Оглавление

[Основные определения, связанные со строками. 2](#_Toc61367302)

[Классические алгоритмы поиска (линейный, поиск с барьером, бинарный). 4](#_Toc61367303)

[Префикс-функция: понятие, эффективный алгоритм построения. 5](#_Toc61367304)

[Z-функция: понятие, эффективный алгоритм построения. 6](#_Toc61367305)

[Поиск подстроки в строке: наивный алгоритм поиска подстроки в строке, алгоритм Кнута-Морриса-Пратта, алгоритм поиска подстроки в строке на основе использования Z-функции. 7](#_Toc61367306)

[Поиск подстроки в строке: алгоритм Бойера-Мура-Хорспула. 8](#_Toc61367307)

[Поиск подстроки в строке: алгоритм Рабина-Карпа. 9](#_Toc61367308)

[Поиск подстроки в строке: алгоритм Shift-And. 9](#_Toc61367309)

[Поиск подстроки в строке: алгоритм Shift-Or. 9](#_Toc61367310)

[Бор: понятие, алгоритм построения. 9](#_Toc61367311)

[Сжатый бор: понятие, алгоритм построения. 10](#_Toc61367312)

[Поиск в боре. 11](#_Toc61367313)

[Суффиксное дерево: понятие, наивный алгоритм построения, поиск в суффиксном дереве. 11](#_Toc61367314)

[Суффиксное дерево: понятие, алгоритм Э. Укконена. 13](#_Toc61367315)

[Суффиксный массив: понятие, наивный алгоритм построения, поиск в суффиксном массиве. 14](#_Toc61367316)

[Суффиксный массив: понятие, алгоритм построения с временной сложностью O(n2logn). 14](#_Toc61367317)

[Суффиксный массив: понятие, алгоритм построения с временной сложностью O(nlogn). 14](#_Toc61367318)

# Основные определения, связанные со строками.

**Символ** (англ. symbol) – объект, имеющий собственное содержание и уникальную читаемую форму.​​

**Алфавит** (англ. alphabet) – конечное непустое множество символов. Условимся обозначать алфавит большой греческой буквой Σ.​

​

Пример: Наиболее часто используются следующие алфавиты:​

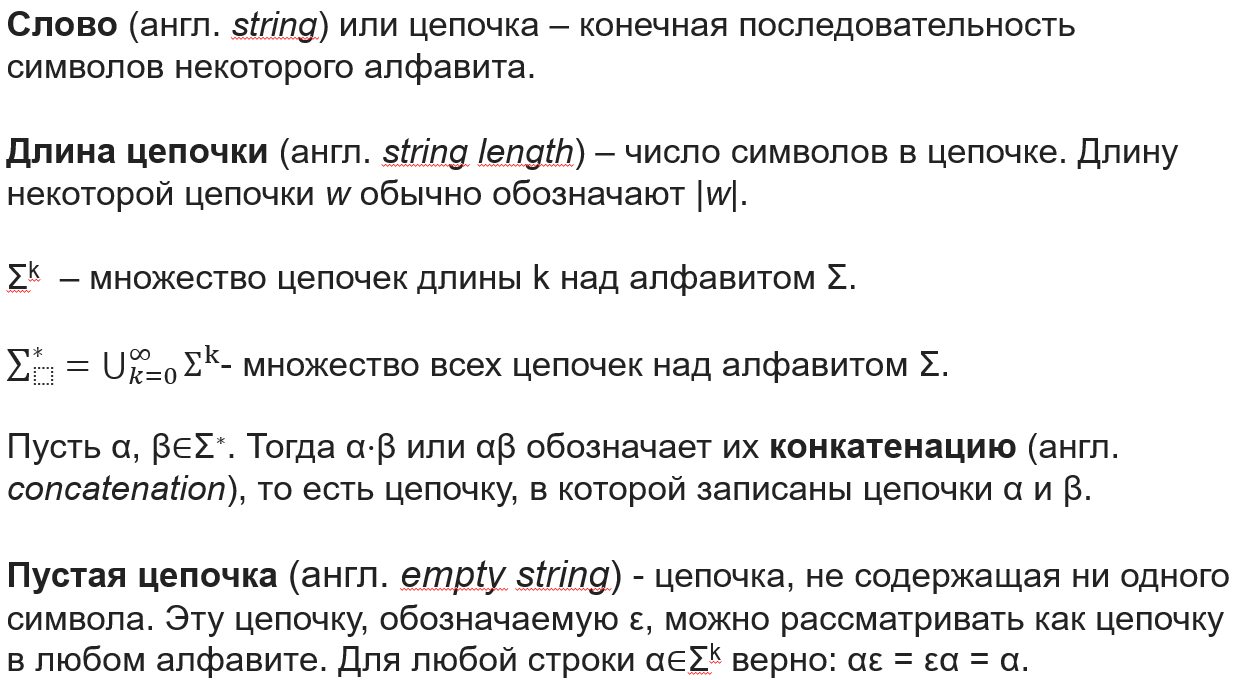
Σ={0,1} – бинарный или двоичный алфавит.​

