

Поиск в тексте

Постановка задачи. Пусть есть некоторый текст T и слово (или образ) W .

Необходимо найти первое вхождение этого слова в указанном тексте.

Это действие типично для любых систем обработки текстов.

Элементы массивов T и W – символы некоторого конечного алфавита – например, $\{0, 1\}$, или $\{a, \dots, z\}$, или $\{a, \dots, я\}$.

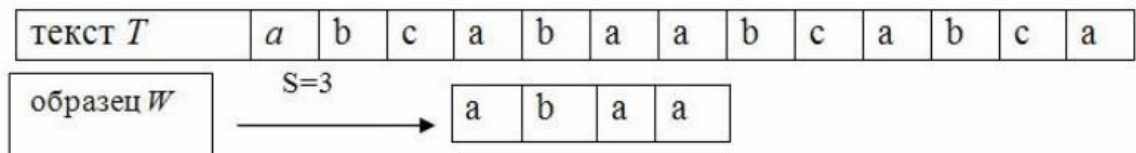
Алгоритм прямого поиска

Поиск строки формально определяется следующим образом.

Пусть задан массив T из N элементов и массив W из M элементов, причем $0 < M \leq N$.

Поиск строки обнаруживает первое вхождение W в T , результатом будем считать индекс i , указывающий на первое с начала строки (с начала массива T) совпадение с образом (словом).

Пример. Требуется найти все вхождения образца $W = abaa$ в текст $T = abcabaabscabca$. Это последовательный поиск.



Образец входит в текст только один раз, со сдвигом $S=3$, индекс $i=4$.

findstr(char *s, char *temp) od

I – индекс строки s , j – индекс строки $temp$, ls – длина s , lt – длина $temp$

$i \leftarrow 0, j \leftarrow 0, f \leftarrow 0, ls \leftarrow \text{length}(s), lt \leftarrow \text{length}(temp);$

if ($ls < lt$) неудачный поиск -1;

while ($i < ls - lt$) //сравнений $ls - lt$ ($N - M$)

od

$f \leftarrow i;$

while($i < ls$ and $j < lt$ and $s[i] == temp[j]$) do //сравнений $3 * lt$ ($3 * M$)

$i \leftarrow i + 1; j \leftarrow j + 1;$

od

if ($j = lt$) удачный поиск f ; //еще одно

else

do

$j \leftarrow 0; i \leftarrow i + 1; \}$

od

od

Неудачный -1;

Od

Сложность алгоритма:

Худший случай (обнаружение образца в конце текста).

Пусть строка $T \rightarrow \{AAA \dots AAAB\}$, длина $|T| = N$, образец $W \rightarrow \{A \dots AB\}$, длина $|W| = M$.

Очевидно, что для обнаружения совпадения в конце строки потребуется произвести порядка $N * M$ сравнений, то есть $O(N * M)$.

Недостатки алгоритма:

1. высокая сложность — $O(N * M)$, в худшем случае — $\Theta((NM + 1) * M)$;

2. после несовпадения просмотр всегда начинается с первого символа образца и поэтому может включать символы T, которые ранее уже просматривались (если строка читается из вторичной памяти, то такие возвраты занимают много времени);
3. информация о тексте T, получаемая при проверке данного сдвига S, никак не используется при проверке последующих сдвигов.

Алгоритм Бойера-Мура (БМ)

Робертом Бойером и Джеем Муром

Идея БМ-поиска – сравнение символов начинается с конца образца, а не с начала, то есть сравнение отдельных символов происходит справа налево.

Пример

T: Иван Иванович по фамилии Иваникин

W: Иваникин

Идея алгоритма

Для всех символов образа W, кроме последнего, длина сдвига устанавливается так:

- для предпоследнего =1
- для следующего от конца=2, затем 3 и т.д.
- для первого=n-1.

Если в образе один и тот же символ встречается несколько раз (без учета последнего), то для него устанавливается наименьшее значение.

Для всех остальных символов алфавита, который используется для представления строк T и W, в том числе и для последнего символа, если он не повторяется в образе, длина сдвига устанавливается равной длине образа W.

При несовпадении любой пары символов T[j] и W[k] осуществляется сдвиг на величину, установленную для первого с конца символа сравниваемой части строки T[i].

