Алгоритмы поиска в тексте

Наверное, каждому, кто много работает за компьютером, знакома подобная ситуация: перелистывая страницы книги в поисках нужного фрагмента, невольно начинаешь думать о том, как вызвать команду «поиск по всему тексту». Действительно, современные программы обработки текста приучили нас к такой удобной возможности, как поиск и замена фрагментов, и если вы разрабатываете подобную программу, пользователь вправе ожидать, что вы предоставите в его распоряжение соответствующие команды. Эту проблему нельзя эффективно решить при помощи стандартных функций, поскольку большинство средств разработки включает только малоэффективные средства. Во-первых, в стандартных функциях не всегда используются самые эффективные алгоритмы, а во-вторых, вполне возможно, что вам понадобится изменить стандартное поведение этих функций (например, предусмотреть возможность поиска по шаблону).

Рассматриваются два наиболее эффективных алгоритма поиска в тексте.

Алгоритм грубой силы

Прежде, чем приступить к рассмотрению эффективных алгоритмов, составим простейший (и самый медленный) алгоритм прямого поиска, называемый «методом грубой силы».

Задача 1. Найти первое вхождение подстроки P в строку T. |P|=m, |T|=n, m<=n.

Решение. Функция Find ищет подстроку P в строке T и возвращает индекс первого символа подстроки или 0, если подстрока не найдена. Хотя в общем случае этот метод, как и большинство методов грубой силы, малоэффективен, в некоторых ситуациях он вполне приемлем.

Задачи

2. Перевернуть строку.
3. Найти последнее вхождение подстроки в строку.
4. Определить все вхождения образа в строке.
5. Ввести опцию чувствительности/нечувствительности к регистру.
6. Модифицировать простейший алгоритм поиска для нахождения в тексте строки y^i -строки, состоящей из i повторений подстроки y.

Алгоритм грубой силы

Программа решения задачи 1. Найти первое вхождение подстроки P в строку T. |P|=m, |T|=n, m<=n.

```
program Dir1;

{$APPTYPE CONSOLE}

Var P,T:string;

function Find(const T,P: string): Integer;
```

```
var
  i,k,m,n: Integer;
begin
  Result := 0;
  n:=Length(T);
  m:=Length(P);
  if m > n then Exit;
  for i := 1 to n-m+1 do begin
  k:=0;
  while (k<m) and (P[k+1]=T[i+k]) do k:=k+1;
  {По очереди сравниваем все символы начиная с i-ого}
  if k=m then begin {если все символы совпадали}
  Result := i;
  Exit;
end end end;

Begin
Write('P='); Readln(P);
Write('T='); Readln(T);
writeln('Index= ',Find(T,P)); {сообщение о нахождении строки в
тексте}
Readln
End.
```

Задачи

2. Перевернуть строку.
3. Найти последнее вхождение подстроки в строку.
4. Определить все вхождения образа в строке.
5. Ввести опцию чувствительности/нечувствительности к регистру.

Алгоритм Кнута-Морриса-Пратта

Задача 1.

Пусть есть некоторый текст T и слово (или образ) P. Необходимо найти первое вхождение этого слова в указанном тексте. Это действие типично для любых систем обработки текстов.

Решение. Этот алгоритм был создан в 1970 году и получил свое название от имен его разработчиков. Алгоритм Кнута-Морриса-Пратта появился в результате тщательного анализа алгоритма грубой силы. Исследователи хотели найти способы более полно использовать информацию, полученную во время сканирования (алгоритм грубой силы ее просто выбрасывает). Он состоит из двух этапов: 1-подготовительного и 2-основного.

1. На подготовительном этапе учитывается структура подстроки. Для этого формируется массив D, в котором учитывается совпадения следующих символов подстроки (суффикс) с началом подстроки (префикс) следующим образом: для каждой позиции i, совпадающей с началом подстроки, вычисляется максимальное количество предшествующих ей символов. Размер массива D равен длине подстроки.

Другими словами, нас интересуют начала Z слова P[1]..P[i+1], одновременно являющиеся его концами - из них нам надо выбрать самое длинное. Откуда берутся эти начала? Каждое из них (не считая пустого) получается из некоторого слова Z' приписыванием буквы P[i+1]. Слово Z' является началом и концом слова P[1]..P[i]. Однако не любое слово, являющееся началом и концом слова P[1]..P[i], годится - надо, чтобы за ним следовала буква P[i+1].

