Сириус
IT-Колледж

Лекция 12. Поиск подстроки в строке

Наивный алгоритм. Z-функция, префикс функция, алгоритм Кнута-Морриса-Пратта.

Поиск подстроки в строке

Поиск подстроки в строке — одна из простейших задач поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах, языках программирования и т. п.

Поиск подстроки в строке (String searching algorithm) — класс алгоритмов над строками, которые позволяют найти паттерн (pattern) в тексте (text).

Формулировка задачи: Дан текст $t[0..n-1]$ и паттерн $p[0..m-1]$ такие, что $n \geqslant m$ и элементы этих строк — символы из конечного алфавита Σ. Требуется проверить, входит ли паттерн p в текст t.

Наивный алгоритм

Будем говорить, что паттерн p встречается в тексте t со сдвигом s, если $0 \leqslant s \leqslant n - m$ и $t[s\,..\,s + m - 1] = p$. Если строка p встречается в строке t, то p является подстрокой t.

В наивном алгоритме поиск всех допустимых сдвигов производится с помощью цикла, в котором проверяется условие $t[s\,..\,s + m - 1] = p$ для каждого из $n - m + 1$ возможных значений s.

```
def naiveStringMatcher(t, p):
    n = len(t)
    m = len(p)
    ans = []
    for i in range(0, n - m):
        if t[i : i + m - 1] == p:
            ans.push_back(i)
    return ans
```

Наивный алгоритм. Время работы

Алгоритм работает за $O(m \cdot (n - m))$. В худшем случае $m = \frac{n}{2}$, что даёт $O(\frac{n^2}{4}) = O(n^2)$. Однако если m достаточно мало по сравнению с n, то тогда асимптотика получается близкой к $O(n)$, поэтому этот алгоритм достаточно широко применяется на практике.

Наивный алгоритм. Сравнение с другими алгоритмами

Преимущества

- Требует $O(1)$ памяти.
- Приемлемое время работы на практике. Благодаря этому алгоритм применяется, например, в браузерах и текстовых редакторах (при использовании Ctrl + F), потому что обычно паттерн, который нужно найти, очень короткий по сравнению с самим текстом. Также наивный алгоритм используется в стандартных библиотеках языков высокого уровня (C++, Java), потому что он не требует дополнительной памяти.
- Простая и понятная реализация.

Недостатки

- Требует $O(m \cdot (n - m))$ операций, вследствие чего алгоритм работает медленно в случае, когда длина паттерна достаточно велика.

Сириус
IT-Колледж

Z-функция

Z-функция (Z-function) от строки S и позиции x — это длина максимального префикса подстроки, начинающейся с позиции x в строке S, который одновременно является и префиксом всей строки S. Более формально, $Z[i](s) = \max k \mid s[i \ldots i+k] = s[0 \ldots k]$.

Иными словами, $z[i]$ — это длина наибольшего общего префикса строки s и её i-го суффикса.

Значение Z-функции от первой позиции не определено, поэтому его обычно приравнивают к нулю или к длине строки.

"aaaaa" - $[0, 4, 3, 2, 1]$
"aaabaab" - $[0, 2, 1, 0, 2, 1, 0]$
"abacaba" - $[0, 0, 1, 0, 3, 0, 1]$

Z-функция. Тривиальный алгоритм

Формальное определение можно представить в виде следующей элементарной реализации за $O(n^2)$:

```
def z_func(s, n):
    z = [0] * n
    for i in range(1, n - 1):
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
    return z
```

Мы просто для каждой позиции i перебираем ответ для неё $z[i]$, начиная с нуля, и до тех пор, пока мы не обнаружим несовпадение или не дойдём до конца строки.

Разумеется, эта реализация слишком неэффективна, перейдём теперь к построению эффективного алгоритма.

Сириус
IT-Колледж

Z-функция. Эффективный алгоритм

Чтобы получить эффективный алгоритм, будем вычислять значения $z[i]$ по очереди — от $i = 1$ до $n - 1$, и при этом постараемся при вычислении очередного значения $z[i]$ максимально использовать уже вычисленные значения.

Назовём для краткости подстроку, совпадающую с префиксом строки s, отрезком совпадения. Например, значение искомой Z-функции $z[i]$ — это длина длиннейшего отрезок совпадения, начинающийся в позиции i (и заканчиваться он будет в позиции $i + z[i] - 1$).

Для этого будем поддерживать координаты $[l; r]$ самого правого отрезка совпадения, т.е. из всех обнаруженных отрезков будем хранить тот, который оканчивается правее всего. В некотором смысле, индекс r — это такая граница, до которой наша строка уже была просканирована алгоритмом, а всё остальное — пока ещё не известно.