Т.е. если сравнение началось с символа T[i], а не совпавшим символом любой сравниваемый символ T[j], то сдвиг будет на одну и ту же величину, установленную для символа T[i].

Образец W – Иваникин

Текст –T

Иван И~~в~~анови~~ч~~ по фами~~и~~лии Ивани~~к~~ин

Иваникин (а сдвиг на 5)

Иваникин (ч сдвиг на 8)

Иваникин (и сдвиг на 1)

Иваникин (л сдвиг на 8)

Иваникин (и сдвиг на 1)

Иваникин (к сдвиг на 2)

Иваникин строка найдена

Таблица сдвигов для символов w справа налево, начиная с предпоследнего и -1, к-2, н-4, а-5, в-6, И-7

В T выделяется подстрока длины W. Сравнивается подстрока T с конца с W. Если символы не совпали, то выполняется сдвиг вправо по тексту T на установленное для не совпавшего символа смещение.

Декомпозиция алгоритма

- Создать таблицу сдвигов P
- Создать алгоритм поиска вхождения подстроки в строку

Анализ сложности

На практике алгоритм БМ-поиска наиболее эффективен, если образец W длинный, а мощность алфавита достаточно велика. Почти всегда, кроме специально построенных примеров, БМ-поиск требует значительно меньше N сравнений.

В самых же благоприятных обстоятельствах, когда последний символ образца всегда попадает на несовпадающий символ текста, число сравнений равно (N / M) .

Временная сложность в худшем же случае – $O(N+M)$.

По памяти $O(N+M+P)$. P - длина таблицы сдвигов

Алгоритм Кнута-Морриса-Пратта (КМП)

Эта задача является классическим применением префикс-функции (и, собственно, она и была открыта в связи с этим).

Префикс - функция

Для определения смещений для каждого символа в подстроке специально разработана префикс-функция.

Определение

Дана строка $S[0..n]$ Требуется вычислить для неё префикс-функцию, т.е. массив чисел $P[0..n-1]$, где $P[i]$ определяется следующим образом:

это такая наибольшая длина наибольшего собственного суффикса подстроки $S[0..i]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой).

В частности, значение $P[0]$ полагается равным нулю.

Математически определение префикс-функции можно записать следующим образом:

$$P[i] = \max\{k: S[0..k-1] = S[i-k+1..i]\}$$

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 "aabaab" она равна: 0,1,0,1,2,3.

Постановка задачи Алгоритма КМП

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 .

Решение

Образуем строку $s\#t$, где символ $\#$ — это разделитель, который не должен встречаться в тексте. 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 $P[i]$ показывает наибольшую длину подстроки, оканчивающуюся в позиции i и совпадающего с префиксом.

При этом $P[i]$ — 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 равенство $P[i]=n$ (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 $P[i]=n$, то в позиции $i-(n+1)-n+1=i-2n$ строки t начинается очередное вхождение строки S в строку t . Если известно, что значения префикс-функции не будут превышать некоторой величины, то достаточно хранить не всю строку и префикс-функцию, а только её начало. Это означает, что нужно хранить в памяти лишь строку $S\#$ и значение префикс-функции на ней, а потом уже считать по одному символу строку t и пересчитывать текущее значение префикс-функции.

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

Примеры

```
1)string ss = "abcabedcabcabcd";
```

```
string t = "abcabed";
```

```
ss = t + "$" + ss;
```

```
abcabed$abcabedcabcabcd
```

```
vector<int> P= prefix_function(ss);
```

Содержимое P

```
00012300123456701234567
```

```
14 22
```

$i=14$ тогда позиция размещения значения $i-2n=14-2*7=0$

$i=22 \Rightarrow 22-2*7=8$

```
2)
```

```
string ss="Iv Ivanovic po familii Ivanikin";
```

```
string t = "Ivaniki";
```

```
0000000012012340000000000000000012345670
```

```
37-14=23
```

//Формирование массива префиксов

```
vector<int> prefix_function(string s) {
```

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

```
vector<int> P(n);
```

```
for (int i = 1; i<n; ++i) {
```

```
int j = P[i - 1];
```

```
while (j > 0 && s[i] != s[j])
```

```
j = P[j - 1];
```

```
if (s[i] == s[j]) ++j;
```

```
P[i] = j;
```

```
}
```

```
return P;
```

```
}
```

Алгоритм Рабина-Карпа (РК)

1987 году Майклом Рабином и Ричардом Карпом.

