**Поиск подстроки в строке. Конечный детерминированный автомат**

**Префикс-функция**

Определение 1. *Префиксом строки S называется подстрока вида* S[1 .. i], 1 ≤ i ≤ n, *где п — это длина строки S.*

Определение 2. *Суффиксом строки S называется подстрока вида* S [i.. n], 1 ≤ i ≤ n, *где п* — *это длина строки S*.

Например, для строки *Windows ХР* строки *W* и *Win* являются префиксами, а строка *ows ХР —* суффиксом. Интересна строка *колокол*, для которой трехбуквенный префикс *кол* одновременно является и трехбуквенным суффиксом.

Определение 3. *Префикс-функцией от строки S называется п-мерный вектор, i -я компонента которого* — *это длина наибольшего префикса строки* S[1 .. i], *который не совпадает с самой строкой* S[1 .. i] *и одновременно является ее суффиксом. Если такого префикса не существует, то i -я компонента равна нулю. Здесь п — это длина строки S.*

Префикс-функцию от строки *S* будем обозначать *π(S)*, а i-ю ее компоненту — π(S,i).

Из определения следует, что для любой строки *S* справедливо π(S,1)= 0.

Найдем префикс-функцию для строки *S* = *abcadabccabca.*

π(S,1)= 0. Для строки S[1] *= а* не существует префикса, не совпадающего с самой строкой S[1].

π(S,2)= 0. Для строки S[1.. 2] = *аb* не существует префикса, который не совпадает с самой строкой S[1..2] и одновременно является ее суффиксом.

π(S,3)= 0. Для строки S[1..3] = *аbс* не существует префикса, который не совпадает с самой строкой S[1 ..3] и одновременно является ее суффиксом.

π(S,4)= 1. Для строки S[1.. 4] = *аbса* существует префикс *а*, который не совпадает с самой строкой S[1.. 4] и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 1.

π(S,8)= 3. Для строки S[1 .. 8] = *abcadabc* существует префикс *abc*, который не совпадает с самой строкой S[1.. 8] и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 3.

π(S,13)= 4. Для строки S[1..13] = *abcadabccabca* существует префикс *abca*, который не совпадает с самой строкой S[1 ..13] и одновременно является ее суффиксом. Он является максимальным, длина этого префикса равна 4.

*π(S)*= { 0 0 0 1 0 1 2 3 0 1 2 3 4 } .

Найдем префикс-функцию для строки

*S = abcabdxacabcabca.*

Для удобства восприятия составим таблицу:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| S | a | b | c | a | b | d | x | a | c | a | b | c | a | b | c | a |
| π(S,i) | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 3 | 4 |

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

vector<int> prefix\_function (string s) {

int n = (int) s.length();

vector<int> pi (n);

for (int i=0; i<n; ++i)

for (int k=0; k<=i; ++k)

if (s.substr(0,k) == s.substr(i-k+1,k))

pi[i] = k;

return pi;

}

Разберем алгоритм, строящий префикс-функцию за линейное от длины строки время.

Дана строка . Требуется вычислить для неё префикс-функцию,



т.е. массив чисел

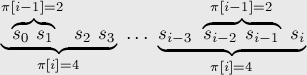


**Вносятся несколько важных замечаний**:

Первое важное замечание — что значение  не более чем на единицу превосходит значение  для любого .



Действительно, в противном случае, если бы , то рассмотрим этот суффикс, оканчивающийся в позиции  и имеющий длину  — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции  и имеющий длину , что лучше , т.е. пришли к противоречию. Иллюстрация этого противоречия (в этом примере  должно быть равно 3):

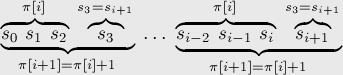


(на этой схеме верхние фигурные скобки обозначают две одинаковые подстроки длины 2, нижние фигурные скобки — две одинаковые подстроки длины 4)

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

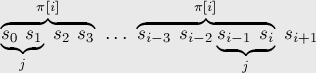
Пойдём дальше — **избавимся от явных сравнений подстрок**. Для этого постараемся максимально использовать информацию, вычисленную на предыдущих шагах.

