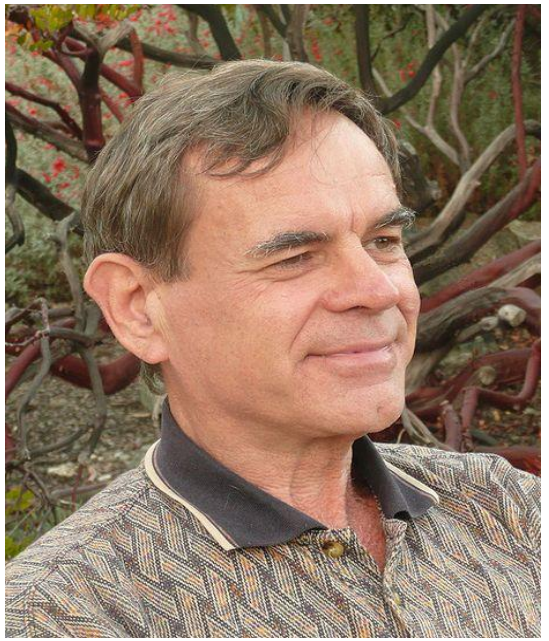
Префикс-функция.
Алгоритм Кнута-Морриса-Пратта.



Префикс-функция. Определение

Дана строка s [0 .. n – 1]. Требуется вычислить для неё префикс-функцию, то есть массив чисел prefix [0 .. n – 1], где prefix[i] определяется следующим образом: это такая длина наибольшего собственного суффикса подстроки s[0 .. i], совпадающего с её префиксом (собственный суффикс – значит не совпадающий со всей строкой). В частности, значение prefix[0] = 0.

Математически определение префикс-функции можно записать следующим образом:

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

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].

Тривиальный алгоритм

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

```
void prefix_function (char string[], int n, int prefix[])
{
    for (int i = 0; i < n; ++i)
    {
        for (int k = 0; k <= i; ++k)
        {
            char firstSubstring[n];
            char secondSubstring[n];
            haveSubstring(firstSubstring, string, 0, k, n);
            haveSubstring(secondSubstring, string, i - k + 1, k, n);
            if (isEqualSubstrings(firstSubstring, secondSubstring, n))
            {
                prefix[i] = k;
            }
        }
    }
}
```

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

Эффективный алгоритм

Первая оптимизация

Первое важное замечание – что значение prefix[i + 1] не более чем на единицу превосходит значение prefix[i] для любого i. Действительно, в противном случае, если бы prefix[i + 1] > prefix[i] + 1, то рассмотрим это суффикс, оканчивающийся на позиции i + 1 и имеющий длину prefix[i + 1] – удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину prefix[i + 1] - 1, что лучше prefix[i] – противоречие.

Таким образом, при переходе к следующей позиции очередной элемент префикс-функции мог либо увеличиться на единицу, либо не измениться, либо уменьшиться на какую либо величину. Уже это факт, позволяет сократить асимптотическое время работы алгоритма до $O(n^2)$ - поскольку за один шаг значение могло вырасти максимум на единицу, то суммарно для всей строки могло произойти не более n увеличений на единицу, и, как следствие, не более n уменьшений.

Вторая оптимизация

Пойдем дальше – избавимся от явных сравнений подстрок. Для этого постараемся максимально использовать информацию, посчитанную на предыдущих шагах.
Пусть мы вычислили значение префикс-функции prefix[i] для некоторого i.
Теперь, если s[i + 1] = s[prefix[i]], то с уверенностью можно утверждать, что prefix[i + 1] = prefix[i] + 1.

Пусть теперь, наоборот, оказалось, что string[i + 1] != string[prefix[i]]. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) позиции j < prefix[i], что по-прежнему выполняется префикс-свойство в позиции i, то есть string[0 .. j – 1] == string[i – j + 1 .. j].
Действительно, когда мы найдем такую длину j, то нам снова достаточно сравнить символы string[i + 1] и string[j] – если они совпадут, то утверждать, что
prefix[i + 1] == j + 1. Иначе нам надо будет снова найти наименьшее значение j, для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся – это происходит, когда j = 0. В этом случае, если string[i + 1] = string[0], то prefix[i + 1] = 1, иначе prefix[i + 1] = 0.

Итак, общая схема алгоритма у нас есть, нерешенным остался вопрос об эффективном нахождении таких длин j. Поставим этот вопрос формально: по текущей длине j и позиции i (для которых выполняется префикс-свойство) требуется найти наибольшее k < j, для которого по прежнему выполняется префикс-свойство.

После столь подробного описания уже становится очевидным тот факт, что это значение k есть не что иное, как значение префикс-функции prefix[j – 1], которое уже посчитано ранее. Таким образом, находить эти длины мы можем за $O(1)$ каждую.

Итоговый алгоритм

Мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет $O(n)$ действий.

Приведем здесь итоговую схему алгоритма:

- Считать значения префикс-функции prefix[i] будем по очереди: от i = 1 к i = n – 1 (prefix[0] = 0 – база).
- Для подсчета текущего prefix[i] заводим переменную j, обозначающую длину текущего рассматриваемого образца. Изначально j = prefix[i – 1].
- Проверяем образец длины j, для чего сравниваем символы string[i] и string[j]. Если они совпадают – то prefix[i] = j + 1 и переходим к индексу i + 1, иначе уменьшаем длину j, полагая ее равной prefix[j – 1], повторяем этот шаг алгоритма снова.
- Если дошли до длины j = 0 и не нашли совпадения, то prefix[i] = 0, после чего переходим к следующему индексу (i + 1).

Реализация

В итоге алгоритм получился весьма простым и красивым:

```
void prefix_function (char string[], int n, int prefix[])
{
    prefix[0] = 0;
    for (int i = 1; i < n; ++i)
    {
        int j = prefix[i - 1];
        while (j > 0 && string[j] != string[i])
        {
            j = prefix[j - 1];
        }
        if (string[i] == string[j])
        {
            ++j;
        }
        prefix[i] = j;
    }
}
```

Полезные ссылки

- http://e-maxx.ru/algo/prefix_function
- https://ru.wikipedia.org/wiki/Алгоритм_Кнута_—_Морриса_—_Пратта
- http://habrahabr.ru/post/191454/

Спасибо за внимание!