Информатика

Горшенин Юрий Васильевич
Студент Московского физико-технического института (МФТИ), II курс, факультет инноватики высоких технологий (ФИВТ).

Суффиксные массивы на Руби

Прошу Вас, ради всего святого, сначала научитесь простому.
И только потом переходите к сложному.
ЭПИКТЕТ (EPICTETUS), Беседы IV.i

Иногда бывает необходимо для данного текста быстро отвечать на многочисленные запросы – есть ли здесь заданная подстрока, или нет, и если есть, то как часто и в каких местах встречается? Когда таких запросов действительно много, хорошей идеей будет предварительная обработка текста (препроцессинг), чтобы уменьшить время каждого запроса. Для этого и были придуманы суффиксные массивы.

Немного теории

Пусть даны $T[1 \,..\, m]$ – исходный «текст» длины m, $P[1 \,..\, n]$ – та самая подстрока, которую мы ищем. Мы будем говорить, что ω – префикс, или начало строки x, если существует такая строка y, что $x = \omega y$ (конкатенация). Будем говорить, что ω – суффикс строки x, если существует такая строка y, что $x = y\omega$. Заметим, что как ω, так и y могут быть пустыми строками. Например, для строки «*abcdeab*» префиксами будут: «», «*a*», «*ab*», «*abc*», ... , «*abcdeab*». А суффиксами будут: «», «*b*», «*ab*», «*eab*», ... , «*abcdeab*». Каждый суффикс однозначно задаётся числом – позицией, с которой он начинается в строке.

В строке «*abcdeab*» суффикс «*deab*» начинается с 4-й позиции, а суффикс «*bcdeab*» со 2-й. Для удобства суффикс текста *T*, начинающийся с *s*-й позиции, будем обозначать *T*[*s* .. *m*]. Запишем все суффиксы исходного текста, начиная с 1-й позиции, заканчивая *m*-й.

Пусть *T* = «suffix». Затем отсортируем суффиксы в таблице в лексикографическом порядке. Тогда таблицы должны выглядеть следующим образом.

Было:

N	*s*	Суффикс
1	1	suffix
2	2	uffix
3	3	ffix
4	4	fix
5	5	ix
6	6	x

Стало:

N	*s*	Суффикс
1	3	ffix
2	4	fix
3	5	ix
4	1	suffix
5	2	uffix
6	6	x

Вот теперь можно записать полученные значения *s* в массив. И получится у нас массив Pos[1 .. *m*]. В данном примере массив Pos будет выглядеть так: Pos = [3, 4, 5, 1, 2, 6]. Для чего нам такое нужно? Как было сказано выше, суффикс однозначно задаётся числом – позицией, поэтому, так как нам нужны лексикографически отсортированные суффиксы, удобно результат сортировки хранить в числовом (как более экономичном) виде. Пожалуй, всё готово для поиска подстроки P. В этом нам поможет бинарный поиск.

Бинарный поиск

Суть бинарного поиска проста. Пусть нам дан отсортированный по возрастанию массив, например, чисел. И мы ищем число x. А возьмёмка мы число из серединки массива. Это будет y. Очевидно, что если $x < y$, то поиск продолжается в левой части массива. А если $x > y$, то поиск продолжается в правой части массива. И, наконец, если $x = y$, то поиск закончен.

То же самое можно применить и к лексикографически отсортированным строкам. Только сравнивать мы будем не числа.

```
BinSearch(left, right)
      middle = (left + right) / 2
      If P префикс от T[Pos[middle] .. m] Then
            Запоминаем результат
            BinSearch(left, middle - 1)
            BinSearch(middle + 1, right)
      End Else If P < T[Pos[middle] .. m] Then
            BinSearch(left, middle - 1)
      End Else BinSearch(middle + 1, right)
End
```

Разумеется, это в сильно упрощённой форме (да и вообще, это псевдокод). Зачем запоминать каждое найденное совпадение? Есть куда более рациональная идея. Найдём такие $i_{\min}$ и $i_{\max}$, что $T[\text{Pos}[i_{\min}] .. m]$ – самый левый суффикс в отсортированном массиве, префиксом которого является P. Соответственно, $T[\text{Pos}[i_{\max}] .. m]$ – самый правый суффикс там же, префиксом которого является P. Очевидно, что, так как массив Pos задаёт лексикографически отсортированные суффиксы, то для любого $i_{\min} \le i \le i_{\max}$ P будет префиксом суффикса $T[\text{Pos}[i] .. m]$. Поэтому ответом на вопрос задачи (все вхождения P в T) будет кусок массива $\text{Pos}[i_{\min} .. i_{\max}]$. Всё бы хорошо, но выполняться будет это за оценочное время $O(n \cdot \log_2 m)$. Это плохо для больших текстов. А идея оптимизации не сложна.

