Информатика

Ворожцов Артём Викторович
Кандидат физико-математических наук,
преподаватель кафедры информатики
Московского физико-технического института (МФТИ),
тренер сборной команды МФТИ по программированию.

Алгоритм быстрой сортировки. Реализация на языке Ruby

Мы рассмотрим алгоритм быстрой сортировки и две различные реализации на языке Ruby. Этот простой алгоритм (занимает на Ruby 5 строчек) является одним из эффективных алгоритмов сортировки, его модификации представлены в различных широко используемых стандартных библиотеках. Интересно заметить, что язык Ruby достиг того уровня, когда сам код является легко читаемым текстом, написанным почти на естественном языке.

Также изучим, как на языке Ruby осуществляется массовое тестирование методов. Это совсем несложно, к тому же при использовании тестирования ценность ваших разработок на порядок повысится. Некоторые прагматичные программисты делают крайние утверждения: «Непротестированный код не является написанным».

Сразу скажем, что в языке Ruby для контейнеров (объектов классов **Array**, **Set** и **Hash**) реализован метод сортировки sort, и нет практической необходимости самому писать алгоритм сортировки.

Ниже показан пример кода, в котором конструируются объекты классов **Array**, **Set** и **Hash** и выводится результаты выполнения метода sort:

```
require 'set'
a = [7,1,5,7,13,4,4] # объект класса Array
p a.sort # => [1, 4, 4, 5, 7, 7, 13]
s = Set[7,1,5,7,13,4,4]  # объект класса Set
p s.sort # => [1, 4, 5, 7, 13]
h = {7=>"a", 1=>"b", 5=>"c", 13=>"d"} # объект класса Hash
p h.sort # => [[1, "b"], [5, "c"], [7, "a"], [13, "d"]]
```

Во всех трёх случаях результат выполнения `sort` – это массив (объект класса `Array`).

Итак, программисту Ruby (впрочем, как и программисту, использующему любой другой язык программирования, кроме языка Pascal) в принципе не нужно знать алгоритмы сортировки. Единственное, что нужно знать – это то, как пользоваться уже готовым методом `sort`, который обычно определён в стандартной библиотеке языка.

Но, тем не менее, здесь мы рассмотрим один из эффективных алгоритмов сортировки – алгоритм быстрой сортировки. Этот алгоритм несёт в себе интересные идеи, которые ценны сами по себе, как инструменты решения алгоритмических задач.

Алгоритм быстрой сортировки (quick sort)

Алгоритм quick sort является ярким примером использования идеи «разделяй и властвуй» (divide and conquer) и первым нетривиальным примером использования *рекурсии*.

Кроме того, алгоритм quick sort – это самый простой алгоритм сортировки, который работает в среднем время порядка $N \cdot \log_2 N$, где N – число элементов в сортируемом массиве.

Рассмотрим фразу «работает в среднем время порядка $N \cdot \log_2 N$». За ней скрывается довольно сложная вещь.

Во-первых, для тех, кто не знает, что такое $\log_2 N$, скажем коротко, что $\log_2 N$ примерно равно числу раз, сколько нужно делить N на 2, чтобы получить число меньше 1. Значение $\log_2 N$ медленно увеличивается с ростом N. Например, для N=1024 оно равно 10, а для числа 1 000 000 000 – примерно 30. Чтобы $\log_2 N$ увеличилось *на* 1 необходимо, чтобы N увеличилось *в* 2 раза.

Возьмём самые различные варианты входного массива; не теряя общности, можно считать, что это все возможные упорядоченные наборы из N натуральных чисел из отрезка $[1;N]$ с возможным повторением.

Применим ко всем этим наборам алгоритм сортировки quick sort и суммарное время работы разделим на число этих наборов – получим среднее время работы алгоритма сортировки на наборах чисел длины N. Обозначим это среднее как $T(N)$.

В нашем случае фраза «работает в среднем время порядка $N \cdot \log_2 N$» означает следующее:

Существуют положительные числа A и B такие, что, начиная с некоторого большого N, верно двойное неравенство

$$A \cdot N \log_2 N < T(N) < B \cdot N \log_2 N .$$

При увеличении размера массива в K раз среднее время работы алгоритма быстрой сортировки увеличится чуть больше, чем в K раз (сравните с алгоритмом сортировки методом пузырька, для которого время сортировки будет увеличиваться в K^2 раз).

Доказательство данного факта мы отложим до следующей статьи. Здесь же обратим внимание на практические аспекты – технику программирования на Ruby, тестирование.

Алгоритм quick sort сводит сортировку данного ей массива к *разделению* (partitioning) этого массива на две части и сортировке этих двух групп по отдельности.

ШАГ 1. Если размер массива 0 или 1, то закончить, вернуть массив как результат.

ШАГ 2. Взять первый элемент массива f, а остальные элементы массива разделить на две части – массив l элементов, меньших либо равных f, и массив r элементов, больших f.

ШАГ 3. Выполнить сортировку массивов l и r.

