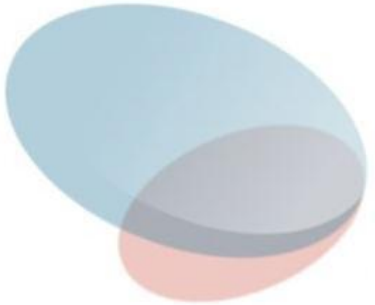


Поиск в тексте
Лекция 6

Поиск в тексте

Задачи, требующие эффективных алгоритмов работы с текстом: работа в текстовом редакторе,

- поисковые запросы в базах данных,
- задачи в биоинформатике,
- лексический анализ программ и т.д.

Задачи поиска слова в тексте используются также:

- в криптографии,
- в различных разделах физики,
- при сжатии данных,
- в распознавании речи и других сферах человеческой деятельности.

Терминология

Алфавит – конечное **множество** символов.

Строка (слово) – последовательность символов из некоторого алфавита. **Длина строки** – количество символов в строке.

Строка X называется **подстрокой** строки Y, если найдутся такие строки Z1 и Z2, что Y=Z1X Z2.

Подстрока X называется **префиксом** строки Y, если есть такая подстрока Z, что Y=XZ.

Подстрока X называется **суффиксом** строки Y, если есть такая подстрока Z, что Y=ZX.

Поиск в тексте

Задача поиска подстроки в строке

Пусть **задана строка**, состоящая из некоторого количества **символов**. Проверим, **входит ли заданная подстрока в данную строку**. Если входит, то **найдем номер, начиная с какого символа строки**, то есть, определим **первое вхождение заданной подстроки в исходной строке**.

Будем считать, что задан массив Txt из **N элементов**, называемый **текстом**, и массив Wrd из **M элементов**, называемый **словом**, причем $0 < M \leq N$. Описать их можно как строки.

Прямой (последовательный)поиск

Данный алгоритм является самым простым и очевидным.

Он заключается в **посимвольном сравнении текста со словом**.

В начальный момент происходит сравнение первого символа текста с первым символом слова, второго символа текста со вторым символом слова и т. д.

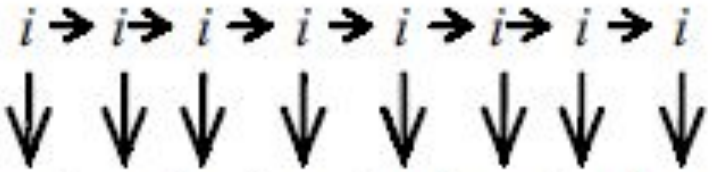
Если **все пары сравниваемых символов совпали**, то фиксируется факт нахождения слова.

В противном случае производится «сдвиг» слова на одну позицию вправо, и повторяется посимвольное сравнение, т.е. сравнивается второй символ текста с первым символом слова, третий символ текста со вторым символом слова и т. д.

Эти «сдвиги» слова повторяются до тех пор, пока конец слова не достигнет конца текста или не произойдет полное совпадение символов слова с текстом (т.е. слово будет найдено).

Прямой (последовательный)поиск

Алгоритм прямого поиска в тексте



Текст	A	B	C	A	B	C	A	A	B	C	A	B	D
Слово	A	B	C	A	B	D							
		A	B	C	A	B	D						
			A	B	C	A	B	D					
				A	B	C	A	B	D				
					A	B	C	A	B	D			
						A	B	C	A	B	D		
							A	B	C	A	B	D	

Прямой (последовательный)поиск

```
//Описание функции прямого поиска подстроки в строке C++
int DirectSearch(char *string, char *substring){
  int sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )     cout << "Неверно задана строка\n";
  else if ( ssl == 0 )     cout << "Неверно задана подстрока\n";
  else
    for (int i = 0; i < sl - ssl + 1; i++)
      for (int j = 0; j < ssl; j++)
        if ( substring[j] != string[i+j] )
          break;
        else if ( j == ssl - 1 ){
          res = i;
          break;
        }
  return res;
}
```

Прямой (последовательный)поиск

Алгоритм **работает достаточно эффективно**, если **несовпадение пары символов происходит после незначительного количества сравнений во внутреннем цикле**.