Алгоритм Рабина — Карпа — это алгоритм поиска строки, который ищет образец (подстроку), в тексте, используя хеширование.

Алгоритм редко используется для поиска одиночного шаблона, но имеет значительную теоретическую важность и очень эффективен в поиске совпадений множественных шаблонов одинаковой длины. Одно из простейших практических применений алгоритма Рабина — Карпа состоит в определении плагиата. Алгоритм Рабина — Карпа пытается ускорить проверку эквивалентности образца с подстроками в тексте, используя хеш-функцию. Хеш-функция — это функция, преобразующая каждую строку в числовое значение, называемое хеш-значением (хеш);

В алгоритме Рабина-Карпа создадим хеш для образца, который мы ищем, и будем проверять, соответствует текущий хэш текста хэшу шаблона или нет. Если не соответствует, мы можем гарантировать, что шаблон не существует в тексте. Однако, если он соответствует, шаблон может присутствовать в тексте.

Пример

S1: Олег Иванович по фамилии Иваникин

S2: Иван

Чтобы вычислить хеш, нам нужно использовать простое число. Это может быть любое простое число. Для этого примера возьмем $\text{prime} = 11$.

Мы определим хеш-значение, используя эту формулу:

$(\text{код 1 символа}) * (\text{prime})^0 + (\text{код 2 символа}) * (\text{prime})^1 + (\text{код 3 символа}) * (\text{prime})^2 + \dots$

Обозначим коды алфавита образца Иван

И -> 1

в -> 2

а -> 3

н -> 4

о -> 5

и -> 6

ч -> 7

ф -> 8

л -> 9

к -> 10

О -> 11

е -> 12

г -> 13

м -> 14

Хеш-значение Иван будет:

$$1 * 11^0 + 2 * 11^1 + 3 * 11^2 + 4 * 11(3) = 5710$$

Теперь мы находим текущий хеш нашего текста.

Рассматриваемый текст из исходной строки длиной 4 символа: Олег

Если скользящий хеш совпадает с хеш-значением нашего шаблона, мы проверим соответствие строк или нет. Поскольку наш шаблон имеет 4 буквы, мы возьмем с 1-ой 4 буквы и вычислим значение хеша. Мы получаем:

$$11 * 11^0 + 9 * 11^1 + 12 * 11^2 + 13 * 11(3)$$

О л е г полученная сумма не совпадает с хеш образца.

RabinKarp(s[1..n], sub[1..m]) do

hsub := hash(sub[1..m])

hs := hash(s[1..m])

```

for i <- 1 to (n-m+1) do
  if hs = hsub
    if s[i..i+m-1] = sub
      return i
    hs := hash(s[i+1..i+m])
  od
return not found
od

```

Анализ алгоритма Рабина - Карпа

Две проблемы, связанные с таким поиском

Первая состоит в том, что, так как существует очень много различных строк, между двумя различными строками может произойти коллизия — совпадение их хешей. В таких случаях необходимо посимвольно проверять совпадение самих подстрок, что занимает достаточно много времени, если данные подстроки имеют большую длину (эту проверку делать не нужно, если ваше приложение допускает ложные срабатывания).

При использовании достаточно хороших хеш-функций (смотрите далее) коллизии случаются крайне редко, и в результате среднее время поиска оказывается невелико. Одно из простейших практических применений алгоритма Рабина — Карпа состоит в определении плагиата.

Например, что студент пишет работу по какой-то теме. Коварный профессор находит различные исходные материалы по этой теме и автоматически извлекает список предложений в этих материалах. Затем алгоритм Рабина — Карпа может быстро найти в проверяемой статье примеры вхождения некоторых предложений из исходных материалов.

Для устранения чувствительности алгоритма к небольшим различиям можно игнорировать детали, такие как регистр или пунктуация, при помощи их удаления.