Получаем такой рецепт отыскания слова Z. Рассмотрим все начала слова P[1]..P[i], являющиеся одновременно его концами. Из них выберем подходящие - те, за которыми идет буква P[i+1]. Из подходящих выберем самое длинное. Приписав в его конец P[i+1], получим искомое слово Z.

Основным отличием алгоритма Кнута, Морриса и Пратта от алгоритма прямого поиска заключается в том, что сдвиг подстроки выполняется не на один символ на каждом шаге алгоритма, а на некоторое переменное количество символов. Следовательно, перед тем как осуществлять очередной сдвиг, необходимо определить величину сдвига D. Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим.

P=ABCABD, m=6

D=(0,0,0,1,2,0)

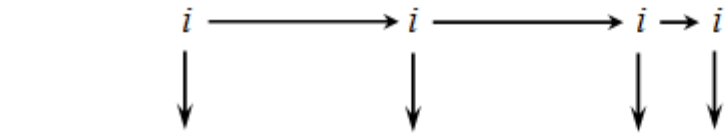
Теперь пора воспользоваться сделанными нами приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему префикс-функции D. D[i] есть длина наибольшего начала слова P[1]..P[i], одновременно являющегося его концом. Вот что получается:

```
i:=1; D[1]:= 0;
{таблица D[1]..D[i] заполнена правильно}
while i <> n do begin
| k := D[i]
| {k - длина начала слова P[1]..P[i], которое  является
|    его концом; все более длинные начала оказались
|    неподходящими}
| while (P[k+1] <> P[i+1]) and (k > 0) do
| | {начало не подходит, применяем к нему функцию D}
| | k := D[k];
| {нашли подходящее или убедились в отсутствии}
| if P[k+1] = P[i+1] do begin
| | {P[1]..P[k] - самое длинное подходящее начало}
| | D[i+1] := k+1;
| end else
| | {подходящих нет}
| | D[i+1] := 0;
| i := i+1;
end;
```

2. Теперь рассмотрим основной этап. Поиск начинается со сравнения первого символа строки с первым символом подстроки. В случае несовпадения происходит сдвиг подстроки на количество символов, указанных соответствующим элементом массива D. Если совпадения подстроки со строкой не будет (то есть данной подстроки в строке нет), то программа

выйдет из цикла для поиска подстроки, когда i будет равняться длине строки, то есть если ни один символ подстроки не совпадает ни с одним символом строки, то программа выполнит N сравнений, если же совпадения отдельных элементов подстроки (но не всей подстроки со строкой) будут найдены, то в наихудшем случае потребуется N+M сравнений. Если же совпадение подстроки со строкой обнаружится сразу, то потребуется M сравнений.

Основным отличием алгоритма Кнута, Морриса и Пратта от алгоритма прямого поиска заключается в том, что сдвиг подстроки выполняется не на один символ на каждом шаге алгоритма, а на некоторое переменное количество символов. Следовательно, перед тем как осуществлять очередной сдвиг, необходимо определить величину сдвига. Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим. На рисунке символы, подвергшиеся сравнению, выделены жирным шрифтом.



Строка	A	B	C	A	B	C	A	A	B	C	A	B	D
Подстрока	**A**	**B**	**C**	**A**	**B**	**D**							
				A	**B**	**C**	**A**	**B**	D				
							A	**B**	C	A	B	D	
								A	**B**	**C**	**A**	**B**	**D**

T=ABCABCAABCABD
P=ABCABD, m=6
D=(0,0,0,1,2,0)
Позиция вхождения=8

Сначала вычисляем таблицу D[1]..D[m] как раньше. Затем пишем такую программу:

```
j:=0; k:=0;
{k - длина максимального начала подслова P, одновременно
       являющегося концом слова T[1]..T[j]}
while (k <> n) and (j <> m) do begin
| while (P[k+1] <> T[j+1]) and (k > 0) do begin
| | {начало не подходит, применяем к нему функцию D}
| | k := D[k];
| end;
| {нашли подходящее или убедились в отсутствии}
| if P[k+1] = T[j+1] do begin
| | {P[1]..P[k] - самое длинное подходящее начало}
| | k := k+1;
| end else
| | {подходящих нет}
| | k := 0;
| j := j+1;
```

```
end;
{если k=n, слово P встретилось; иначе мы дошли до конца
   строки T, так и не встретив P}
```

Задачи

2. Найти последнее вхождение подстроки в строку.