Σ={a,b,…,z} – множество строчных букв английского алфавита.​

Σ={0,1,2,…,9} – алфавит цифр.​

Σ={⋅,−} – алфавит, лежащий в основе азбуки Морзе.​

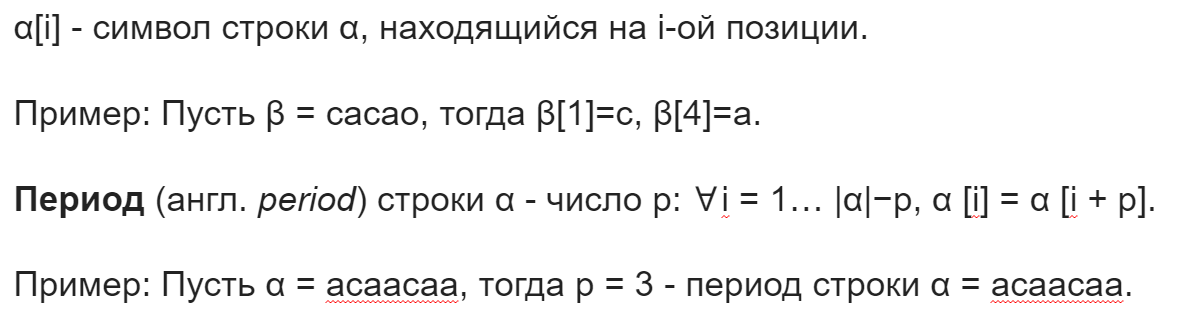
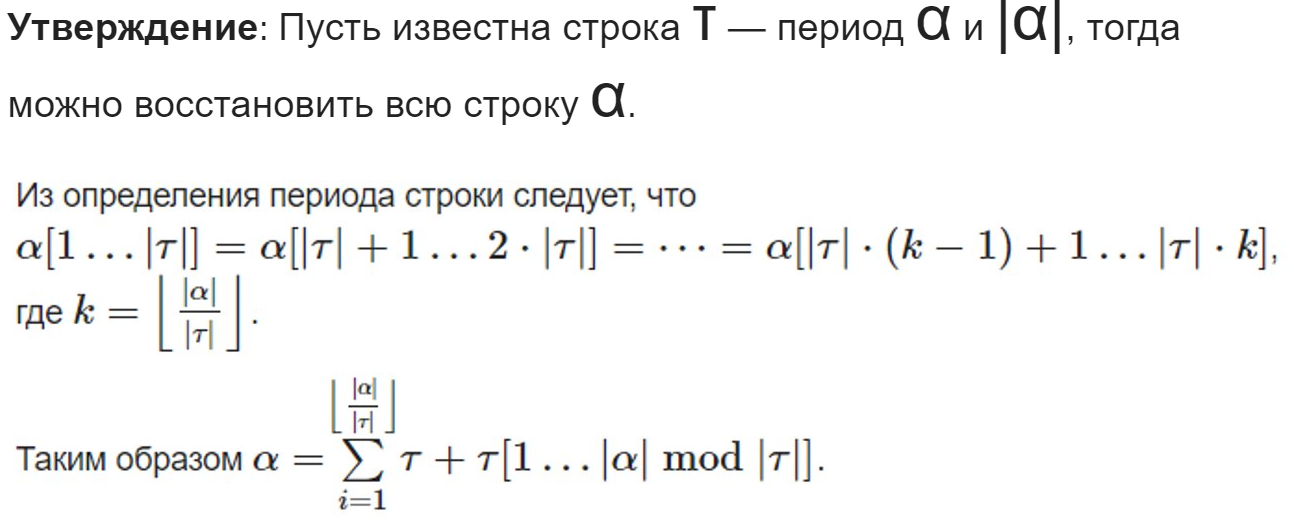
Нотные знаки​