При достаточно большом множестве символов это довольно частый случай. Можно предполагать, что для текстов, составленных из 128 символов алфавита, несовпадение будет обнаруживаться после одной или двух проверок.

В худшем случае алгоритм будет малоэффективен, так как его **сложность будет пропорциональна O((N – M)·M).**

Алгоритм Кнута, Морриса и Пратта

Приблизительно в 1970 году Д. Кнут, Д. Моррис и В. Пратт изобрели алгоритм (КМП-алгоритм), фактически требующий только **O(N) сравнений** даже **в худшем случае**.

Алгоритм основывается на том соображении, что **после частичного совпадения начальной части слова с соответствующими символами текста** фактически известна пройденная часть текста и можно «**вычислить**» некоторые **сведения** (на основе самого слова), с помощью которых затем **быстро продвинуться по тексту**.

Основным **отличием** КМП-алгоритма от алгоритма прямого поиска является **выполнение сдвига слова не на один символ на каждом шаге алгоритма, а на некоторое переменное количество символов**. **Перед тем как осуществлять очередной сдвиг, необходимо определить величину сдвига**.

Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был как можно большим.

Алгоритм Кнута, Морриса и Пратта

Если **j** определяет **позицию в слове, содержащую первый несовпадающий символ** (как в алгоритме прямого поиска), то **величина сдвига Shift_j** определяется как

Shift_j = **j – LenSuff – 1**.

По определению, если **j=1** или **j=2**, то **Shift_j =1**.

Суффиксом в этом алгоритме называется **самая длинная последовательность символов слова, непосредственно предшествующих позиции j, которая полностью совпадает с началом слова.**

Замечание. Суффикс является частью слова и не совпадает со всем словом.

Значение **LenSuff** определяется как **размер суффикса** (длина). **LenSuff зависит** только **от слова** и **не зависит от текста**. **Для каждого j** будет своя величина сдвига Shift, которую обозначим **Shift_j.**

Алгоритм Кнута, Морриса и Пратта

Пример 1. Текст='ABCABCAABCABD'.

Слово='ABCABD'.

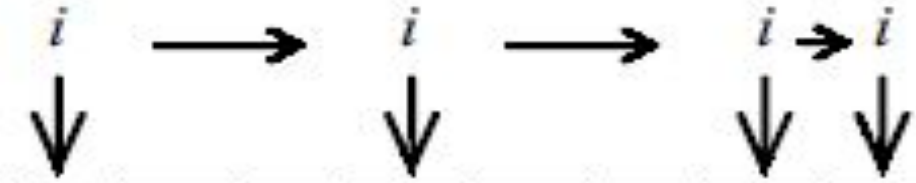
1 шаг. i=1. j=6. suff='AB'. shift= **j – LenSuff – 1**=6-2-1=3.

2 шаг. i=4. j=5. suff='A'. shift=5-1-1=3.

3 шаг. i=7. j=2. shift=1, по определению

4 шаг. i=8.

Символы, подвергшиеся сравнению, подчеркнуты.



Текст	A	B	C	A	B	C	A	A	B	C	A	B	D
Слово	A	B	C	A	B	D							
				A	B	C	A	B	D				
							A	B	C	A	B	D	
								A	B	C	A	B	D

Алгоритм Кнута, Морриса и Пратта

Замечание: при каждом несовпадении пары символов слово сдвигается на переменную величину, и меньшие сдвиги не могут привести к полному совпадению.

Так как величины Shift_j зависят только **от слова**, то **перед началом фактического поиска можно вычислить вспомогательную таблицу Shift.** Это оправдано в том случае, **если размер текста значительно превышает размер слова (M<<N).**

Если нужно искать многие вхождения одного и того же слова, то можно пользоваться одной и той же таблицей Shift.

Точный анализ КМП-алгоритма сложен. Его изобретатели доказали, что **требуется порядка O(M + N) сравнений символов**, **что значительно лучше, чем O((N – M)·M) сравнений при прямом поиске**.

Алгоритм требует O (M) дополнительное пространство памяти.