ШАГ 4. Вернуть в качестве результата массив, получающийся в результате соединения массивов l и r (путём дописывания массива r в конец массива l).

Обратите внимание на шаг 3, там происходит сортировка массивов l и r, а именно, осуществляется *рекурсивный вызов*: во время выполнения алгоритма quick sort для некоторого массива происходит запуск этого же самого алгоритма, но уже для других массивов меньшего размера.

Метод 1

Итак, давайте реализуем этот алгоритм на языке Ruby. Определим метод **qsort** для экземпляров класса **Array**.

Самым сложным, на первый взгляд, является шаг разделения массива. Но прежде, чем реализовывать метод partition, давайте заглянем в документацию языка Ruby. Есть!

Такая функция уже есть в стандартной библиотеке. Вот что написано в документации:

> Метод partition определён для экземпляров Enumerable (Enumerable#partition).
> Использование:
>
> ```
> enum.partition {| obj | block }  => [ true_array, false_array ]
> ```
>
> Возвращает два массива, первый содержит те элементы, для которых ассоциированный блок даёт значение true, а второй содержит остальные. Пример:
>
> ```
> (1..6).partition {|i| (i&1).zero?}   #=> [[2, 4, 6], [1, 3, 5]]
> ```

Что ж, это как раз то, что нам нужно. Для того чтобы разделить массив m на два массива l и r, где l содержит элементы меньшие либо равные f, а r – элементы больше f, достаточно выполнить команду

```
l,r = m.partition {|x| x <= f}
```

Давайте несколько оптимизируем алгоритм. Пусть первый элемент массива равен *f*. Тогда разделим массив на три части:

- *l* – элементы меньше *f*;
- *c* – элементы равные *f*;
- *r* – элементы больше *f*.

Операция слияния массивов в языке Ruby реализована как бинарный оператор «+». В итоге получается следующий код:

```
class Array
  def qsort
     return self.dup if size <=1
     f = self.first              # делить на две части будем
                                 # по первому элементу
     l,r = partition   {|x| x <= f}
     c,l = l.partition {|x| x == f}
     l.qsort + c + r.qsort  # слияние трёх массивов
  end
end
```

Метод 2

Удобно было бы иметь метод partition3 деления массива на 3 части, чтобы за одно действие получать три части *l*, *c* и *r*. Определим его:

```
class Array
   # ассоциированный блок должен возвращать 0,1 или 2
   # -1 соответствует 2
   # возвращает три массива
   def partition3
      a = Array.new(3) {|i| []}
      each do |x|
         a[yield(x)].push x
      end
      a
   end
   def qsort
      return self.dup if size <=1
     # c - элементы, равные первому элементу
     # l - элементы, которые меньше первого
     # r - элементы, которые больше первого
     c,l,r = partition3 {|x| first <=> x}
     l.qsort + c +  r.qsort
   end
end
```

Реализация qsort!

В языке Ruby принято определять два варианта каждого метода, преобразующего массив – метод, который создаёт новый модифициро-

ванный массив, и метод, модифицирующий непосредственно тот массив, для которого метод вызывается. Имя метода во втором случае принято заканчивать на восклицательный знак. Вот один из вариантов реализации **qsort!**:

```
class Array
   def qsort!
      self.replace(self.qsort)
   end
end
a = [1,7,6,5,4,3,2,1]
a.qsort!
p a
#
# => [1, 1, 2, 3, 4, 5, 6, 7]
#
```

Метод, создающий методы-модификаторы

Целый ряд методов в стандартных классах Ruby имеет аналоги, модифицирующие сам объект. Имена всех этих функций заканчиваются на **!**, например, **sort!**, **uniq!**, **map!.**

Их все можно было определить разом. Для этого можно определить метод, который по существующим методам **sort**, **uniq**, **map** определят методы **sort!**, **uniq!**, **map!.** Обратите внимание на саму идею – написать метод, определяющий новые методы. Такие штуки невозможны в языках подобных C/C++, Pascal или Java, а в динамических языках, подобных Ruby, Python, Lua и др., возможны и активно используются.

В языке Ruby есть метод **eval,** которому можно передать строку, содержащую корректный код на языке Ruby. Обычно многострочные объекты класса String записывают с помощью конструкции **%{ … }.**

```
def make_modifiers(*methods)
   methods.each do |method|
      eval %{
         class #{self.to_s}
            def #{method}!(*args,&block)
               self.replace(self.#{method}(*args,&block))
            end
         end
      }
   end
end
class Array
   make_modifiers :qsort, :uniq, :map
end
```

Метапрограммирование (использование методов, определяющих другие методы), в принципе, очень интересный и мощный инструмент – он позволяет

значительно упростить и оптимизировать структуру кода. Метапрограммирование активно используется в Ruby. Яркими примерами являются методы **`attr_accessor`** (метод, создающий методы для доступа к атрибутам объекта) и **`scaffold`** (метод, создающий методы контроллера в web-инструментарии rails – http://rubyonrails.org).