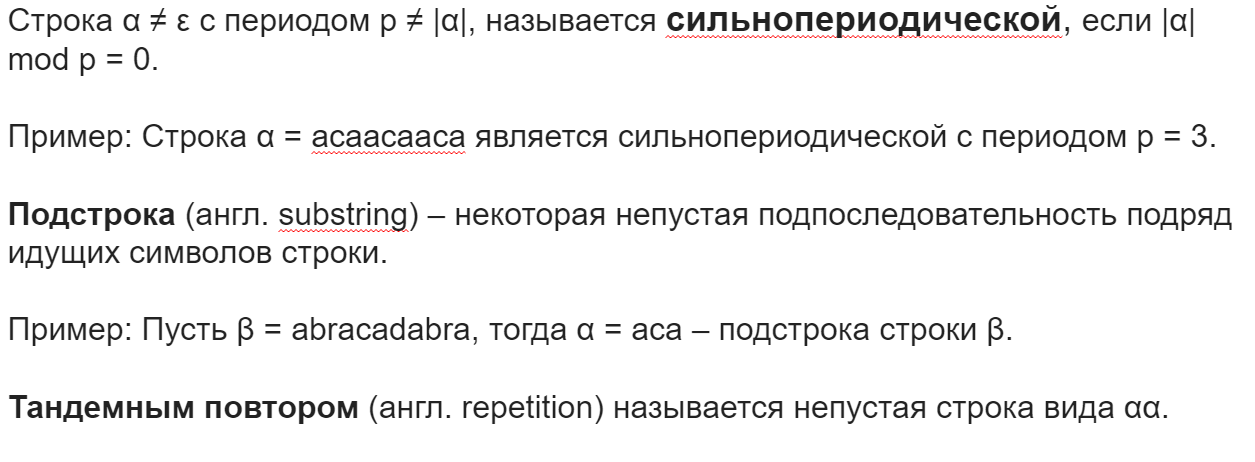
​ 

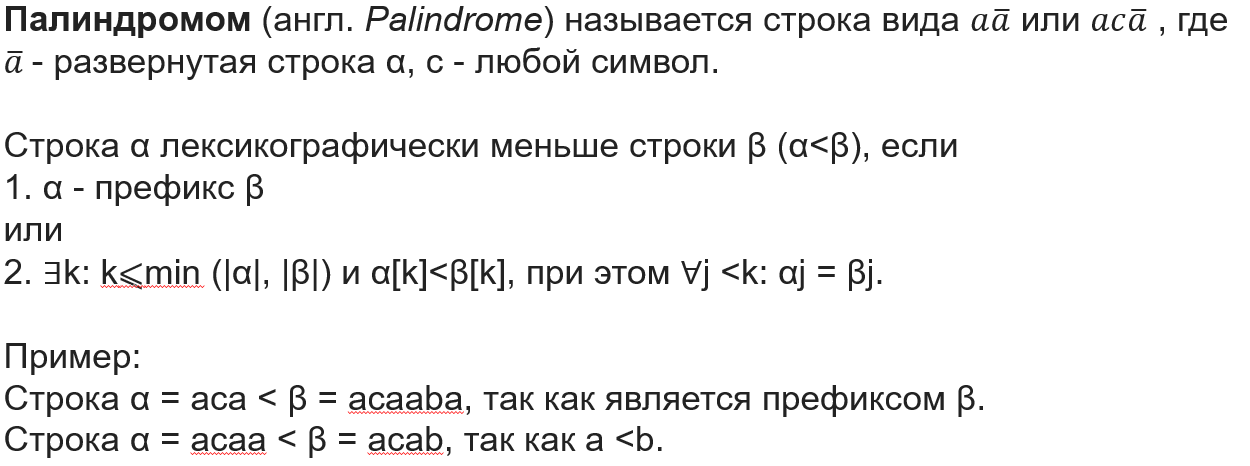
Множество слов над алфавитом Σ с операцией конкатенацией является **моноидом**

**Префикс** (англ. *prefix*) строки β - строка α: β = αγ.​​  
Пример: Пусть β = abracadabra, тогда α = abr - префикс β.​  
​  
**Суффикс**(англ. *suffix*) строки β - строка α: β = γα.​​  
Пример: Пусть β = abracadabra, тогда α = bra - суффикс β.​  
​  
**Грань** или **Бордер** (англ. *circumfix*) строки β - строка α: β = γα = αη.​​  
Пример: Пусть β = abracadabra, тогда α = abra - грань β.​  
​

**ОТНОШЕНИЯ МЕЖДУ СТРОКАМИ**

​ 





# Классические алгоритмы поиска (линейный, поиск с барьером, бинарный).

Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

*Алгоритм* ***последовательного поиска***

Шаг 1. Полагаем, что *значение* переменной *цикла* i=0.

Шаг 2. Если *значение* элемента массива x[i] равно значению ключа key, то возвращаем *значение*, равное номеру искомого элемента, и *алгоритм* завершает работу. В противном случае *значение* переменной *цикла* увеличивается на единицу i=i+1.

Шаг 3. Если i<k, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается *значение* равное -1.