Итак, пусть мы вычислили значение префикс-функции  для некоторого . Теперь, если , то мы можем с уверенностью сказать, что , это иллюстрирует схема:



(на этой схеме снова одинаковые фигурные скобки обозначают одинаковые подстроки)

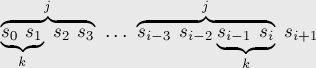
Пусть теперь, наоборот, оказалось, что . Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине , что по-прежнему выполняется префикс-свойство в позиции , т.е. :



Действительно, когда мы найдём такую длину , то нам будет снова достаточно сравнить символы  и  — если они совпадут, то можно утверждать, что . Иначе нам надо будет снова найти меньшее (следующее по величине) значение , для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения  кончатся — это происходит, когда . В этом случае, если , то , иначе .



Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин . Поставим этот вопрос формально: по текущей длине  и позиции  (для которых выполняется префикс-свойство, т.е. ) требуется найти наибольшее , для которого по-прежнему выполняется префикс-свойство:



По определению префикс-функции для строки префикс является суффиксом, т.е. . Продолжим .



После столь подробного описания уже практически напрашивается, что это значение  есть не что иное, как значение префикс-функции , которое уже было вычислено нами ранее (вычитание единицы появляется из-за 0-индексации строк).



**Итоговый алгоритм**

Итак, мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет  действий.



Приведём здесь итоговую схему алгоритма:

* Считать значения префикс-функции  будем по очереди: от  к  (значение  просто присвоим равным нулю).



* Для подсчёта текущего значения  мы заводим переменную , обозначающую длину текущего рассматриваемого образца. Изначально .



* Тестируем образец длины , для чего сравниваем символы  и . Если они совпадают — то полагаем  и переходим к следующему индексу . Если же символы отличаются, то уменьшаем длину , полагая её равной , и повторяем этот шаг алгоритма с начала.



* Если мы дошли до длины  и так и не нашли совпадения, то останавливаем процесс перебора образцов и полагаем  и переходим к следующему индексу .



Представим программу, которая строит префикс-функцию с учетом того, что в JScript строки нумеруются с нуля.

.

t=WScript.StdIn.ReadLine()

//t*=’*qqwqqverqqwqqwetqqwqqwerqqwqqwerqqwqqq’,

m=t.length

pi=new Array()

pi[0]=0

k=0

for(i=1;i<m;i++) {

while((k>0) *&&* (t.charAt(k)!=t.charAt(i)))

k=pi[k-1];

if (t.charAt(k)== t.charAt(i))

k++

pi[i]=k

}

WScript.Echo(pi)

Во второй строке в комментарии приведен хороший тестовый пример.

**Применения**

Поиск подстроки в строке.

Задача поиска подстроки *Т* в строке *S.*

Обозначим через n длину строки S, а через m— длину подстроки T.

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

Таким образом, если в какой-то позиции i оказалось *π(i)*=m, то в позиции i-(m+1)-m+1= =i-2m строки S начинается очередное вхождение подстроки T в строку S.

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 T+& и значение префикс-функции на ней, а потом уже считывать по одному символу строку S и пересчитывать текущее значение префикс-функции.

**Алгоритм Морриса—Пратта**

Вернемся к задаче поиска подстроки *Т* в строке *S.* Пусть поиск начинается с i-й позиции строки S, 1 ≤ i ≤ n-m+1, т. е. мы сопоставляем Т[1] и S[i]. Пусть первое несоответствие произошло на (k+1)-й, 1 < k+1 ≤ m , позиции шаблона Т, тогда S [i.. i+k-1] = Т[1.. k]*.* При сдвиге шаблона *Т* можно ожидать, что префикс фрагмента Т[1 .. k]совпадет с соответствующим суффиксом фрагмента S [і.. i+k-1]. Тогда сдвинуться можно будет не на один символ, а на ***k-* π(T,k)** и продолжить сопоставление с той позиции, на которой было обнаружено рассогласование.

Рассмотрим пример поиска подстроки *Т = abaxabaz* в длинной строке S. На схеме будет приведен лишь ее фрагмент S[100.. 113].