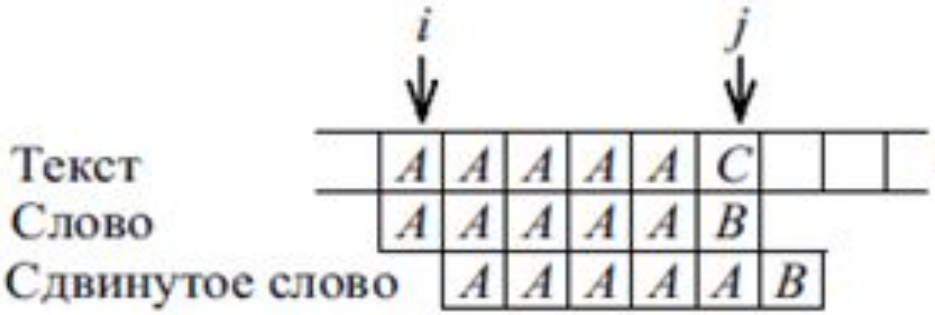
Алгоритм Кнута, Морриса и Пратта

КМП-алгоритм дает подлинный **выигрыш только тогда**, когда **неудаче предшествовало некоторое число совпадений**. Лишь в этом случае слово сдвигается более чем на единицу.

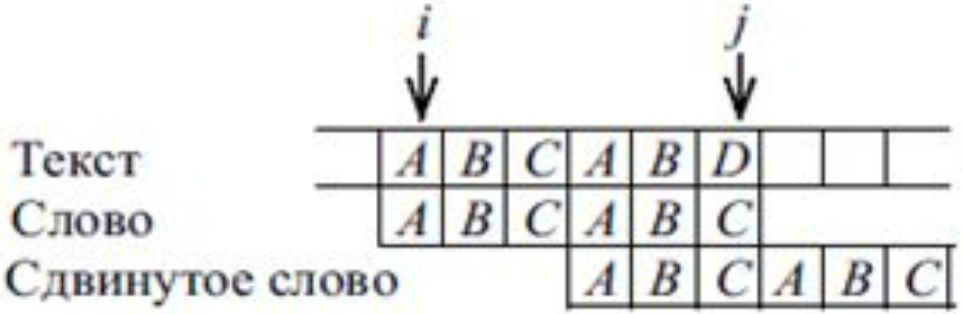
Но это скорее исключение, чем правило: совпадения встречаются значительно реже, чем несовпадения. Поэтому выигрыш от использования КМП-алгоритма в большинстве случаев поиска в обычных текстах весьма незначителен.

Алгоритм Кнута, Морриса и Пратта

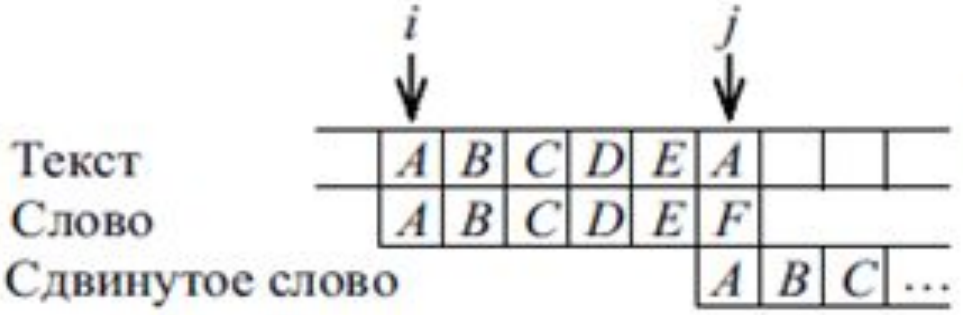
Пример 2. Частичное совпадение со словом и вычисление Shift_j.
Чем меньше длина суффикса, тем больше сдвиг.