При наличии в массиве нескольких элементов со значением key данный *алгоритм* находит только первый из них (с наименьшим индексом).

int LinearSearch(int \*x, int k, int key){

int i = 0;

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

if ( x[i] == key )

break;

return i < k ? i : -1;

}

*Время выполнения* данного алгоритма поиска для вещественных чисел n/\varepsilon, где n – количество элементов *множества*, а \varepsilon – *точность*. *Поиск* на дискретном множестве из n элементов осуществляется в худшем случае за n итераций, а в среднем этот *алгоритм* требует n/2 итераций *цикла*. Следовательно, временная сложность *последовательного поиска* пропорциональна O(n). Никаких ограничений на порядок элементов в массиве данный *алгоритм* не накладывает.

Идея ***поиска с барьером*** состоит в том, чтобы не проверять каждый раз в цикле условие, связанное с границами *множества*. Это можно обеспечить, установив в данном множестве так называемый *барьер*. Под барьером понимается любой элемент, который удовлетворяет *условию поиска*. Тем самым будет ограничено изменение индекса.

*Выход* из *цикла*, в котором теперь остается только условие поиска, может произойти либо на найденном элементе, либо на барьере. Существует два способа установки барьера: дополнительным элементом или вместо крайнего элемента массива.

//описание функции последовательного поиска с барьером

int LinearSearchWithBarrier(int \*x, int k, int key){

x = (int \*)realloc(x,(k+1)\*sizeof(int));

x[k] = key;

int i = 0;

while ( x[i] != key )

i++;

return i < k ? i : -1;

}

Заметим, что *поиск* с барьером работает быстрее, но временная сложность алгоритма остается такой же O(n), где n – количество элементов *множества*. Гораздо больший интерес представляют методы, не только работающие быстро, но и реализующие алгоритмы с меньшей сложностью.

**Бинарный (двоичный, дихотомический) поиск** – это *поиск* заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого *множества* на две части таким образом, что искомый элемент попадает в одну из этих частей. *Поиск* заканчивается при совпадении искомого элемента с элементом, который является границей между частями *множества* или при отсутствии искомого элемента.

//описание функции бинарного поиска

int BinarySearch(int \*x, int k, int key){

bool found = false;

int high = k - 1, low = 0;

int middle = (high + low) / 2;

while ( !found && high >= low ){

if (key == x[middle])

found = true;

else if (key < x[middle])

high = middle - 1;

else

low = middle + 1;

middle = (high + low) / 2;

}

return found ? middle : -1 ;

}

В процессе работы алгоритма *бинарного поиска* размер фрагмента, где этот *поиск* должен продолжаться, каждый раз уменьшается примерно в два раза. Это обеспечивает сложность алгоритма пропорциональную O(log n), где n – количество элементов *множества*.

# Префикс-функция: понятие, эффективный алгоритм построения.

**Префикс** (англ. prefix) строки β - строка α: β = αγ.​​

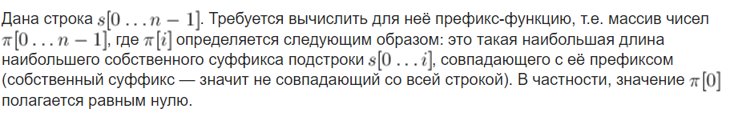
Пример: Пусть β = abracadabra, тогда α = abr - префикс β.​​

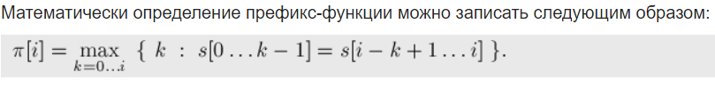
**Суффикс** (англ. suffix) строки β - строка α: β = γα.​​

Пример: Пусть β = abracadabra, тогда α = bra - суффикс β.​​

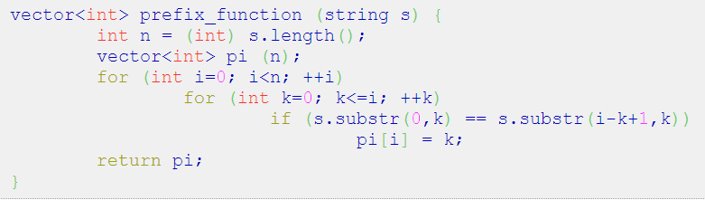
**Грань** или **Бордер** (англ. circumfix) строки β - строка α: β = γα = αη.​​