1. Построим префикс-функцию для Т.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| T | a | b | a | x | a | b | a | z |
| π(T,i) | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 0 |

2. Пусть поиск начался с 100-й позиции строки S, несоответствие произошло на 8-м символе Т, таким образом, *і* = 100, *k* = 7.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
| S | a | b | a | x | a | b | a | b | a | x | a | b | a | z |
| T | a | b | a | x | a | b | a | z |  |  |  |  |  |  |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |  |  |  |  |  |

Префикс фрагмента *Т*[1..7] совпадает с суффиксом фрагмента S[100.. 106], а именно Т[1..3] = S[104..106]. Надо сдвинуть шаблон на 4 позиции.

Теперь мы готовы ответить на вопрос об автоматическом выборе величины сдвига! Величина сдвига равна *k-* π(T,k); в данном случае *k-* π(T,k)= 7-3, т. е. 4 позиции.

Делаем новую попытку сопоставления. Поиск начался со 104-й позиции строки S (в прошлый раз начинали с 100-й и на 4 сдвинулись), несоответствие произошло на 4-м символе Т, таким образом, *і =* 104, *k* = 3.

Важно отметить, что сопоставление S и *Т* началось со сравнения S[107] и Т[4]; то, что Т[1..3] = S[104.. 106], известно с предыдущего шага.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
| S | a | b | a | x | a | b | a | b | a | x | a | b | a | z |
| T |  |  |  |  | a | b | a | x | a | b | a | z |  |  |
| j |  |  |  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |  |

Не задумываясь над тем, что префикс фрагмента Т[1 ..3] совпадает с суффиксом фрагмента S[104.. 106], сразу же определяем величину сдвига по формуле *k-* π(T,k)*.* Таким образом, *k-* π(T,k)= 3-1= 2.

Делаем новую попытку сопоставления. Поиск начался со 106-й позиции строки S, таким образом, *і =* 106. Сопоставление *S* и *Т* началось со сравнения S[107] и Т[2]; то, что Т[1]=S[106], известно с предыдущего шага. Посимвольная проверка показывает совпадение.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 |
| S | a | b | a | x | a | b | a | b | a | x | a | b | a | z |
| T |  |  |  |  |  |  | a | b | a | x | a | b | a | z |
| j |  |  |  |  |  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Описание алгоритма

1. Построить префикс-функцию образца T, обозначим ее pi.
2. Положить k = 0, i = 0.
3. Сравнить символы T[k] и S[i].
   1. Если символы равны, увеличить k на 1. Если при этом k стало равно длине образца, то вхождение образца T в строку S найдено, индекс вхождения равен i – |T| + 1, продолжаем поиск следующего вхождения, присвоим k = pi[k–1].
   2. Если символы не равны, используем префикс-функцию для оптимизации сдвигов. Пока k > 0, присвоим k = pi[k–1] и перейдем в начало шага 3.
4. Пока i < |S|, увеличиваем i на 1 и переходим в шаг 3.

**Z-функция строки и её вычисление**

Пусть дана строка T длины m. Тогда **Z-функция** ("зет-функция") от этой строки — это массив длины m, i-ый элемент которого равен наибольшему числу символов, начиная с позиции i, совпадающих с первыми символами строки T.

Иными словами, Z[i] — это наибольший общий префикс строки T и её i-го суффикса.

**(** считаем первый символ строки имеет индекс 0, а последний — m-1).

Первый элемент Z-функции, Z[0], обычно считают неопределённым. Будем считать, что он равен нулю (хотя ни в алгоритме, ни в приведённой реализации это ничего не меняет).

**Примеры**

Приведём для примера подсчитанную Z-функцию для нескольких строк:

* :



* :



* :



**Эффективный алгоритм вычисления Z-функции**

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

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

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

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

* **i>right** — 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 , то мы будем обязаны обновить координаты самого правого отрезка  **[left;right]**—



т.к. i+Z[i]-1 гарантированно окажется больше right.

* **i<=right**   — т.е. текущая позиция лежит внутри отрезка совпадения **[left;right]**.

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

Для этого заметим, что подстроки T[left..right] и T[0..right-left]  **совпадают**. Это означает, что в качестве начального приближения для Z[i] можно взять соответствующее ему значение из отрезка T[0..right-left] , а именно, значение Z[i-left].