mlr-оптимизация

Вернёмся к тому псевдокоду выше. Что в нём не совсем хорошо? То, что P сравнивается каждый раз целиком. Введём новую функцию **lcp(str1, str2)**, которая для двух строк выдаст длину их общего префикса. Так вот, если в самом начале бинарного поиска сделать следующее:

```
mlr = min(lcp(P, T[Pos[left] .. m]),
          lcp(P, T[Pos[right] .. m]),
```

то в дальнейшем на первые mlr символов суффиксов в диапазоне от Pos[left] до Pos[right] можно «забить» и проверять только оставшиеся хвостики строк. Это даёт оценку времени работы запроса как $O(n + \log_2 m)$. А это просто великолепно!

Код на Руби

На языке Руби все эти идеи удивительно хорошо и быстро воплощаются в жизнь. Мы мало того что напишем класс **SuffixArray**, так ещё и коснёмся классики – тестировать работу будем на романе Шолохова «Тихий Дон», благо, он большой.

```
# Создаёт суффиксный массив для данного текста
# Пусть длина текста равна m, а длина искомой подстроки - n
# Тогда время препроцессинга (initialize) - O(m * m * logm)
# Время поиска (lookup) - O(n * logm), а при
# использованиии mlr-оптимизации - O(n + logm)

# Подключаем модуль для тестирования прораммы
require 'benchmark'
#
class SuffixArray
protected
```

```
# Служебные функции минимума и максимума
 def min(x, y)
   (x < y ? x : y)
 end
 def max(x, y)
   (x > y ? x : y)
 end
# А это - собственно функция lcp.
# Известно, что искать будем в строках
# text и term (аналоги T и # P), поэтому есть
# смысл задавать только позиции, с которых
# начнётся поиск длины общего префикса.
 def lcp(pos_term, pos_text)
   ret = 0
   while pos_term < @term.size && @term[pos_term] == @text[pos_text]
     ret += 1
     pos_term += 1
     pos_text += 1
   end
# Почему нет проверки pos_text < @text.size?
# При выходе мы вместо ошибки получим nil,
# который, кстати, можно без проблем сравнивать
# с чем угодно.
# Возвращаем lcp в виде счётчика ret
   ret
 end
# Та самая функция сравнения - меньше,
# больше либо равно для
# term'а и серединки. 0 - term является
#  префиксом, -1 - меньше, 1 - больше
 def cmp_prefix(pos_term, pos_text)
   shared = lcp(pos_term, pos_text)
   return 0 if shared + pos_term == @term.size
   return @term[shared + pos_term] <=> @text[shared + pos_text]
 end
# Процедура бинарного поиска.
 def bin_search_mlr(left, right, mlr)
   return if left > right
   middle = (left + right) / 2
   mlr += min(lcp(mlr, mlr + @pos[left]), lcp(mlr, mlr + @pos[right]))
   res = cmp_prefix(mlr, @pos[middle] + mlr)
   if res.zero?
     @min = middle if middle < @min
     @max = middle if middle > @max
     bin_search_mlr(left, middle - 1, mlr)
     bin_search_mlr(middle + 1, right, mlr)
   elsif res < 0
     bin_search_mlr(left, middle - 1, mlr)
```

```
    else
      bin_search_mlr(middle + 1, right, mlr)
    end
  end
public
# Собственно, весьма кратко
# записанное создание массива Pos.
  def initialize(text, wordsize)
    @text = text
# То есть на деле мы сортируем массив чисел от 0 до m - 1.
# Только в качестве сравнения - слова
# Кстати - сравнивать можно только первые wordsize символов -
# Это несколько ускоряет процесс сортировки.
# Взамен - невозможность искать термы большей длины.
    @pos = (0 ... text.size).sort_by { |i|
       text[i ... min(i + wordsize, text.size)]
    }
  end
# Lookup возвращает результат поиска -
  def lookup(term)
    @term = term
# min и max - те самые imin и imax.
# Изначально заведомо "нехорошие" значения.
    @min = @text.size
    @max = -1
    bin_search_mlr(0, @text.size - 1, 0)
# Возвращаем кусок массива Pos от imin до imax.
    @pos[@min .. @max]
  end
end

text = ""
out = []
object = nil
# Собственно, процесс тестирования
Benchmark.bm(4) do |b|
# Отдельная графа для считывания текста
  b.report("read...") do
    text = File.open("c:/code/don.txt").read
# Как вариант -
#text=Net::HTTP.get(URI.parse('http://www.lib.ru/PROZA/SHOLOHOW/tihijd#
on12.txt'))
  end
# Собственно, препроцессинг. Самая долгая часть.
  b.report("init...") do
    object = SuffixArray.new(text, 8)
  end
# Ищем слово "баркас" и печатаем -
# сколько раз оно встречается - это
# размер возвращаемого массива.
```

```
 b.report("lookup баркас ") do
   out = object.lookup("баркас")
   puts out.size
 end
# Аналогичные манипуляции со словом "война"
 b.report("lookup война ") do
   out = object.lookup("война")
   puts out.size
 end
# ... имя главного героя
 b.report("lookup Гриша ") do
   out = object.lookup("Гриша")
   puts out.size
 end
# и немного от себя
  b.report("lookup руби ") do
    out = object.lookup("руби")
    puts out.size
  end
end

Всё. Можно идти пить кофе. Результатом вывода программы будет:
           user     system      total        real
read...  0.000000   0.031000   0.031000 (  0.032000)
init... 84.454000   0.594000  85.048000 ( 85.109000)
lookup баркас 51
  0.000000   0.000000   0.000000 (  0.000000)
lookup война 63
  0.000000   0.000000   0.000000 (  0.000000)
lookup Гриша 164
  0.016000   0.000000   0.016000 (  0.016000)
lookup руби 120
  0.015000   0.000000   0.015000 (  0.016000)
```