Пример: Пусть β = abracadabra, тогда α = abra - грань β.​

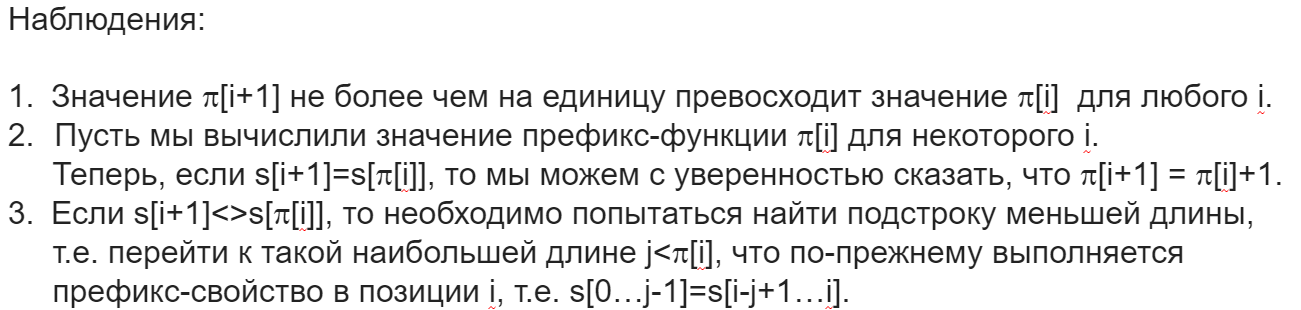


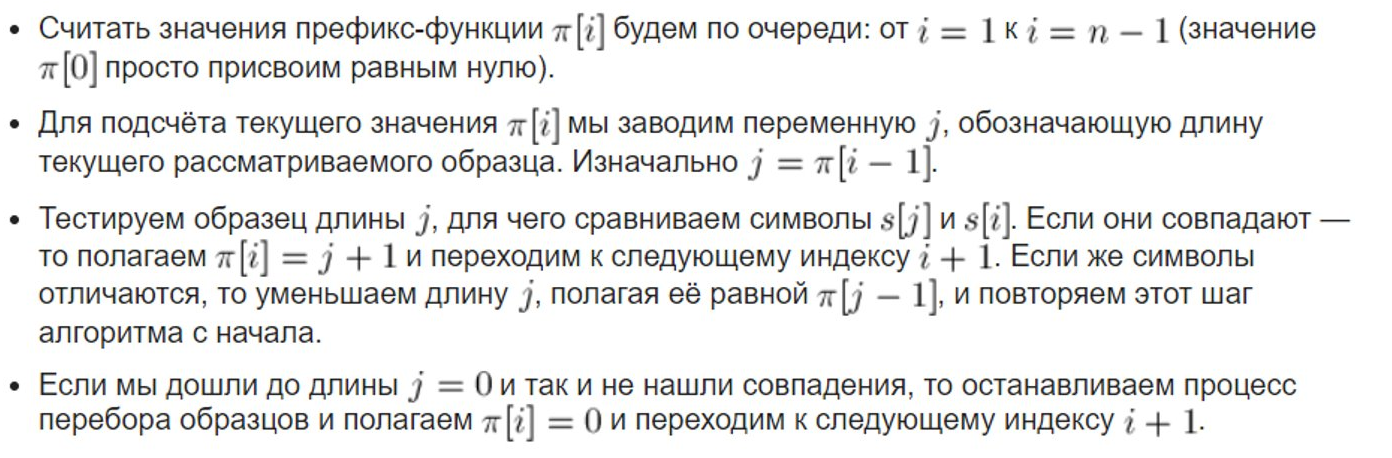
​

Простой алгоритм.



Эффективный алгоритм.





# Z-функция: понятие, эффективный алгоритм построения.

Пусть дана строка S дины n. ​​

Определение: Z-функция строки S – это массив длины n, i-ый элемент которого равен наибольшему префиксу подстроки строки S, начинающейся с позиции i, совпадающего с префиксом строки S.​​

Замечание: первый элемент Z-функции считают неопределенным. Мы будем считать, что он равен нулю.​​

Пример:​

S=‘aabbaabab’​

z=[0, 1, 0, 0, 3, 1, 0, 1, 0]​

a a b b a a b a b​

Наивный алгоритм нахождения значений Z-функции:​​

n = S.length(); ​

z = 0;​

for i=1..n-1:​

while (i + z[i] < n && ​

S[z[i]] == S[i+z[i]])​

z[i]++;​

return z;​

Эффективный алгоритм нахождения значений Z-функции:​​

Идея - максимально использовать уже вычисленные значения z-функции​

Определение: z-блоком называется подстрока, совпадающая с префиксом строки S.​

Т.о. Z[i] – наибольший z-блок, начинающийся в позиции i и заканчивающийся в позиции i+z[i]-1.​

Пусть left и right – координаты самого правого на данный момент z-блока, т.е. z-блока c максимальной границей right.​

Пусть мы нашли первые i-1 значения z-функции. Необходимо найти i-ое.​

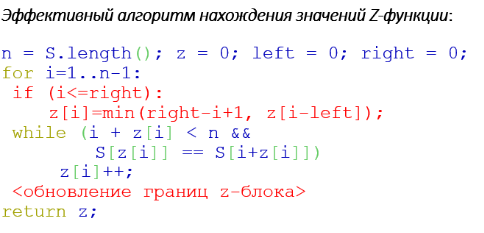
​

Возможны два случая:​

1 случай: i>right, тогда z[i] ищется наивным алгоритмом​

2 случай: i<=right, ​тогда можно ​проинициализировать ​z[i] не нулем, а большим значением, используя уже найденные значения z[i].​