Однако значение Z[i-left]  могло оказаться слишком большим: таким, что при применении его к позиции i оно "вылезет" за пределы границы right. Этого допустить нельзя, т.к. про символы правее right  ничего не знаем, и они могут отличаться от требуемых.

Приведём **пример** такой ситуации, на примере строки:



Когда мы дойдём до последней позиции (i=6), текущим самым правым отрезком будет [5;6]. Позиции 6 с учётом этого отрезка будет соответствовать позиция 6-5=1, ответ в которой равен Z[1]=3. Очевидно, что таким значением инициализировать Z[6]  нельзя, оно совершенно некорректно. Максимум, каким значением могли проинициализировать — это 1, поскольку это наибольшее значение, которое не вылазит за пределы отрезка [left;right] .

Таким образом, в качестве **начального приближения** для Z[i] безопасно брать только такое выражение:

Z[i]=min(right-i+1, Z[i-left]).

Проинициализировав Z[i] таким значением, дальше действуем **тривиальным алгоритмом** — потому что после границы right, вообще говоря, могло обнаружиться продолжение отрезка совпадение, предугадать которое одними лишь предыдущими значениями Z-функции мы не могли.

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.

**Реализация**

m=T.length

z=new Array()

left = 0

right = 0;

for (var j = 0; j <m; j++) {

z[j]=0

}

for (var j = 1; j < m; j++) {

if (j <= right) z[j] = Math.min(right - j + 1, z[j - left]);

while (j + z[j] < m && T.charAt(z[j]) == T.charAt(j + z[j])) z[j]++;

if (j + z[j] - 1 > right) {

left = j;

right = j + z[j] - 1;

}

}

Поиск подстроки в строке.

Задача поиска подстроки *Т* в строке *S.*

Обозначим через n длину строки S, а через m— длину подстроки T.

Образуем строку T+&+S, где символ &— это разделитель, который не должен нигде более встречаться.

Посчитаем для полученной строки Z-функцию. Тогда для любого i в отрезке [0;n-1] по соответствующему значению Z[i+m+1] можно понять, входит ли образец T в текст S, начиная с позиции i: если это значение Z-функции равно m, то да, входит, иначе — нет.

**Переход между Z- и префикс- функциями.**

Пускай Z-функция уже посчитана и хранится в массиве *Z*[0..*m*-1]. Будем идти в цикле, перебирая все номера i ячейки в массиве Z. Обратим внимание, что в силу определений как z-, так и префикс-функций, если элемент *Z*[*i*] не равен единице, то для всех элементов с индексом *i* + *j*, где 0 < *j* < *Z*[*i*] значение *P*[*i* + *j*] будет не меньше, чем *j* + 1. Несложно заметить и то, что это значение будет больше тогда и только тогда, если оно уже было установлено как более высокое когда мы исследовали меньший индекс *i*. Отсюда получаем алгоритм перехода от Z-функции к prefix- функции. Перебираем в цикле индекс элемента в z-функции *i*. Если *Z*[*i*] не равно нулю, то устанавливаем значение *P*[*i* + *Z*[*i*] - 1] равным *Z*[*i*]. После этого в цикле перебираем все номера элементов сверху вниз от *i* + *Z*[*i*] - 1 до *i*, но преждевременно прерываем наш проход, если значение P в этой точке уже подсчитано (это и обеспечивает нам линейную скорость работы, т.к. на таких условиях каждая ячейка *P*[*i*] будет просмотрено не более двух раз).

for(i = 1; i < m; i++)

if(z[i]>0)

for(j = z[i] - 1; j >= 0; j--)

if (p[i + j]==0)

p[i + j] = j + 1;

**Построение и использование конечного детерминированного автомата.**

Конечным детерминированным автоматом *А* называется пятерка < Q, Σ, *qo*, *F*, *δ* >, где

*Q* — конечное множество состояний автомата;

Σ — конечный алфавит, т. е. множество букв, которое может читать автомат;

*qo —* начальное состояние автомата, *qo* ∊ *Q;*

*F* — множество терминальных состояний *F* С Q;