Поскольку количество строк, которые мы ищем, k , очень большое, обычные алгоритмы поиска одиночных строк становятся неэффективными. Для текста длины n и шаблона длины m его среднее и лучшее время исполнения равно $O(n)$ при правильном выборе хеш-функции, но в худшем случае он имеет эффективность $O(nm)$, что является одной из причин того, почему он не слишком широко используется.

Для приложений, в которых допустимы ложные срабатывания при поиске, то есть, когда некоторые из найденных вхождений шаблона на самом деле могут не соответствовать шаблону, алгоритм Рабина — Карпа работает за гарантированное время $O(n)$ и при подходящем выборе рандомизированной хеш-функции вероятность ошибки можно сделать очень малой.

Также алгоритм имеет уникальную особенность находить любую из заданных k строк одинаковой длины в среднем (при правильном выборе хеш-функции) за время $O(n)$ независимо от размера k . Эффективность в лучшем случае $O(n)$, в худшем случае $O(nm)$

Цифровой поиск

Цифровой поиск – частный случай поиска заданной подстроки (образца) в длинной строке (тексте). Поиск подстроки по образцу используется в текстовых редакторах, в Интернетных поисковиках и т.п. Примеры цифрового поиска: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).

Еще один пример – словарь с побуквенными метками для первого и последнего слова на каждой страницы (обычно помещаются в верхнем колонтитуле).

При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). Развивая идею побуквенных меток до ее логического завершения, получаем схему поиска. Цифровой поиск – реализация этой идеи.

Ожидаемое число сравнений порядка $O(\log_m N)$, где m – число различных букв, используемых в словаре, N – мощность словаря.

В худшем случае дерево содержит k , уровней, где k – длина максимального слова.

Пример.

Пусть множество используемых букв (алфавит) $\{A, B, C, D\}$.

Мы добавим к алфавиту еще одну букву (пробел).

По определению слова AA , $AA_$, $AA_ _$, совпадают.

Пусть $\{A, AA, ABB, AC, ADBD, BCA, BCD, CBA\}$ – словарь (множество ключей).

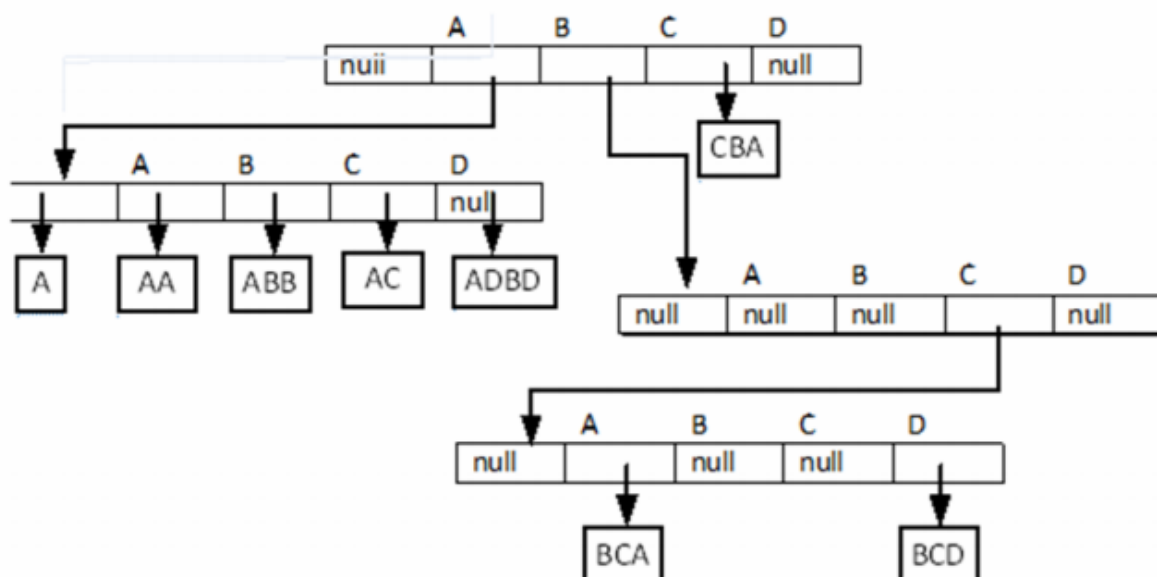
Построим m -ричное дерево, где $m = 5 = | _ , A, B, C, D |$. (для понимания алгоритма)

Следующая небольшая хитрость позволит иногда сократить поиск:

если в словаре есть слово $a_1a_2a_3\dots a_k$ и первые i его букв ($i < k$) задают уникальное значение и комбинация $a_1\dots a_i$ встречается в словаре только один раз, то не нужно строить для него отдельное дерево для $j > i$, так как слово можно идентифицировать по первым i буквам.