Заметим, что S[left..right]=S[0..right-left], следовательно, в качестве начального значения z[i] можно взять z[i-left], но данный z-блок не должен выходить за пределы строки, т.е. в качестве начального значения z[i] можно взять min(right-i+1, z[i-left]).​



Временная сложность данного алгоритма О(n), т.к. каждая итерация цикла while увеличивает правую границу текущего самого правого z-блока на единицу.​

Применение Z-функции​

​

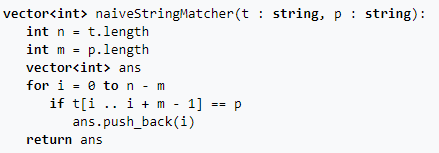
1. Поиск подстроки P в строке Т​​

2. Нахождение количества различных непустых подстрок данной строки​​

3. Сжатие строки, поиск минимального периода строки​

# Поиск подстроки в строке: наивный алгоритм поиска подстроки в строке, алгоритм Кнута-Морриса-Пратта, алгоритм поиска подстроки в строке на основе использования Z-функции.

В **наивном** **алгоритме** поиск всех допустимых сдвигов производится с помощью цикла, в котором проверяется условие t[s..s+m−1]=p для каждого из n−m+1 возможных значений s.

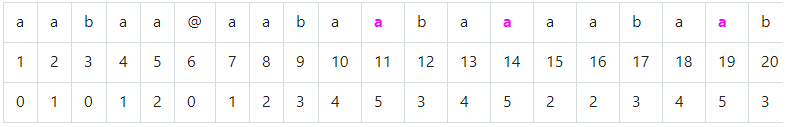


Алгоритм работает за O(m⋅(n−m)). В худшем случае m= n2, что даёт O(n24)=O(n2). Однако если m достаточно мало по сравнению с n, то тогда асимптотика получается близкой к O(n), поэтому этот алгоритм достаточно широко применяется на практике.

Алгоритм **Кнута-Морриса-Пратта** использует префикс-функцию для оптимизации сдвигов при наивном поиске.

*Дана цепочка (текст) S и образец (подстрока) P.*

* Построим строку T=P#S, где # - любой символ, не входящий в алфавит P и T.
* Найдем для строки T префикс-функцию p.
* Заметим, что для любого i значение префикс функции меньше либо равное |P| благодаря разделительному символу #.
* Если в какой-то позиции p[i]=|P|, где i>|P|, то в этой позиции начинается очередное вхождение образца в цепочку.



Применение **Z-функции​**

​

шаг 1. построить строку S=P#T​

шаг 2. найти z-функцию для строки S​

шаг 3. найти z[i]=P.length()​

# Поиск подстроки в строке: алгоритм Бойера-Мура-Хорспула.

Алгоритм Бойера — Мура — Алгоритм Бойера Мура поиска строки считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке.

Сначала строится таблица смещений для искомого шаблона. Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона.

Если последний символ шаблона и соответствующий ему при наложении символ строки не совпадают, то образец сдвигается относительно строки на величину, полученную из таблицы смещений. Причём символ берется из строки (а не из шаблона), соответствующий сдвиг находится в таблице. Производится сдвиг и снова начинается проверка с последнего символа.

Если же символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. Если же какой-то (не последний) символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на один символ вправо, и снова проверка снова начинается с последнего символа.

Весь алгоритм выполняется до тех пор, пока либо не будет найдено вхождение искомого образца, либо не будет достигнут конец строки.

Алгоритм основан на двух идеях :

1. Сравнение искомого слова с текстом начинается с последнего символа.

2. В случае несовпадения символа слово смещается на число знаков, получаемое из таблицы стоп-символов:

Таблица содержит список всех символов в шаблоне. В соответствие каждому символу ставится его порядковый номер, считая с конца строки. Если символ встречается несколько раз, то используется самое правое вхождение.