3. Определить все вхождения образа в строке по методу Кнута, Морриса и Пратта.

4. Написать программу поиска образа в строке по методу Кнута, Морриса и Пратта. Ввести опцию чувствительности/нечувствительности к регистру.

5. Реализовать в программе алгоритм прямого поиска строки и КМП-алгоритм. Сравнить эффективность поиска образа в строке обоими алгорит мами по количеству итераций.

6. Ниже приведен листинг программы, формирующей таблицу *d* (Pf в программе) по КМП-алгоритму. При каком образе таблица *d* будет сформирована неверно? При какой строке и каком образе положительный результат не будет получен?

```
m = strlen(img);
j = 0; k = −1; d[0] = –1;
while(j <m)
{
while((k >= 0) && ( img[j] != img[k])) k = d[k];
j++; k++;
if (img[j] == img[k]) d[j] = d[k];
else d[j]= k;
}
```

7. Модифицировать алгоритм Кнута-Морриса- Пратта для нахождения в тексте строки y^i -строки, состоящей из i повторений подстроки y.

8. Дана строка S. Требуется найти такую строку T, что строка S получается многократным повторением T. Из всех возможных T нужно выбрать наименьшую по длине.

Эту задачу очень просто решить за O(N) с помощью префикс-функции. Итак, пусть массив Pf - префикс-функция строки S, которую можно вычислить за O(N). Теперь рассмотрим значение последнего элемента Pf: Pf[N-1] (если индекс начинается с 0). Если N делится на (N - Pf[N-1]), то ответ существует, и это N - Pf[N-1] первых букв строки S. Если же не делится, то ответа не существует.

Корректность этого метода легко понять. Pf[N-1] равно длине наидлиннейшего собственного суффикса строки S, совпадающего с префиксом S. Если ответ существует, то, очевидно, начальный кусок строки S длиной (N - Pf[N-1]) и будет ответом, и, следовательно, N будет делиться на (N - Pf[N-1]). Если же ответа не существует, то (N - Pf[N-1]) будет равно какому-то непонятному значению, на которое N делиться не будет (иначе бы ответ существовал).

9. Объясните, как найти вхождения образца P в текст T, зная функцию Pf для PT (т.е. строки длиной m+n, полученной в результате конкатенации строк P и T).

Алгоритм Кнута-Морриса-Пратта

Программа решения задачи1

```
program KMP2;

{$APPTYPE CONSOLE}
type TMas = array[1..300] of word;
var P,T: string;
    Pf: TMas; {массив, в котором хранятся значения префикс-функции}
    i,k,m,n: Integer;

Procedure Prefix(P:String; Var Pf:TMas);
{процедура, вычисляющая префикс-функцию}
var k,m,i: Integer;
Begin
 Pf[1]:=0; {префикс строки из одного символа имеет нулевую длину}
 k:=0;
 m:=Length(P);
 for i:=2 to m do
 {вычисляется для префиксов строки длинной от 2 до m символов}
 begin
 {значение функции может быть получено из ранее сделанных вычислений}
  while (k>0) and (P[k+1]<>P[i]) do k:=Pf[k];
  if P[k+1]=P[i] then k:=k+1;
  Pf[i]:=k; {присвоение префикс-функции}
 end;
End;

{Алгоритм Кнута-Мориса-Пратта, устанавливающий}
{вхождение непустой подстроки P в строку S}
Begin
 {Ввод текста и образца}
 Write('P='); Readln(P);  m:=length(P);  // образец
 Write('T='); Readln(T);  n:=length(T);  // строка текста

 Prefix(P,Pf); {Вычисление префикс-функции}
 k:=0; {количество символов, совпавших на данный момент}
 for i:=1 to n do
 begin
  while (k>0) and (P[k+1]<>T[i]) do k:=Pf[k];
  if P[k+1]=T[i] then k:=k+1;
  if k = m then {если совпали все символы}
  begin
   writeln('Index=', i-m+1); break
  end
 end;
Readln
End.
```

Задачи

1. Найти последнее вхождение подстроки в строку.

2. Определить все вхождения образа в строке по методу Кнута, Морриса и Пратта.