$j = 6$, LenSuff = 4, Shift[6] = j – LenSuff – 1 = 1
Wrd[1]..Wrd[4] = Wrd[2]..Wrd[5]



$j = 6$, LenSuff = 2, Shift[6] = j – LenSuff – 1 = 3
Wrd[0]..Wrd[1] = Wrd[3]..Wrd[4]
Wrd[0]..Wrd[2] ≠ Wrd[2]..Wrd[4]
Wrd[0]..Wrd[3] ≠ Wrd[1]..Wrd[4]



$j = 6$, LenSuff = 0, Shift[6] = j – LenSuff – 1 = 5
Wrd[0]..Wrd[0] ≠ Wrd[4]..Wrd[4]
Wrd[0]..Wrd[1] ≠ Wrd[3]..Wrd[4]
Wrd[0]..Wrd[2] ≠ Wrd[2]..Wrd[4]
Wrd[0]..Wrd[3] ≠ Wrd[1]..Wrd[4]

Алгоритм Кнута, Морриса и Пратта

```
//Листинг C++ стр.1
//описание функции алгоритма Кнута, Морриса и Пратта C++
int KMPSearch(char *string, char *substring){
  int  sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )     cout << "Неверно задана строка\n";
  else if ( ssl == 0 )     cout << "Неверно задана подстрока\n";
  else {
    int  i, j = 0, k = -1;
    int  *d;
    d = new int[1000];
    d[0] = -1;
```

Алгоритм Кнута, Морриса и Пратта

```
//Листинг C++ стр.2
while ( j < ssl - 1 ) {
      while ( k >= 0 && substring[j] != substring[k] )
        k = d[k];
      j++;
      k++;
      if ( substring[j] == substring[k] )
        d[j] = d[k];
      else
        d[j] = k;
    }
```

Алгоритм Кнута, Морриса и Пратта

```
//Листинг C++ стр.3
i = 0;
    j = 0;
    while ( j < ssl && i < sl ){
      while ( j >= 0 && string[i] != substring[j] )
        j = d[j];
      i++;
      j++;
    }
    delete [] d;
    res =  j == ssl ? i - ssl : -1;
  }
  return res;
}
```

Алгоритм Боуера и Мура

Метод, предложенный Р. Боуером и Д. Муром в 1975 (1977?) году (БМ-алгоритм), не только **улучшает обработку самого плохого случая** КМП-алгоритма (совпадений при сравнениях не было), но и **дает выигрыш в промежуточных ситуациях.**

БМ-алгоритм основывается на необычном соображении – **сравнение символов начинается с конца слова**, а не с начала.

Как и в случае КМП-алгоритма, **перед фактическим поиском на основе слова формируется некоторая таблица Shift_x:**

- для каждого символа **x** **из алфавита** величина **Shift_x** есть **расстояние от самого правого в слове вхождения x до правого конца слова;**
- если символ **x** из алфавита **не входит в слово**, то **Shift_x** равно длине слова M.

Алгоритм Боуера и Мура

Сравнение символов начинается с конца слова.

Пусть обнаружено расхождение между словом и текстом, причем **символ в тексте**, **который не совпал**, есть **x**.

Пусть **j** – позиция в слове символа, который не совпал с **x**,

В этом случае **слово** сразу же **можно сдвинуть вправо так**, чтобы **самый правый символ слова, равный x, оказался бы в той же позиции, что и символ текста x.**

То есть **слово смещается вправо** на **Shift = j+ Shift[x]-M =Shift[x]-(M – j) символов**, где **M** – длина слова.

Этот сдвиг, скорее всего, будет на число позиций, большее единицы.

Если несовпадающий символ текста x в слове вообще не встречается, **то** сдвиг становится даже больше: **сдвигаем вправо так, чтобы ни один символ слова не накладывался на символ x.**

Алгоритм Боуера и Мура

Пример. Текст=‘ABCAFDFABCABD’. Слово=‘ABCABD’.