Пример

Для шаблона «abbad» таблица имеет следующий вид.

a b d

1 2 5

# Поиск подстроки в строке: алгоритм Рабина-Карпа.

# Поиск подстроки в строке: алгоритм Shift-And.

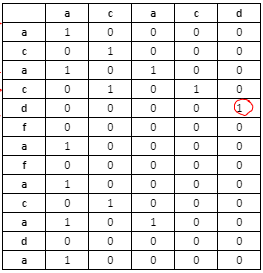
Дан текст Т (|T|=n) и подстрока Р (|P|=m).​ В данном алгоритме находятся не только все вхождения Р в Т, но и вхождения в всех возможных префиксов Р.​

Пример. Пусть текст Т=‘acacdfafacada’, а подстрока P=‘acacd’.​​

P имеет пять префиксов: a, ac, aca, acac, acacd.​​

Для каждой позиции текста необходимо знать, является ли она концом вхождения не только подстроки, но и каждого ее префикса.​​

Для отображения этой информации построим массив М[n,m], состоящий из нулей и единиц.​ M[i,j]=1, если j-ый префикс подстроки Р является суффиксом i-го фрагмента текста Т (Т[0..i]), в противном случае M[I,j]=0.​



M[i,j]=1 тогда и только тогда, когда​

​

1) в T входит и заканчивается ​

на предыдущей позиции меньший​

префикс ‘acac’, т. е. M[i-1,j-1]=1​

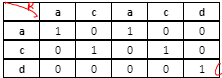
2) текущий символ текста T[i] ​

совпадает с j-ым символом ​

подстроки P, т. е. T[i]=P[j].​

Эффективная работа.

Нет необходимости хранить весь массив М, а достаточно хранить только его предыдущую строку (M[i-1]). Построим вектор Bit-Shift(M[i-1]) сдвигом вправо на одну позицию строки M[i-1] и приписав в начало единицу. Введем дополнительные характеристические векторы для символов подстроки (X[P[j],j]=1)



Тогда строка M[i] получается из M[i-1] следующим образом:​​

M[i]=Bit-Shift(M[i-1]) AND X[T[i]]​.

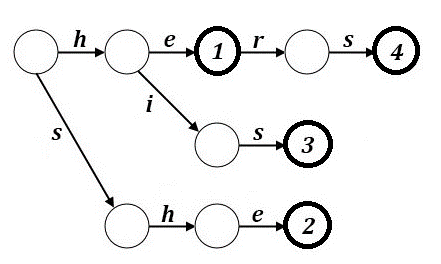
Работу с массивами можно заменить двоичными числами и битовыми операциями над ними, которые выполняются быстро.

# Поиск подстроки в строке: алгоритм Shift-Or.

# Бор: понятие, алгоритм построения.

**Бор** (англ. *trie*, луч, нагруженное дерево) – структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

**Пример**: Бор для набора образцов {he, she, his, hers}:



Пусть *n* – это мощность алфавита, а *k* – это количество слов во множестве слов. Бор храним как набор вершин, у каждой из которых есть метка, обозначающая, является ли вершина терминальной, символ, который привел в данную вершину и указатели (рёбра) на другие вершины или на NULL.

struct vertex:

    vertex next[n]

char ch

    bool isTerminal

**Алгоритм построения бора для множества слов M={P0, P1,…, Pk}**

Начало.

Шаг 1. Создадим дерево из одной вершины (в нашем случае корня).

Шаг 2. Добавление элементов в дерево.

Добавляем слова *Pi* одно за другим. Следуем из корня по рёбрам, отмеченным буквами из *Pi*, пока возможно.

Если *Pi* заканчивается в *v*, отмечаем вершину *v* как терминальную.

Если ребра, отмеченного очередной буквой *Pi* нет, то создаем новое ребро и вершину для символа строки *Pi*.

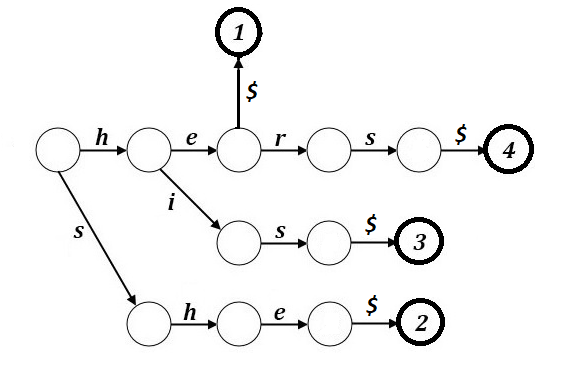
Конец.

# Сжатый бор: понятие, алгоритм построения.

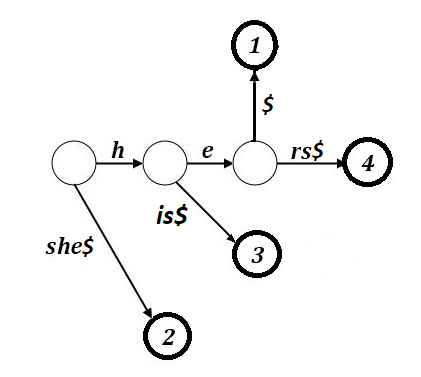
**Пример**: Бор для набора образцов {he, she, his, hers}:

**Модификация 1.** Только листы являются терминальными вершинами.

Для этого модифицируем исходное множество слов, добавив в конец каждого символ, не принадлежащий алфавиту, например, «$».