Решение 1 (универсальное). Пусть функция SearchPos(StartPos: Integer; const T, P: String) : Integer; возвращает позицию первого символа первого вхождения образца P в строке T. Если последовательность P в T не найдена, функция возвращает 0. Параметр StartPos позволяет указать позицию в строке T, с которой следует начинать поиск. Для поиска всех вхождений P в T с самого начала строки следует задать StartPos равным 1. Если результат поиска не равен нулю, то для того, чтобы найти следующее вхождение P в T, нужно задать StartPos равным значению «предыдущий результат плюс длина образца».

Решение 2 (специальное). Объясните, как найти вхождения образца P в текст T, зная функцию Pf для PT (т.е. строки длиной m+n, полученной в результате конкатенации строк P и T).

3. Написать программу поиска образа в строке по методу Кнута, Морриса и Пратта. Ввести опцию чувствительности/нечувствительности к регистру.

4. Реализовать в программе алгоритм прямого поиска строки и КМП-алгоритм. Сравнить эффективность поиска образа в строке обоими алгорит мами по количеству итераций.

5. Ниже приведен листинг программы, формирующей таблицу *d* (Pf в программе) по КМП-алгоритму. При каком образе таблица *d* будет сформирована неверно? При какой строке и каком образе положительный результат не будет получен?

```
m = strlen(img);
j = 0; k = −1; d[0] = −1;
while(j <m)
{
while((k >= 0) && ( img[j] != img[k])) k = d[k];
j++; k++;
if (img[j] == img[k]) d[j] = d[k];
else d[j]= k;
}
```

7. Модифицировать алгоритм Кнута-Морриса- Пратта для нахождения в тексте строки y^i -строки, состоящей из i повторений подстроки y.

8. Дана строка S. Требуется найти такую строку T, что строка S получается многократным повторением T. Из всех возможных T нужно выбрать наименьшую по длине.

Эту задачу очень просто решить за O(N) с помощью префикс-функции. Итак, пусть массив Pf - префикс-функция строки S, которую можно вычислить за O(N). Теперь рассмотрим значение последнего элемента Pf: Pf[N-1] (если индекс начинается с 0). Если N делится на (N - Pf[N-1]), то ответ существует, и это N - Pf[N-1] первых букв строки S. Если же не делится, то ответа не существует.

Корректность этого метода легко понять. Pf[N-1] равно длине наидлиннейшего собственного суффикса строки S, совпадающего с префиксом

S. Если ответ существует, то, очевидно, начальный кусок строки S длиной (N - Pf[N-1]) и будет ответом, и, следовательно, N будет делиться на (N - Pf[N-1]). Если же ответа не существует, то (N - Pf[N-1]) будет равно какому-то непонятному значению, на которое N делиться не будет (иначе бы ответ существовал).

Алгоритм Бойера-Мура

Задача 1.

Найти подстроку P в строке T.

Решение. Алгоритм Бойера-Мура, разработанный двумя учеными – Бойером (Robert T. Boyer) и Муром (J. Strother Moore), считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке. Прежде чем рассмотреть работу этого алгоритма, уточним некоторые термины. Под *строкой* мы будем понимать всю последовательность символов текста. Собственно говоря, речь не обязательно должна идти именно о тексте. В общем случае строка – это любая последовательность байтов. Поиск подстроки в строке осуществляется по заданному *образцу*, т. е. некоторой последовательности байтов, длина которой не превышает длину строки. Наша задача заключается в том, чтобы определить, содержит ли строка заданный образец.

Простейший вариант алгоритма Бойера-Мура состоит из следующих шагов. На первом шаге мы строим таблицу смещений для искомого образца. Процесс построения таблицы будет описан ниже. Далее мы совмещаем начало строки и образца и начинаем проверку с последнего символа образца. Если последний символ образца и соответствующий ему при наложении символ строки не совпадают, образец сдвигается относительно строки на величину, полученную из таблицы смещений, и снова проводится сравнение, начиная с последнего символа образца. Если же символы совпадают, производится сравнение предпоследнего символа образца и т. д. Если все символы образца совпали с наложенными символами строки, значит мы нашли подстроку и поиск окончен. Если же какой-то (не последний) символ образца не совпадает с соответствующим символом строки, мы сдвигаем образец на один символ вправо и снова начинаем проверку с последнего символа. Весь алгоритм выполняется до тех пор, пока либо не будет найдено вхождение искомого образца, либо не будет достигнут конец строки.