На рисунке изображено дерево поиска (5-ричное).

Прямоугольниками изображены вершины дерева, в небольших прямоугольниках – значения слов (ключей) и связанная с ним информация (комментарий к слову или любая другая информация, которую мы будем обрабатывать). Тем самым любая вершина дерева – массив из m элементов (в данном примере $m = 5$). Каждый элемент вершины содержит либо ссылку на другую вершину m -ичного дерева, либо на содержащий информацию (ключ).



Рассмотрим корень дерева: как видно из словаря все слова начинаются с A, B или C, причем с C начинается только одно слово – CBA. Поэтому ссылки на узел для слов, начинающихся с A, на узел для слов, начинающихся с B, и на слово CBA (оно единственно). Рассмотрим узел для слов, начинающихся с A: у него пять ссылок на слова A, AA, ABB, AC, ADBD. Все перечисленные слова уникальны, так что дополнительных узлов не требуется. Рассмотрим узел для слов, начинающихся с B: оба слова начинаются с комбинации BC. Это приводит к необходимости завести еще один узел для слов, начинающихся с BC. Это обстоятельство ухудшает эффективность, так как рассматриваемое дерево содержит много нулевых указателей (7), что вызывает большой перерасход памяти. Можно, конечно, исправить положение, модифицировав поиск.

Поиск по бору – цифровой поиск

Такое название алгоритм получил от слова выборка, так как это суть алгоритма поиска.

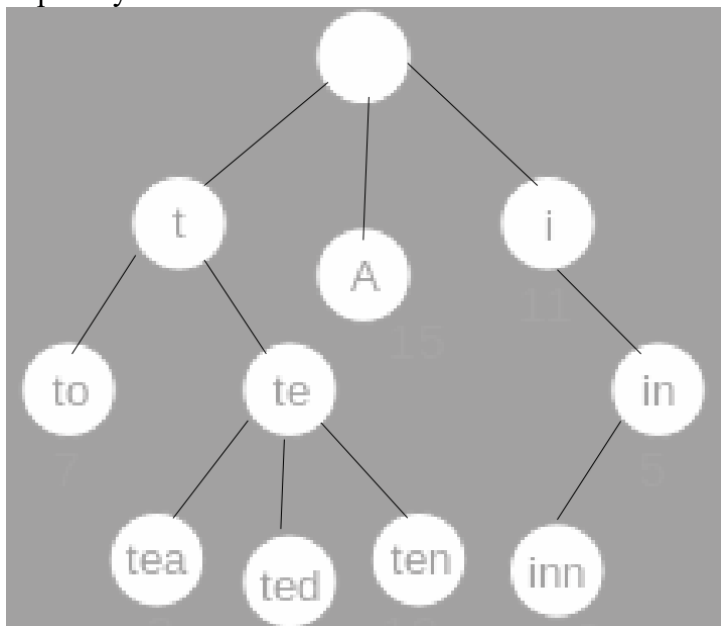
Бор представляет собой t-арное дерево. Каждый узел уровня h — это множество всех ключей, начинающихся с определенной последовательности из h символов.

Для хранения m-ричного дерева используется таблица. Каждый узел – представляет столбец таблицы (это массив размером m, индексированные с 0). Каждый элемент массива это либо ключ, либо ссылка на узел дерева (возможно пустая).

Алфавит ключей	Номера узлов			
	1	2	N
—				
A				
B				

Пробел () — обязательный символ таблицы. В первом узле записывается первая буква или цифра ключа. Во втором узле к ней добавляется еще один символ и т.д. Если слово,

начинающееся с определенной буквы (цифры), единственное, то оно сразу записывается в первом узле.