**Модификация 2.** Избавимся от вершин, имеющих только одного потомка.

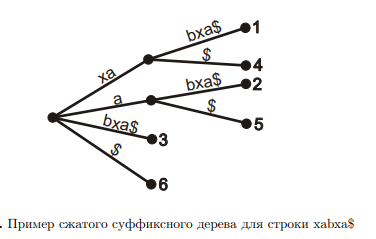


Получили сжатый бор.

# Поиск в боре.

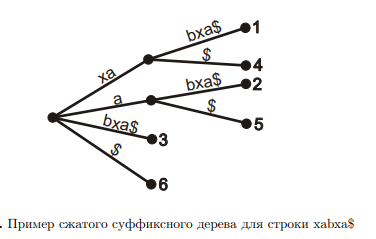
# Суффиксное дерево: понятие, наивный алгоритм построения, поиск в суффиксном дереве.

Суффиксное дерево — бор, содержащий все суффиксы некоторой строки (и только их). Позволяет выяснять, входит ли строка w в исходную строку t, за время O(|w|), где |w| — длина строки w. Каждый суффикс s[i..n-1] может быть прочитан на пути из корня до какого-нибудь листа и, наоборот, каждая строка, прочитанная на пути из корня до какого-нибудь листа, является суффиксом s.



# Суффиксное дерево: понятие, алгоритм Э. Укконена.

Суффиксное дерево — бор, содержащий все суффиксы некоторой строки (и только их). Позволяет выяснять, входит ли строка w в исходную строку t, за время O(|w|), где |w| — длина строки w. Каждый суффикс s[i..n-1] может быть прочитан на пути из корня до какого-нибудь листа и, наоборот, каждая строка, прочитанная на пути из корня до какого-нибудь листа, является суффиксом s.



<http://brenden.github.io/ukkonen-animation/> - визуализатор

Алгоритм Укконена (англ. Ukkonen's algorithm) — алгоритм построения суффиксного дерева для заданной строки s за линейное время.

Алгоритм Укконена строит последовательность неявных суффиксных деревьев, последнее из которых преобразуется в настоящее суффиксное дерево строки S.

# Суффиксный массив: понятие, наивный алгоритм построения, поиск в суффиксном массиве.

Cуффиксным массивом (англ. suffix array) строки s[1..n] называется массив suf целых чисел от 1 до n, такой, что суффикс s[suf[i]..n] — i-й в лексикографическом порядке среди всех непустых суффиксов строки s.

Как построить наивно:  
Выписать все пары <суффикс T[i:], i>.  
Упорядочить по T[i:] в лексикографическом порядке.  
Получившаяся перестановка – суффиксный массив.

Пример построения: T=abacaba  
<abacaba, 0>  
<bacaba, 1>  
<acaba, 2>  
<caba, 3>  
<aba, 4>  
<ba, 5>  
<a, 6>

Сортируем лексикографически:  
<a, 6>  
<aba, 4>  
<abacaba, 0>  
<acaba, 2>  
<ba, 5>  
<bacaba, 1>  
<caba, 3>

Суффиксный массив: [6, 4, 0, 2, 5, 1, 3]

Поиск при помощи бинарного поиска:  
Пример: строка(T)=abacaba, подстрока(P)=abac.  
Суфф массив (SA) = [6, 4, 0, 2, 5, 1, 3]

<a, 6> <- left  
<aba, 4>  
<abacaba, 0>  
<acaba, 2> <- middle  
<ba, 5>  
<bacaba, 1>  
<caba, 3> <- right

Abac < acaba, смещаем правую границу

<a, 6> <- left  
<aba, 4> <- middle  
<abacaba, 0>  
<acaba, 2> <- right  
<ba, 5>  
<bacaba, 1>  
<caba, 3>

Aba < abac, смещаем левую границу

<a, 6>   
<aba, 4> <- left  
<abacaba, 0> <- middle  
<acaba, 2> <- right  
<ba, 5>  
<bacaba, 1>  
<caba, 3>

Искомая подстрока является префиксом найденного суффикса.

Сложность: О(|P| \* log|T|).

# Суффиксный массив: понятие, алгоритм построения с временной сложностью O(n2logn).

Cуффиксным массивом (англ. suffix array) строки s[1..n] называется массив suf целых чисел от 1 до n, такой, что суффикс s[suf[i]..n] — i-й в лексикографическом порядке среди всех непустых суффиксов строки s.

# Суффиксный массив: понятие, алгоритм построения с временной сложностью O(nlogn).

Cуффиксным массивом (англ. suffix array) строки s[1..n] называется массив suf целых чисел от 1 до n, такой, что суффикс s[suf[i]..n] — i-й в лексикографическом порядке среди всех непустых суффиксов строки s.