Shift[‘A’]=2; Shift[‘B’]=1; Shift[‘C’]=3; Shift[‘D’]=0;

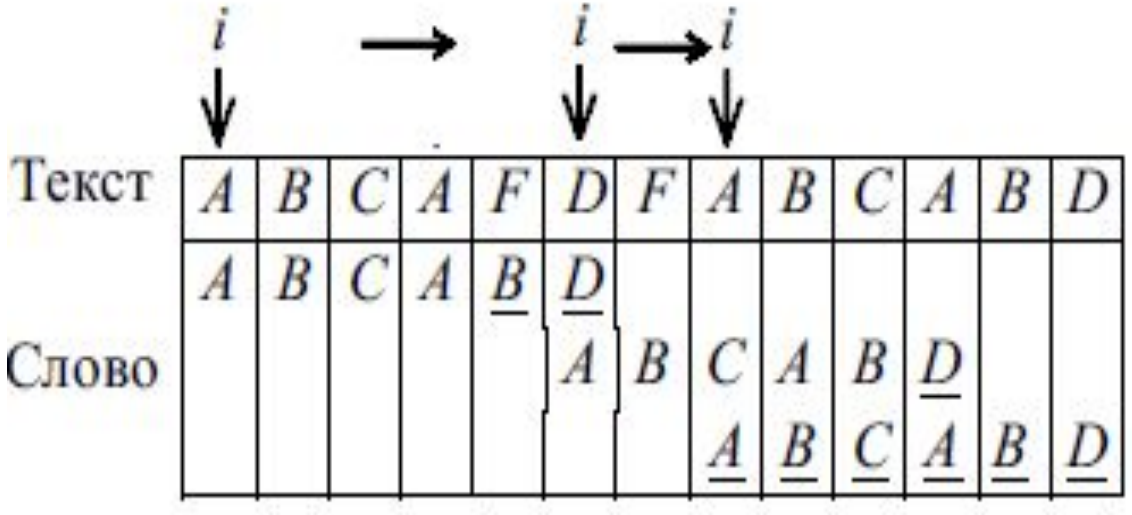
Shift[‘F’]=6=M, (Shift[x]=6=M, x∈Alf и x∉слову)

1 шаг. i=1. j=5. x= ‘F’ не совпало с ‘B’. ‘F’ не входит в слово, поэтому shift[‘F’]=6 (длине слова).

Слово смещается на j+ shift[‘F’]-M =5+6-6=5 символов вправо.

2 шаг. i=6. j=6. x= ‘A’ не совпало с ‘D’. shift[‘A’]=2. Слово смещается на j+ shift[‘A’]-M =6+2-6=2 символа вправо.

3 шаг. i=8.



Не совпало с "F", "F" нет в слове

Не совпало с "A", Shift ["A"] = 2

Алгоритм Боуера и Мура

Почти всегда, кроме специально построенных примеров, **алгоритм требует значительно меньше O(N) сравнений**.

В **наилучшем случае для этого алгоритма, когда последний символ слова всегда попадает на несовпадающий символ текста (и не входящий в слово)**, число сравнений пропорционально **O(N/M).**

В наихудшем случае трудоемкость рассматриваемого **алгоритма O(N+M).**

Алгоритм Боуера и Мура на хороших данных очень быстр, а вероятность появления плохих данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск.

Таким образом, данный алгоритм является наиболее эффективным в обычных ситуациях, а его **быстродействие повышается при увеличении длины подстроки или алфавита**.

Алгоритм Боуера и Мура

```
//Листинг C++ стр.1
//описание функции алгоритма Бойера и Мура C++
int BMSearch(char *string, char *substring){
  int  sl, ssl;
  int res = -1;
  sl = strlen(string);
  ssl = strlen(substring);
  if ( sl == 0 )     cout << "Неверно задана строка\n";
  else if ( ssl == 0 )     cout << "Неверно задана подстрока\n";
  else {
    int  i, Pos;
    int  BMT[256];
    for ( i = 0; i < 256; i ++ )
       BMT[i] = ssl;
    for ( i = ssl-1; i >= 0; i-- )
      if ( BMT[(short)(substring[i])] == ssl )
        BMT[(short)(substring[i])] = ssl - i - 1;
```

Алгоритм Боуера и Мура

```
//Листинг C++ стр.2
Pos = ssl - 1;
   while ( Pos < sl )
      if ( substring[ssl - 1] != string[Pos] )
        Pos = Pos + BMT[(short)(string[Pos])];
      else
        for ( i = ssl - 2; i >= 0; i-- ){
          if ( substring[i] != string[Pos - ssl + i + 1] ) {
            Pos += BMT[(short)(string[Pos - ssl + i + 1])] - 1;
            break;
          }
          else
            if ( i == 0 )
              return Pos - ssl + 1;
          cout << "\t" << i << endl;
        }
  }
  return res;
}
```

Алгоритм Боуера и Мура

Усовершенствование алгоритма

Существуют попытки совместить присущую алгоритму Кнута, Морриса и Пратта эффективность в "плохих" случаях и скорость алгоритма Бойера и Мура в "хороших“.