*δ* — функция перехода, т. е. отображение вида *Q* х Σ *—> Q;*

Полезной является графовая интерпретация автомата. Состояния изображаются кругами — это вершины графа. Вершины соединены дугами; на дугах пишутся буквы в соответствии с функцией перехода *δ.* Именно вершины *qj* ∊ *Q* соединены дутой, ведущей от *qi* к *qj,* а на дуге приписана буква *а* ∊ Σ, если *δ(qi, а) = qj.*

Имея строку *Т* и зная алфавит строки S, можно построить специальный конечный детерминированный автомат, который способен распознать вхождение подстроки *Т* в строку *S.* Состояниями *Q* будут всевозможные префиксы строки *Т* и состояние λ, которое соответствует пустому префиксу — пустой строке. Алфавит Σ состоит из объединения алфавитов строк *Т* и *S.* Начальным состоянием является λ, терминальным — состояние, соответствующее префиксу, совпадающему со всей строкой Т. Переход из состояния *q*∊*Q* по букве *а* ∊Σ осуществляется по правилу *,* где *d —* максимальный суффикс строки *qa,* являющийся префиксом строки *Т.* В частности, возможно, что *d* = *qa.*

Автомат, распознающий вхождение подстроки в строку, можно хранить в виде таблицы переходов — двумерного массива. Специальным символом \* будем обозначать любой символ строки S, не принадлежащий строке *Т.* Использование символа \* позволяет заполнить таблицу переходов без априорного знания алфавита строки *S*

и, кроме того, зачастую сократить объем таблицы переходов.

Например, для шаблона *Т* = *аbс* имеем

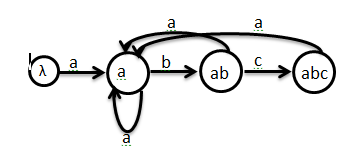
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | \* |
| λ | a | λ | λ | λ |
| a | a | ab | λ | λ |
| ab | a | λ | abc | λ |
| abc | a | λ | λ | λ |

Первая строка: λa=a, a является префиксом строки *Т=abc; λb=b, b не* является префиксом строки *Т=abc; λc=c, c не* является префиксом строки *Т=abc; λ\*=\*, \* не* является префиксом строки *Т=abc.*

Вторая строка: aa, a является префиксом строки *Т=abc; ab, ab* является префиксом строки *Т=abc; ac, ac не* является префиксом строки *Т=abc; a\*, a\* не* является префиксом строки *Т=abc.*

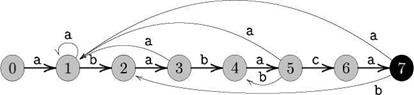
Третья строка: aba, a является префиксом строки *Т=abc; abb, b не* является префиксом строки *Т=abc; abc, abc* является префиксом строки *Т=abc; ab\*, \* не* является префиксом строки *Т=abc.*

Четвертая строка: abсa, a является префиксом строки *Т=abc; abсb, b не* является префиксом строки *Т=abc; abсc, c не* является префиксом строки *Т=abc; abс\*, \* не* является префиксом строки *Т=abc.*



Для шаблона *Т* = *аbabaсa:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | a | b | c | \* |
| 0 | λ | 1 (a) | 0 (λ) | 0 (λ) | 0 |
| 1 | a | 1 (a) | 2 (ab) | 0 (λ) | 0 |
| 2 | ab | 3 (aba) | 0 (λ) | 0 (λ) | 0 |
| 3 | aba | 1 (a) | 4 (abab) | 0 (λ) | 0 |
| 4 | abab | 5 (ababa) | 0 (λ) | 0 (λ) | 0 |
| 5 | ababa | 1 (a) | 4 (abab) | 6 (ababac) | 0 |
| 6 | ababac | 7 (ababaca) | 0 (λ) | 0 (λ) | 0 |
| 7 | аbabaсa | 1 (a) | 2 (ab) | 0 (λ) | 0 |



