стека, так как ей может потребоваться сделать $O(n)$ вложенных рекурсивных вызовов. В улучшенных реализациях, в которых рекурсивный вызов происходит только для сортировки меньшей из двух частей массива, глубина рекурсии гарантированно не превысит $O(\lg n)$.
- [Неустойчив](#) — если требуется устойчивость, приходится расширять ключ.

пирамидальная сортировка

Пирамидальная сортировка ([англ. Heapsort](#), «Сортировка кучей»[\[1\]](#)) — [алгоритм сортировки](#), работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за $\Theta(n \log n)$ операций при сортировке n элементов.[\[2\]](#) Количество применяемой служебной памяти не зависит от размера массива (то есть, $O(1)$).

Может рассматриваться как усовершенствованная [сортировка пузырьком](#), в которой элемент всплывает ([min-heap](#)) / тонет ([max-heap](#)) по многим путям.

Алгоритм





Пример сортирующего дерева



структура хранения данных сортирующего дерева

Сортировка пирамидой использует [сортирующее дерево](#). Сортирующее дерево — это такое двоичное дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо , либо $d - 1$,  — максимальная глубина дерева.
2. Значение в любой вершине больше, чем значения её потомков.

Удобная структура данных для сортирующего дерева — такой массив `Array`, что `Array[1]` — элемент в корне, а потомки элемента `Array[i]` — `Array[2i]` и `Array[2i+1]`.

Алгоритм сортировки будет состоять из двух основных шагов:

1. Выстраиваем элементы массива в виде сортирующего дерева:

$$Array[i] \geq Array[2i]$$

$$Array[i] \geq Array[2i + 1]$$

при $1 \leq i < n/2$.

Этот шаг требует $O(n)$ операций.

2. Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем `Array[1]` и `Array[n]`, преобразовываем `Array[1]`, `Array[2]`, ..., `Array[n-1]` в сортирующее дерево. Затем переставляем `Array[1]` и `Array[n-1]`, преобразовываем `Array[1]`, `Array[2]`, ..., `Array[n-2]` в сортирующее

дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда $\text{Array}[1], \text{Array}[2], \dots, \text{Array}[n]$ — упорядоченная последовательность.

Этот шаг требует $O(n \log n)$ операций.

Достоинства и недостатки

Достоинства

- Имеет доказанную оценку худшего случая $O(n \log n)$.
- Требуется всего $O(1)$ дополнительной памяти (если дерево организовывать так, как показано выше).

Недостатки

- Сложен в реализации.
- Неустойчив — для обеспечения устойчивости нужно расширять ключ.
- На почти отсортированных массивах работает столь же долго, как и на хаотических данных.
- На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.

Сортировка слиянием при расходе памяти $O(n)$ быстрее ($O(n \cdot \log n)$ с меньшей константой) и не подвержена деградации на неудачных данных.

Из-за сложности алгоритма выигрыш получается только на больших n . На небольших n (до нескольких тысяч) быстрее сортировка Шелла.

Основная теорема о рекуррентных
оценках

<http://www.edu.msiu.ru/files/10985-lecture.html>