Бор — структура данных, которая хранит информацию о наборе строк и например позволяет проверять вхождения заданной строки t в заданный набор строк за длину t . Здесь под вхождением понимается проверка того, что строка совпадает с элементом набора. Бор — это такое дерево на ребрах которого написаны символы, тогда путь до каждого из листьев произносится как некоторая строка из набора, а каждая подстрока произносится как некий путь от корня. Обычно вершины в которых заканчивается строка помечаются.

Для проверки строки t на принадлежность набору надо спускаться по дереву и если:

- удалось спуститься произнеся строку t целиком
- спуск завершился в помеченной вершине, то строка t является элементом набора

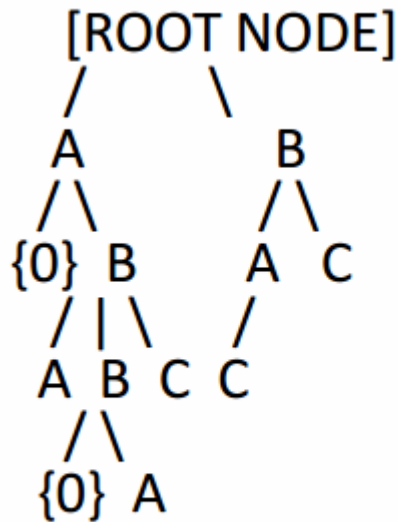
Если выполнится только первый пункт, то строка t является префиксом некоторого элемента из набора. В худшем случае бор требует память равную сумме длин строк набора. Проверка принадлежности строки t набору осуществляется за длину строки t .

Префиксное дерево по бору

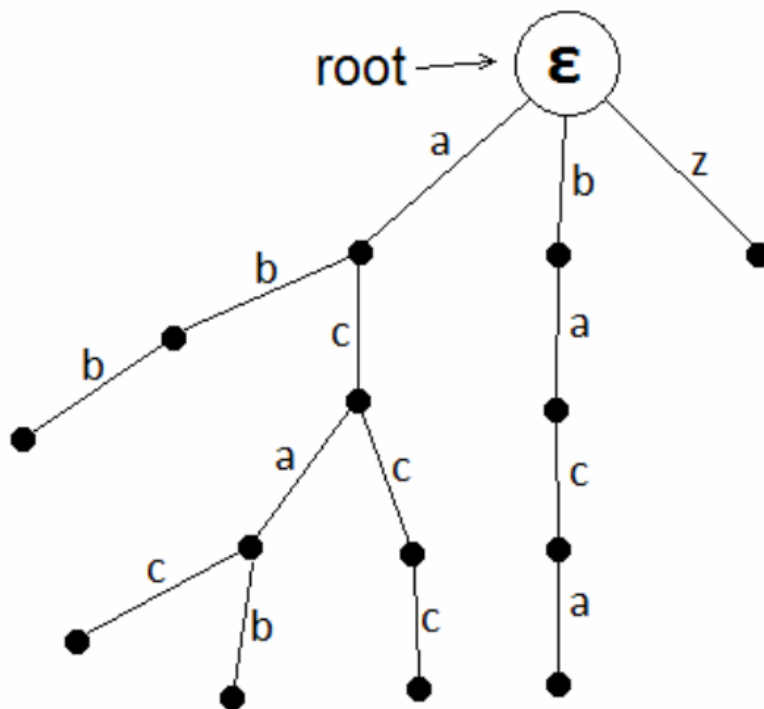
Дерево цифрового поиска (также бор, Префиксное дерево) - структура данных, которая используется для осуществления поразрядного поиска.

Каждый узел дерева содержит значение одного "разряда" (в нашем случае - строковый символ). Вся строка представлена последовательностью узлов от корня до листа дерева.

Пример - дерево цифрового поиска для строк A, ABA, ABV, ABBA, ABC, BAC, BC:

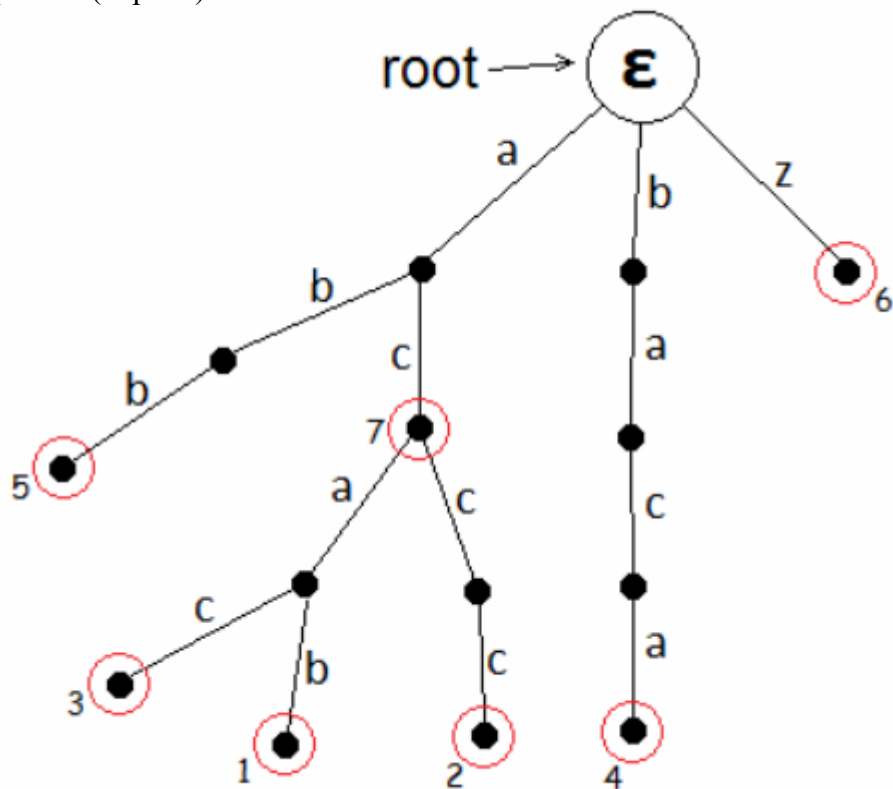


Структура бора Построение бора по набору строк-образцов Бор — это дерево, в котором каждая вершина обозначает какую-то строку (корень обозначает нулевую строку — ε). На ребрах между вершинами написана 1 буква, таким образом, добираясь по ребрам из корня в какую-нибудь вершину и конкатенируя буквы из ребер в порядке обхода, мы получим строку, соответствующую этой вершине. Из определения бора, как дерева, вытекает также единственность пути между корнем и любой вершиной, следовательно — каждой вершине соответствует ровно одна строка (в дальнейшем будем отождествлять вершину и строку, которую она обозначает). Строить бор будем последовательным добавлением исходных строк. Изначально у нас есть 1 вершина, корень (root) — пустая строка. Добавление строки происходит так: начиная в корне, движемся по нашему дереву, выбирая каждый раз ребро, соответствующее очередной букве строки. Если такого ребра нет, то мы создаем его вместе с вершиной. Вот пример построенного бора для строк: 1)acab, 2)accc, 3)acac, 4)baca, 5)abb, 6)z, 7)ac.



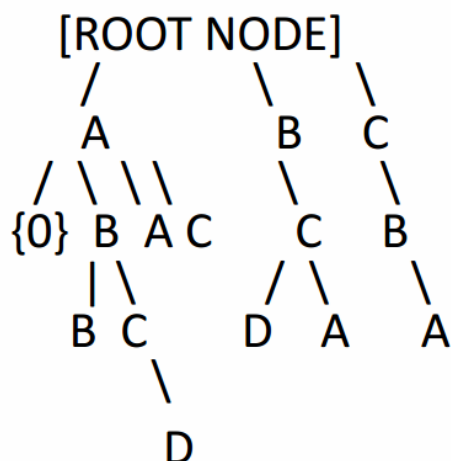
Обратите внимание на добавление строки 7. Она не создает новых вершин и ребер, а процесс ее добавления останавливается во внутренней вершине. Отсюда видно, что для каждой строки необходимо дополнительно хранить признак того является она строкой из условия или нет (красные круги).

Отметим также что, две строки в боре имеют общие ребра при условии наличия у них общего префикса. Крайний случай — все строки образцы попарно не имеют одинаковой начальной части. Значит верхняя оценка для числа вершин в боре — сумма длин всех строк + 1(корень).