Величина сдвига в случае несовпадения последнего символа вычисляется исходя из следующих соображений: сдвиг образца должен быть минимальным, таким, чтобы не пропустить вхождение образца в строке. Если данный символ строки встречается в образце, мы смещаем образец таким образом, чтобы символ строки совпал с самым правым вхождением этого символа в образце. Если же образец вообще не содержит этого символа, мы сдвигаем образец на величину, равную его длине, так что первый символ образца накладывается на следующий за проверявшимся символ строки.

Величина смещения для каждого символа образца зависит только от порядка символов в образце, поэтому смещения удобно вычислить заранее и

хранить в виде одномерного массива, где каждому символу алфавита соответствует смещение относительно последнего символа образца. Поясним все вышесказанное на простом примере. Пусть у нас есть набор символов из пяти символов: a, b, c, d, e и мы хотим найти вхождение образца “abbad” в строке “abeccacbadbabbad”. Следующие схемы иллюстрируют все этапы выполнения алгоритма:

a	b	c	d	e
1	2	5	0	5

Таблица смещений для образца “abbad”.

```
a b e c c a c b a d b a b b a d
a b b a d
```

Начало поиска. Последний символ образца не совпадает с наложенным символом строки. Сдвигаем образец вправо на 5 позиций:

```
a b e c c a c b a d b a b b a d
          a b b a d
```

Три символа образца совпали, а четвертый – нет. Сдвигаем образец вправо на одну позицию:

```
a b e c c a c b a d b a b b a d
            a b b a d
```

Последний символ снова не совпадает с символом строки. В соответствии с таблицей смещений сдвигаем образец на 2 позиции:

```
a b e c c a c b a d b a b b a d
                a b b a d
```

Еще раз сдвигаем образец на 2 позиции:

```
a b e c c a c b a d b a b b a d
                  a b b a d
```

Теперь, в соответствии с таблицей, сдвигаем образец на одну позицию, и получаем искомое вхождение образца:

```
a b e c c a c b a d b a b b a d
                    a b b a d
```

Реализуем указанный алгоритм на языке ObjectPascal. Прежде всего следует определить тип данных «таблица смещений». Для кодовой таблицы, состоящей из 256 символов, определение этого типа будет выглядеть так:

```
type
  TBMTable = array [0..255] of Integer;
```

Далее приводится процедура, вычисляющая таблицу смещений для образца P.

```
procedure MakeBMTable( var BMT : TBMTable; const P : String);
var
  i : Integer;
begin
  for i := 0 to 255 do BMT[i] := Length(P);
```

```
  for i := Length(P) downto 1 do
      if BMT[Byte(P[i])] = Length(P) then
        BMT[Byte(P[i])] := Length(P) - i;
end;
```

Теперь напишем функцию BMSearch(StartPos : Integer; const T, P : String; const BMT : TBMTable) : Integer; осуществляющую поиск.

Функция BMSearch возвращает позицию первого символа первого вхождения образца P в строке T. Если последовательность P в T не найдена, функция возвращает 0 (напомню, что в ObjectPascal нумерация символов в строке String начинается с 1). Параметр StartPos позволяет указать позицию в строке T, с которой следует начинать поиск. Это может быть полезно в том случае, если вы захотите найти все вхождения P в T. Для поиска с самого начала строки следует задать StartPos равным 1. Если результат поиска не равен нулю, то для того, чтобы найти следующее вхождение P в T, нужно задать StartPos равным значению «предыдущий результат плюс длина образца».

Сравнение образа со строкой происходит до тех пор, 1) пока не будет рассмотрен весь образ, что говорит о том, что соответствие между образом и некоторой частью строки найдено, 2) пока не закончится строка, что значит, что вхождений, соответствующих образу в строке нет, 3) либо пока не произойдет несовпадения символов образа и строки, что вызывает сдвиг образа на несколько символов вправо и продолжение процесса поиска.

В том случае, если произошло несовпадение символов, смещение образа по строке определяется значением элемента таблицы BMT, причем индексом данного элемента является код ASCII символа *строки*. Подчеркнем, что, несмотря на то, что массив BMT формируется на основе образа, при смещении индексом служит символ из строки.

На рис. показан образ и значения элементов массива BMT, соответствующие символам данного образа.

образ: `Hooligan`

значения элементов массива *d*: `75543210`

На рис. показан пример работы алгоритма БМ-поиска, сравниваемые символы подчеркнуты.

```
Hoola-Hoola girls like Hooligans.
Hooligan
     Hooligan
       Hooligan
              Hooligan
                     Hooligan
```

Эффективность БМ-алгоритма

Замечательным свойством БМ-поиска является то, что почти всегда, кроме специально построенных примеров, он требует значительно меньше *N* сравнений. В самых же благоприятных обстоятельствах, когда последний

символ образа всегда попадает на несовпадающий символ строки, число сравнений равно N/M.