Так мы опытным путём доказали, что Руби – очень древний язык. Первое число – как долго работала программа сама по себе. Второе – сколько работало ядро системы. Третье – арифметическая сумма первых двух. Последнее – сколько оттикало на настенных часах, то есть можно установить, как сильно отвлекался процессор от задачи выполнения нашей программы на другие задачи. Как видим, весьма быстро выполняются запросы, но зато весьма долго – препроцессинг (тест выполнялся на Celeron 2.40 GHz, 480 Мб ОЗУ). Полторы минуты для такой машины – это неэффективно. Надо улучшить результат. Действительно, быстрая сортировка – это, конечно хорошо, но только если нам мало что известно о сортируемых элементах. А это не так. Мы сортируем суффиксы, все их длины различны и меняются от 1 до m. Суть оптимизированной сортировки такова.

Отсортируем суффиксы по первой букве. Это можно сделать за время $O(m)$. Суффиксы с одинаковой первой буквой будут лежать в одной корзине. Суффикс $T[m \,..\, m]$ следует сразу поместить в начало своей кор-

зины. Пройдёмся по массиву Pos. Для Pos[i] верно, что суффикс Pos[i] – 1 будет первым в своей корзине. Ставим его на ближайшее к началу корзины место и помечаем это место как занятое. Получаем набор корзин с суффиксами, отсортированными по первым двум буквам. Дальше – проходим по Pos[i]. Уже для суффикса Pos[i] – 2 верно, что он будет на первом месте в своей корзине. Ставим его на ближайшее к началу корзины место, место помечаем как занятое. Получаем набор корзин с суффиксами, отсортированными по первым 4-м символам, и т.д. Весь процесс сортировки занимает время $O(m \cdot \log_2 m)$. Это уже приемлемо.

Ссылки

- http://ru.wikibooks.org/wiki/Ruby – книжка про Руби на русском языке.
- http://acm.mipt.ru/twiki/bin/view/Ruby/WebHome – материалы про Руби на сайте МФТИ.
- http://eigenclass.org/hiki/simple+full+text+search+engine – статья про реализацию суффиксных массивов на Руби.
- http://rain.ifmo.ru/cat/view.php/theory/unsorted/suffix-arrays-2005 – очень просто и понятно.
- Manber, Myers-Suffix arrays – A new method for on-line string searches. Книга про суффиксные массивы.

Юмор Юмор Юмор Юмор Юмор Юмор

♦ Теорема о глубине лужи

О глубине лужи нельзя судить до тех пор, пока в неё не упадёшь.

♦ I закон спора

Никогда не вступай в спор с дураком, так как люди могут не заметить разницы между вами.

♦ II закон спора

Кто громче всех кричит, тому и дают слово.