Z-функция. Эффективный алгоритм

Тогда если текущий индекс, для которого мы хотим посчитать очередное значение Z-функции, — это i, мы имеем один из двух вариантов:

- $i > r$ — т.е. текущая позиция лежит за пределами того, что мы уже успели обработать.
  Тогда будем искать $z[i]$ тривиальным алгоритмом, т.е. просто пробуя значения $z[i] = 0$, $z[i] = 1$, и т.д. Заметим, что в итоге, если $z[i]$ окажется > 0, то мы будем обязаны обновить координаты самого правого отрезка $[l; r]$ — т.к. $i + z[i] - 1$ гарантированно окажется больше r.

Z-функция. Эффективный алгоритм

- $i \le r$ — т.е. текущая позиция лежит внутри отрезка совпадения $[l; r]$.

  Тогда мы можем использовать уже подсчитанные предыдущие значения Z-функции, чтобы проинициализировать значение $z[i]$ не нулём, а каким-то возможно бОльшим числом.

  В качестве начального приближения для $z[i]$ безопасно брать только такое выражение:

$$z_0[i] = \min(r - i + 1, z[i - l]).$$

Проинициализировав $z[i]$ таким значением $z0[i]$, мы снова дальше действуем тривиальным алгоритмом — потому что после границы r, вообще говоря, могло обнаружиться продолжение отрезка совпадение, предугадать которое одними лишь предыдущими значениями Z-функции мы не можем.

Z-функция. Эффективный алгоритм

Таким образом, весь алгоритм представляет из себя два случая, которые фактически различаются только начальным значением $z[i]$: в первом случае оно полагается равным нулю, а во втором — определяется по предыдущим значениям по указанной формуле. После этого обе ветки алгоритма сводятся к выполнению тривиального алгоритма, стартующего сразу с указанного начального значения.

Алгоритм получился весьма простым. Несмотря на то, что при каждом i в нём так или иначе выполняется тривиальный алгоритм — мы достигли существенного прогресса, получив алгоритм, работающий за линейное время.

Сириус IT-Колледж

Z-функция. Эффективный алгоритм

```python
def z_func(s):
    n = len(s)
    z = [0] * n
    l = 0
    r = 0
    for i in range(1, n):
        if i < r:
            z[i] = min(r - i, z[i - l])
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
        if i + z[i] > r:
            l = i
            r = i + z[i]
    return z
```

Поиск подстроки в строке с помощью Z-функции

n — длина текста. m — длина образца.

Образуем строку `s = pattern + # + text`, где `#` — символ, не встречающийся ни в `text`, ни в `pattern`. Вычисляем Z-функцию от этой строки. В полученном массиве, в позициях в которых значение Z-функции равно `|pattern|`, по определению начинается подстрока, совпадающая с $pattern$.

```
def substringSearch(text, pattern):
    zf = z_func(pattern + '#' + text)
    for i in range(m + 1, n + 1):
        if zf[i] == m
            return i
```

Сириус
IT-Колледж

Префикс функция

Пусть дана строка s длины n. Тогда $\pi(s)$ - это массив длины n
, i-ый элемент которого ($\pi[i]$) определяется следующим образом: это длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ полагается равным нулю.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max_{k=0 \dots i} k : s[0 \dots k-1] = s[i-k+1 \dots i].$$

Префикс функция

Например, для строки **"abcabcd"** префикс-функция равна: $[0, 0, 0, 1, 2, 3, 0]$, что означает:

```
у строки "a" нет нетривиального префикса, совпадающего с суффиксом;
у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;
у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;
у строки "abca" префикс длины 1 совпадает с суффиксом;
у строки "abcab" префикс длины 2 совпадает с суффиксом;
у строки "abcabc" префикс длины 3 совпадает с суффиксом;
у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом.
```

Другой пример — для строки **"aabaaab"** она равна: $[0, 1, 0, 1, 2, 2, 3]$.

Префикс функция. Тривиальный алгоритм

Непосредственно следуя определению, можно написать такой алгоритм вычисления префикс-функции:

```
def prefix_func(s):
    n = len(s)
    pi = [0] * n
    for i in range(n - 1):
        for k in range(1, i + 1):
            equal = True
            for j in range(k):
                if s[j] != s[i - k  + 1 + j]:
                    equal = False
                    break
            if equal:
                pi[i] = k
    return pi
```

Как нетрудно заметить, работать он будет за $O(n^3)$, что слишком медленно.