Построим автомат, распознающий вхождение подстроки

*Т* = *ananas* в строку *S.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | *a* | *n* | *s* | *\** |
| 0 | *λ* | 1 *(a)* | 0 *(λ)* | 0 *(λ)* | 0 |
| 1 | *a* | 1 *(a)* | *an* (2) | 0 *(λ)* | 0 |
| 2 | *an* | 3 *(ana)* | 0 *(λ)* | 0 *(λ)* | 0 |
| 3 | *ana* | 1 *(a)* | 4 *(anan)* | 0 *(λ)* | 0 |
| 4 | *anan* | 5 *(anana)* | 0 *(λ)* | 0 *(λ)* | 0 |
| 5 | *anana* | 1 *(a)* | 4 *(anan)* | 6 *(ananas)* | 0 |
| 6 | *ananas* | 1 *(a)* | 0 *(λ)* | 0 *(λ)* | 0 |

(назовем табл.авт.)

Попытаемся найти слово *Т = ananas* в следующей строке

*S = anananaananasak*.

|  |  |  |  |
| --- | --- | --- | --- |
| Находился | Прочитал | Перешел | Исследуемый фрагмент *S* |
| *λ* (0) | S[1] = *а* | табл.авт[0][*a*]=1 | ***a****nananaananasak* |
| *a* (1) | *S*[2] = *п* | табл.авт[1][*п*]=2 | ***an****ananaananasak* |
| *an* (2) | S[3] = *а* | табл.авт[2][*a*]=3 | ***ana****nanaananasak* |
| *ana* (3) | S[4] = *n* | табл.авт[3][*n*]=4 | ***anan****anaananasak* |
| *anan* (4) | S[5] = *а* | табл.авт[4][*a*]=5 | ***anana****naananasak* |
| *anana* (5) | S[6] = *n* | табл.авт[5][*n*]=4 | *an****anan****aananasak* |
| *anan* (4) | S[7] = *а* | табл.авт[4][*a*]=5 | *an****anana****ananasak* |
| *anana* (5) | S[8] = *а* | табл.авт[5][*a*]=1 | *ananana****a****nanasak* |
| *a* (1) | S[9] = *n* | табл.авт[1][*п*]=2 | *ananana****an****anasak* |
| *an* (2) | S[10] = *a* | табл.авт[2][*a*]=3 | *ananana****ana****nasak* |
| *ana* (3) | S[11] = *n* | табл.авт[3][*n*]=4 | *ananana****anan****asak* |
| *anan* (4) | S[12] =*a* | табл.авт[4][*a*]=5 | *ananana****anana****sak* |
| *anana* (5) | S[13] =*s* | табл.авт[5][*s*]=6  нашли подстроку | *ananana****ananas****ak* |
| *ananas* (6) | S[14] =*a* | табл.авт[6][*a*]=1 | *anananaananas****a****k* |
| *a* (1) | S[15] =*k* | табл.авт[1][*k*]= табл.авт[1][\*]=0 | *anananaananasak* |

Можем построить **автомат**: состоянием в нём будет текущее значение префикс-функции, переходы из одного состояния в другое будут осуществляться под действием символа.

m=t.length;

t += '#';

pi - префикс-функция

// Определяем алфавит строки t

alph=new Array()

for(i=0;i<m;i++)

alph[t.charAt(i)]=0

// В двумерном массиве aut храним таблицу переходов автомата

aut=new Array(m+1);

for(j=0;j<=m;j++)

aut[j]=new Array()

for(c in alph)

aut [0][c]=0

for (i=0; i<=m; i++)

for (c in alph) {

j = i;

while (j > 0 && c != t.charAt(j))

j = pi[j-1];

if (c == t.charAt(j)) j++;

aut[i][c] = j;

}

Но заметим, что вместо внутреннего цикла , который постепенно укорачивает ответ, можно воспользоваться уже вычисленной частью таблицы: переходя от значения  к значению , фактически говорим, что переход из состояния  приведёт в то же состояние, что и переход , а для него ответ уже точно посчитан (т.к. ):



for (i=0; i<=m; i++)

for (c in alph) {

if (i > 0 && c != t.charAt(i))

aut[i][c] = aut[pi[i-1]][c];

if (c == t.charAt(i))

aut[i][c] = i + 1;

}

В итоге получилась крайне простая реализация построения автомата, работающая за .