Авторы приводят и несколько соображений по поводу дальнейших усовершенствований алгоритма. Одно из них – объединить приведенную только что стратегию, обеспечивающую большие сдвиги в случае несовпадения, со стратегией Кнута, Морриса и Пратта, допускающей «ощутимые» сдвиги при обнаружении совпадения (частичного). Такой метод требует двух таблиц, получаемых при предтрансляции: d_1 – только что упомянутая таблица BMP, а d_2 – таблица, соответствующая КМП-алгоритму. Из двух сдвигов выбирается больший, причем и тот и другой «говорят», что никакой меньший сдвиг не может привести к совпадению.

Задачи

2. Найти последнее вхождение подстроки в строку.

3. Определить все вхождения образа в строке по методу Бойера и Мура.

4. Написать программу поиска образа в строке по методу Бойера и Мура. Предусмотреть возможность существования в образе пробела. Ввести опцию чувствительности/нечувствительности к регистру.

5. Реализовать в программе алгоритм прямого поиска строки и БМ-алгоритм. Сравнить эффективность поиска образа в строке обоими алгоритмами по количеству итераций.

6. Разработать и программно реализовать усовершенствованный алгоритм, объединяющий БМ-алгоритм с КМП-алгоритмом, который при сдвиге образа использует две таблицы, полученные согласно данным алгоритмам.

7. Сравнить по быстродействию алгоритм Кнута-Морриса-Пратта и алгоритм Бойера-Мура при поиске в тексте образца, все символы которого различны.

8. Модифицировать алгоритм Бойера-Мура для нахождения в тексте строки y^i -строки, состоящей из i повторений подстроки y.

9. Обобщить алгоритма Бойера-Мура на алфавит, который содержит d символов.

Алгоритм Бойера-Мура

Программа решения задачи 1.

```
program BM1;

{$APPTYPE CONSOLE}
type TBMTable = array [0..255] of Integer;
var P,T: string;
    BMT: TBMTable; {таблица смещений для образца P}
    i: Integer;

procedure MakeBMTable( var BMT : TBMTable; const P : String);
var
  i : Integer;
begin
  for i := 0 to 255 do BMT[i] := Length(P);
    for i := Length(P) downto 1 do
```

```
      if BMT[Byte(P[i])] = Length(P) then
         BMT[Byte(P[i])] := Length(P) - i;
end;


{ Алгоритм Бойера-Мура, устанавливающий }
{ вхождение непустой строки P в строку T}
function BMSearch( StartPos : Integer; const T, P : String;
  const BMT : TBMTable) : Integer;
var
  Pos, lp, i : Integer;
begin
  lp := Length(P);
  Pos := StartPos + lp -1;
  while Pos <= Length(T) do
    if P[lp] <> T[Pos] then Pos := Pos + BMT[Byte(T[Pos])]
    else for i := lp - 1 downto 1 do
      if P[i] <> T[Pos - lp + i] then
      begin
        Inc(Pos);
        Break;
      end
      else if i = 1 then
      begin
        Result := Pos - lp + 1;
        Exit;
      end;
  Result := 0;
end;


Begin
 {Ввод текста и образца}
 Write('P='); Readln(P);   // образец
 Write('T='); Readln(T);   // строка
 MakeBMTable(BMT,P); {Вычисление таблицы смещений для образца P}
 writeln('Index=', BMSearch(1,T,P,BMT));
Readln
End.
```

Задачи

2. Найти последнее вхождение подстроки в строку.

3. Определить все вхождения образа в строке по методу Бойера и Мура.

4. Написать программу поиска образа в строке по методу Бойера и Мура. Предусмотреть возможность существования в образе пробела. Ввести опцию чувствительности/нечувствительности к регистру.

5. Реализовать в программе алгоритм прямого поиска строки и БМ-алгоритм. Сравнить эффективность поиска образа в строке обоими алгоритмами по количеству итераций.

6. Разработать и программно реализовать усовершенствованный алгоритм, объединяющий БМ-алгоритм с КМП-алгоритмом, который при сдвиге образа использует две таблицы, полученные согласно данным алгоритмам.