Авторы алгоритма 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: Shift' – только что упомянутая таблица, а Shift'' – таблица, соответствующая КМП-алгоритму. **Из двух сдвигов выбирается больший.**

Алгоритм Боуера и Мура

Усовершенствование алгоритма

Но дополнительное усложнение формирования таблиц и самого поиска, может не оправдать видимого выигрыша в производительности. Фактические дополнительные расходы могут быть высокими, и неизвестно, приведут ли эти изменения к выигрышу или проигрышу.

Каждый алгоритм поиска позволяет эффективно действовать лишь для своего класса задач. Алгоритм поиска подстроки в строке следует выбирать только после точной постановки задачи.

Алгоритмы поиска в тексте

Алгоритм прямого поиска – это алгоритм поиска подстроки в строке, при котором происходит посимвольное сравнение строки с подстрокой.

Алгоритм Кнута, Морриса и Пратта – это алгоритм поиска подстроки в строке, при котором сдвиг подстроки выполняется на некоторое переменное количество символов.

Алгоритм Боуера и Мура – это алгоритм поиска подстроки в строке, при котором первоначально строится таблица смещений для искомой подстроки, проверка начинается с последнего символа подстроки после совмещения начала подстроки и строки.

Имеется **алгоритм поиска подслов с помощью конечных автоматов.**

Алгоритмы поиска в тексте

Поиск подстрок с помощью конечных автоматов

Пример

Дана последовательность букв x[1],...,x[n]. Определить, имеются ли в ней идущие друг за другом буквы ‘abcd’. Другими словами, требуется выяснить, есть ли в слове x[1]...x[n] подслово ‘abcd’.

Читая слово x[1]...x[n] слева направо, мы ожидаем появления буквы 'a'. Как только она появилась, мы ждем за ней букву 'b', затем 'c', и, наконец, 'd'. Если наши ожидания оправдываются, то слово ‘abcd’ обнаружено. Если же какая-то из нужных букв не появляется, мы начинаем всё сначала.

Используя терминологию конечных автоматов, можно сказать, что при чтении слова x слева направо мы в каждый момент находимся в одном из следующих состояний: "начальное" (0), "сразу после a" (1), "сразу после ab" (2), "сразу после abc" (3), "сразу после abcd" (4).

Как только мы попадём в состояние 4, поиск заканчивается.

Алгоритмы поиска в тексте

Читая очередную букву, переходим в следующее состояние по правилу, описанному в таблице:

Текущее состояние	Очередная буква	Новое состояние
0 – начальное	a	1
0 – начальное	кроме a	0
1 – сразу после a	b	2
1 – сразу после a	a	1
1 – сразу после a	кроме a, b	0
2 – сразу после ab	c	3
2 – сразу после ab	a	1
2 – сразу после ab	кроме a, c	0
3 – сразу после abc	d	4
3 – сразу после abc	a	1
3 – сразу после abc	кроме a, d	0