Можно еще поменять места строчек в коде.

for (i=0; i<m; i++){

aut[i][ t.charAt(i)] = i + 1;

for (c in alph)

aut[i+1][c] = aut[pi[i]][c];

}

Заметив, что aut[pi[i-1]][ t.charAt(i)]=pi[i], получим.

Фрагмент кода, который позволяет эффективно строить таблицу переходов автомата.

t=WScript.StdIn.ReadLine ()

//t=’abaxabaz’

m=t.length

alph=new Array()

//Определяем алфавит строки t

for(i=0;i<m;i++)

alph[t.charAt(i)]=0

//В двумерном массиве aut храним таблицу переходов

aut=new Array(m+1)

for(j=0;j<=m;j++)

aut[j]=new Array()

//Инициализируем таблицу переходов

for(c in alph)

aut [0] [c]=0

//Формируем таблицу переходов

for(j=0;j<m;j++){

prev=aut[j][t.charAt(j)]

aut[j] [t .charAt(j)]=j+1

for(c in alph)

aut[j+1] [c]=aut [prev] [c]

}

//Выводим таблицу переходов

for(j=0; j<=m; j + + ){

out=’’

for(c in alph)

out+=aut[j] [c] + ‘ ‘

WScript.Echo(out)

}

**Задание.**

Напоминание.

В текстовом файле input.txt записана на первой строке - строка *S*, на второй строке - строка *Т*. Требуется найти все вхождения строки *Т* в строку S, т. е. указать все позиции строки S, начиная с которых читается строка *Т:*

1. использование алгоритма c Z-функцией.
2. использование алгоритма c префикс-функцией.

(Образуем строку T+&+S. Надо найти вхождение строки *Т* в строку S, поэтому будем хранить в памяти лишь строку T+& и значение префикс-функции на ней, а потом уже считывать по одному символу строку S и пересчитывать текущее значение префикс-функции.

В программе, которая строит префикс-функцию, есть цикл по строке T+&+S

pi[0]=0

k=0

for(i=1;i<длина T+&+ S;i++) {

...

pi[i]=k

}

Но надо сделать в реализации два цикла.

*Первый*: for(i=1;i< длинаT+1;i++).

Заполнить массив префикс функции от 0,.., длинаT.

*Второй*: for(i= длинаT+1;i< длина T+&+ S;i++).

В этом цикле заполнять массив префикс функции не надо.

В конце цикла делаем проверку if (k == длинаT).);

1. использование алгоритма Морриса—Пратта.
2. использование конечного детерминированного автомата.

(Построили таблицу переходов.

Смотрим строку S.

префикс=0 - начнаем с пустого префикса

for(i= 0;i< длинаS;i++) {

Если S.charAt(i) есть в T

префикс= aut[префикс] [S.charAt(i)]

Иначе

префикс=0

Если префикс=длина T

Вывод позиции в строке S

})

Попробовать на примере:

1.

S: На улице колокол продолжал звонить. колокода В тот же миг зазвонили церковные колокола. Мы наслаждались неизвестной музыкой разных народов, то торжественной, как удары колокола, то плавной и напевной. колоком В воздухе вились немногочисленные чайки, где-то вдали на яхте зазвонил судовой колокол. колко Даже большой колокол на крыше, должно быть, ощутил силу ветра, поскольку его верёвка то поднималась, то опускалась, будто колокол передвигался с места на место.

T: колокол

2.

S:

кукукушкк Трудно представить лес без кукушки. Громкое «ку-ку» самца и характерную булькающую трель самки каждый из нас неоднократно слышал (кстати, знаменитое «ку-ку!» - крик «мужской», означающий: «Я тут!»; самка-кукушка издаёт звуки, напоминающие хохот). И что кукушка гнёзд не строит, а подкладывает яйца в чужие гнёзда, также всем нам хорошо известно. Вроде бы, про кукушку всё уже сказано. Но, к сожалению, до сих пор человек мало знает об этой необычной птице, которую чрезвычайно трудно изучать. кукушка - птица осторожная и скрытная. Далеко не все видели ее живьем. А в полете она столь похожа на ястреба, что неспециалисты их постоянно путают. кукушк 32209258365498648

T: кукушк