Префикс функция. Эффективный алгоритм

Сириус IT-Колледж

Для удобства будем обозначать подстроки строки s следующим образом: пусть p^k - префикс s длины k, s_i^k - подстрока длины k заканчивающаяся символом с номером i. Напомним, что первый символ строки имеет номер 0.

Будем вычислять $\pi[i]$ последовательно, начиная с $\pi[1]$. $\pi[0]$ очевидно $= 0$. Постараемся на i шаге получить решение, используя уже известную информацию, т.е. предыдущие значения π.

Будем рассматривать убывающую последовательность $k_j : p^{k_j} = s_{i-1}^{k_j}, i > k_j, k_j > k_j + 1, j = 0, 1, \ldots$, т.е. собственные суффиксы строки p^i, являющиеся одновременно ее префиксами, упорядоченные по убыванию длины. Очевидно, что первый из них, для которого выполнено $s[kj] = s[i]$ даст нам $\pi[i] = k_j + 1$.

$\pi[0] = 0$, далее, на каждом шагу алгоритма будем вычислять последовательность k_j. Если для очередного k_j выполнено $s[k_j] = s[i]$, то $\pi[i] = k_j + 1$, переходим к следующему i. Если перебрали все k_j вплоть до нуля и совпадения нет, то $\pi[i] = 0$. Заметим, что дойдя до нуля совпадение тоже нужно проверить, в этом случае можно получить $\pi[i] = 0 + 1 = 1$.

Этот алгоритм был разработан Кнутом (Knuth) и Праттом (Pratt) и независимо от них Моррисом (Morris) в 1977 г. (как основной элемент для алгоритма поиска подстроки в строке).

Легко видеть, что алгоритм имеет сложность $O(n)$: действительно, сложность шага, на котором префикс-функция возрастает, т.е. $\pi[i] = \pi[i-1] + 1$ есть $O(1)$, сложность шага на котором функция убывает есть O$(\pi[i] - \pi[i-1])$. Т.е. общая сложность есть $O(\sum_i |\pi[i] - \pi[i-1]|)$. Сумма положительных приростов префикс-функции не превышает n. А сумма отрицательных изменений не может превысить сумму положительных (иначе мы уйдем в минус). Значит сумма модулей изменений функции не превысит $2n$, значит общая сложность $O(n)$.

Как нетрудно заметить, этот алгоритм является онлайновым, т.е. он обрабатывает данные по ходу поступления — можно, например, считывать строку по одному символу и сразу обрабатывать этот символ, находя ответ для очередной позиции. Алгоритм требует хранения самой строки и предыдущих вычисленных значений префикс-функции, однако, как нетрудно заметить, если нам заранее известно максимальное значение, которое может принимать префикс-функция на всей строке, то достаточно будет хранить лишь на единицу большее количество первых символов строки и значений префикс-функции.

Сириус IT-Колледж

Алгоритм Кнута-Морриса-Пратта

Дан текст t и строка s, требуется найти и вывести позиции всех вхождений строки s в текст t.

Обозначим для удобства через n длину строки s, а через m — длину текста t.

Образуем строку `s+#+t`, где символ `#` — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Теперь рассмотрим её значения, кроме первых $n+1$ (которые, как видно, относятся к строке s и разделителю). По определению, значение $\pi[i]$ показывает наидлиннейшую длину подстроки, оканчивающейся в позиции i и совпадающего с префиксом. Но в нашем случае это $\pi[i]$ — фактически длина наибольшего блока совпадения со строкой s и оканчивающегося в позиции i. Больше, чем n, эта длина быть не может, за счёт разделителя. А вот равенство $\pi[i] = n$ (там, где оно достигается), означает, что в позиции i оканчивается искомое вхождение строки s (только не надо забывать, что все позиции отсчитываются в склеенной строке `s+#+t`).

Таким образом, если в какой-то позиции i оказалось $\pi[i] = n$, то в позиции $i-(n+1)-n+1 = i-2n$ строки t начинается очередное вхождение строки s в строку t.

Как уже упоминалось при описании алгоритма вычисления префикс-функции, если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. В нашем случае это означает, что нужно хранить в памяти лишь строку `s+#` и значение префикс-функции на ней, а потом уже считывать по одному символу строку t и пересчитывать текущее значение префикс-функции.

Алгоритм Кнута-Морриса-Пратта решает эту задачу за $O(n+m)$ времени и $O(n)$ памяти.

Сириус IT-Колледж

Алгоритм Кнута-Морриса-Пратта

```
def kmp(s, t):
    n = len(s)
    m = len(t)
    answer = []
    p = prefix_func(s + "#" + t)
    count = 0
    for i in range(0, m - 1)
        if p[n + i + 1] == n:
            count += 1
            answer[count] = i - n
    return answer
```