Пример создания бора и представления дерева таблицей

Имеется последовательность слов A, AA, ABB, AC, ABCD, BCA, BCD, CBA



	Номера узлов				
Алфавит	1	2	3	4	5
(пробел)		A	AB_	ABC_	BC_
A	(2)	AA			BCA
B	(5)	(3)	ABB		
C	CB A	AC	(4)		
D				ABCD	BCD

Обобщенный алгоритм поиска по бору

По-простому: префикс строки длины X - это первые X символов строки, суффикс строки длины Y - последние Y символов строки.

Алгоритм A1 поиска префикса строки S по дереву:

1. Пусть L = длина строки S.
2. Если L = 0 ("пустая" строка), то выход: префикс не найден.
3. Если дерево не содержит узлов ("пустое" дерево), то выход: префикс не найден.
4. Терминальный узел T = NULL.
5. Текущий узел P = корень дерева ([ROOT NODE]).
6. Цикл по K = 1..L:
7. Если у текущего узла нет дочернего, соответствующего символу S[K], то выход из цикла.
8. Текущий узел P = дочерний, узел, соответствующий символу S[K].
9. Если у узла P есть дочерний узел с терминальным символом {0}, то присваиваем его T.
10. Конец цикла.
11. Если узел P - лист, то выход: префикс найден.
12. Если T не равно NULL, то выход: префикс найден.
13. Префикс не найден.

Алгоритм A2 поиска подстроки строки S по дереву.

1. Пусть L = длина строки S.
2. Если L = 0 ("пустая" строка), то выход: подстрока не найдена.
3. Если дерево не содержит узлов ("пустое" дерево), то выход: подстрока не найдена.
4. Цикл по K = 1..L:
5. Пусть Sk - суффикс строки S, начиная с K-госимвола (суффикс длины L-K+1).
6. Используем алгоритм A1 для поиска префикса строки Sk по дереву; если префикс найден, то выход: подстрока найдена.
7. Конец цикла.
8. Подстрока не найдена.

Эффективность поиска по бору

1. Занимает много памяти. Так для представления 31 ключа часто используемых английских слов была построена таблица размером 30*12, т.е. использует 360 слов памяти, многие пустые, в то время как бинарное дерево поиска занимает 62 слова памяти. Однако можно использовать разреженные структуры данных и задействовать под бор всего 49 слов.

2. Удачный поиск требует около 26 единиц времени. При неудачном поиске бор оказывается быстрее бинарного дерева поиска

Алгоритм Ахо-Корасик

Пусть дан набор строк в алфавите размера k суммарной длины m . Алгоритм Ахо-Корасик строит для этого набора строк структуру данных "бор", а затем по этому бору строит автомат, всё за $O(m)$ времени и $O(mk)$ памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время. Данный алгоритм был предложен канадским учёным Альфредом Ахо (Alfred Vaino Aho) и учёным Маргарет Корасик (Margaret John Corasick) в 1975 г. Алгоритм Ахо-Корасик реализует эффективный поиск всех вхождений всех строк-образцов в заданную строку. Был разработан в 1975 году Альфредом Ахо и Маргарет Корасик.

Условие задачи.

На вход поступают несколько строк `pattern[i]` и строка `s`.

Результат: найти все возможные вхождения строк `pattern[i]` в `s`.

Суть алгоритма заключена в использовании структуры данных — бора и построения по нему конечного детерминированного автомата. Важно помнить, что задача поиска подстроки в строки тривиально реализуется за квадратичное время, поэтому для эффективной работы важно, чтоб все алгоритмы в алгоритме Ахо-Корасика линейно зависели от длины строки.

Цифровой бор

В качестве строк используются двоичные записи чисел, включая ведущие нули.

Таким образом он имеет глубину w .

Так же как и в алгоритме поиска по дереву в узлах дерева хранятся полные ключи, но для выбора правой и левой ветви используется не результат сравнения ключей, а биты аргумента К.

Код слова WITH: 11010 01001 00011 01000. Чтобы найти его на рисунке со словами, сначала надо сравнить со словом THE, так как они не совпали, то смотрим первый бит слова WITH, он равен 1, тогда идем на правую ветвь. Сравниваем со словом OF. Так как они не совпали, и второй бит слова WITH равен 1, то идем на правую ветвь узла OF. Ключ найден из узла можно извлечь дополнительную информацию