Быстрая сортировка массива на месте

В приведённом выше алгоритме сортировки мы на этапе разделения массива из одного массива получали три, два из которых затем по отдельности сортировали и снова соединяли в один массив. Более эффективный алгоритм использует идею сортировки массива на месте.

Пусть нам нужно отсортировать участок массива A с p-го по q-й элемент включительно, будем называть этот участок подмассивом и обозначать как $A[p..q]$.

Алгоритм сортировки подмассива $A[p..q]$:

ШАГ 1. Если размер подмассива 0 или 1, то закончить.

ШАГ 2. Возьмём элемент $A[p]$ и «раскидаем» остальные элементы $A[(p+1)..q]$ по разные стороны от его стороны: меньшие влево, большие – вправо, то есть переставим элементы подмассива $A[p..q]$ так, чтобы вначале шли элементы, меньшие либо равные $A[p]$, потом элементы, большие либо равные $A[p]$.

ШАГ 3. Пусть r есть новый индекс элемента $A[p]$. Вызовем функцию сортировки для подмассивов $A[p..(r-1)]$ и $A[(r+1)..q]$.

Это описание алгоритма на естественном языке можно записать более формально на псевдокоде:

```
QuickSort(A,p,q):
1:  if p < q
2:     then r = Partition(A, p, q)
3:          QuickSort(A,p,r-1)
4:          QuickSort(A,r+1,q)
```

```
Partition(A,p,q):
 1:  x = A[p]
 2:  i = p - 1
 3:  j = q + 1
 4:  while true # бесконечный цикл
 5:     do repeat j--  until A[j] <= x
 6:      repeat i++ until A[i] >= x
 7:      if i < j  then поменять A[i] <-> A[j]
 8:       else return j
```

И, наконец, можно создать работающий код на Ruby. Интересно заметить, что коды на Ruby не намного длиннее псевдокода и описания на естественном языке и ясно, максимально точно передают логику алгоритмов.

```
class Array
 # сортировка массива
 def qsort!
   self.qsort0!(0,size-1)
   self
 end

 # функция сортировки заданного подмассива
  def qsort0!(p,q)
   return if q-p < 1
   x = self[p]
   i = p-1
   j = q+1
   loop do
     while self[j-=1] > x do end
     while self[i+=1] < x do end
     if i < j
       self[i],self[j] = self[j],self[i]
     else
       break
     end
   end
   qsort0!(p,j)
   qsort0!(j+1,q)
   self
  end
end
```

Несложно убедиться, что на простых примерах метод работает хорошо:

```
   a=[3,6,1,5,2,4,3]; p a.qsort!
   # [1, 2, 3, 3, 4, 5, 6]
```

Но этого недостаточно. Необходимо осуществить массовое тестирование функции на больших объёмах данных и на различные крайние случаи. Например, можно сгенерировать множество случайных массивов и убедиться, что то, что выдаёт наша функция **qsort!**, совпадает с результатом стандартной функции **sort**. Для этого удобно использовать модуль test/unit:

```
class Array
 def self.create_random(sz)
   # создаёт случайный массив размера sz
   Array.new(sz) {|i| rand 100}
 end
end

require 'test/unit'
class TestQSort < Test::Unit::TestCase
```

```
# протестируем на простейшие случаи
def test_simple
  [[],[1],[1,2],[2,1],[1,1],(1..10).to_a].each do |m|
    assert_equal(m.sort,m.dup.qsort!)
  end
end
def test_random
  100.upto(200) do |sz|
    m = Array.create_random(sz)
    assert_equal(m.sort,m.dup.qsort!)
  end
end
end
```

Результат работы данной программы выглядит следующим образом:

```
Loaded suite qsort_inplace
Started
..
Finished in 0.406 seconds.

2 tests, 107 assertions, 0 failures, 0 errors
```

Итак, теперь у нас уже больше уверенности в том, что алгоритм работает верно. При сортировке одного из 107 массивов могла произойти ошибка исполнения (error) или результаты выполнения методов **`m.sort`** и **`m.dup.qsort!`** могли различаться (assertion failure). Но этого, к счастью, не произошло. Вы можете сознательно ввести в код какую-либо ошибку и посмотреть, как будут выглядеть результаты выполнения тестирования. Например, вы можете заменить **`qsort0!(p,j)`** на **`qsort0!(p,j-1)`** и повторить тестирование.

Интересно, что для тестирования с помощью модуля **`'test/unit'`** нужно лишь определить класс с произвольным именем, наследующий от класса **`Test::Unit::TestCase`** и определить в этом классе методы, начинающиеся на **`test_`**. После загрузки этого класса автоматически будет запущен цикл создания экземпляров данного класса и запуск этих методов (по одному для каждого экземпляра). Здесь используется возможность интроспекции – анализа классов и определённых методов изнутри самих классов. Подробнее об этой возможности можно прочитать в книге «Полка прагматичного программиста. Программирование на Ruby» Дейва Томаса, Энди Ханта и Чада Фоулера.