Алгоритмы поиска в тексте

Программа для примера

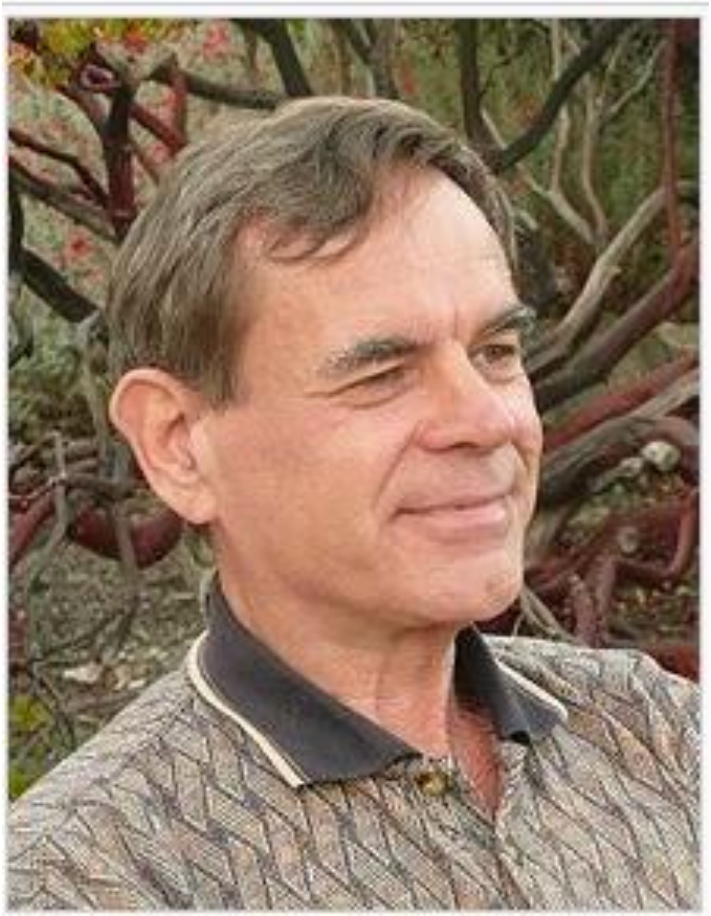
```
//поиск подслова "abcd" в заданном слове x
var n : Integer;  //длина исходного слова
state: byte; //состояние
i : Integer;  //индекс первой непрочитанной буквы
x : String[255];
begin
        //ввод строки x
        n:=Length(x);
        i:=1;   state:=0;
        while (i<>n+1) and (state<>4) do
        begin
                If state=0 then If x[i]='a' then state:=1 else state:=0
                else If state=1 then If x[i]='b' then state:=2
                else If x[i]='a' then state:=1 else state:=0
                else If state=2 then If x[i]='c' then state:=3
                else if x[i]='a' then state:=1 else state:=0
                else If state=3 then If x[i]='d' then state:=4
                else If x[i]='a' then state:=1 else state:=0;
                i:=i+1
        end;
        //вывод результата
end;
```

Алгоритм Кнута, Морриса и Пратта

Дональд Эрвин Кнут (англ. Donald Ervin Knuth родился 10 января 1938) — американский учёный, почётный профессор Стэнфордского университета и нескольких других университетов, иностранный член Российской академии наук, автор 19 монографий, разработчик нескольких известных программных технологий. Автор всемирно известной серии книг, посвящённой основным алгоритмам и методам вычислительной математики, создатель настольных издательских систем TEX и METAFONT, предназначенных для набора и вёрстки книг, посвящённых технической тематике.

Алгоритм Кнута, Морриса и Пратта

В. Пратт (Vaughan Pratt Рональд, род. 1944), заслуженный профессор в Стэнфордском университете , был одним из пионеров в области компьютерных наук . Издается с 1969 года, Пратт сделал несколько вкладов в основополагающих областях, таких как поисковые алгоритмы , алгоритмы сортировки и проверки простоты чисел . В последнее время его исследования были направлены на формальное моделирование параллельных систем.



Алгоритм Кнута, Мориса и Пратта

Джеймс Х. Моррис (р. 1941) является профессором компьютерных наук. Ранее он занимал должность декана Carnegie Mellon Школа компьютерных наук и декана Карнеги-Меллона в Силиконовой долине. В течение десяти лет работал в Xerox PARC (Palo Alto Research Center), где был частью команды, которая разработала Xerox Alto системы.

Алгоритм Боуера и Мура

Р. Боуер (Robert Stephen Boyer Bob Boyer) профессор компьютерных наук, математики и философии в Техасском университете.

Д. Мур (J Strother Moore) ученый в области компьютерных наук. Профессор компьютерных наук, математики в Техасском университете.