УНИВЕРСИТЕТ ИТМО

Поиск подстрок
Конечные автоматы

Лектор

Ткаченко Данил Михайлович

Применение

- **Поиск подстроки в строке** — одна из простейших и чрезвычайно важных задач **поиска** информации.
- **Применение:**
  - Редактирование текста
  - Поиск образцов (в молекулах ДНК, например)
  - Поиск страниц в сети
  - СУБД
  - …

Обозначения

- **T[1…n]** – текст (text/haystack), n = len(T)
- **P[1…]** – образец (pattern/needle), m = len(P)

m <= n

- ∑ - алфавит, содержит символы, составляющие текст
- $\sum^*$- все возможные комбинации строк из алфавита
- **P** встречается в тексте со сдвигом **S**, если:
  - $0 \leq S \leq n - m$
  - T[S+1…S+m] = P[1…m]

  Тогда **S** – <u>допустимый</u> сдвиг
- Символьные массивы = строки
- Элементы массивов из <u>конечного алфавита</u>

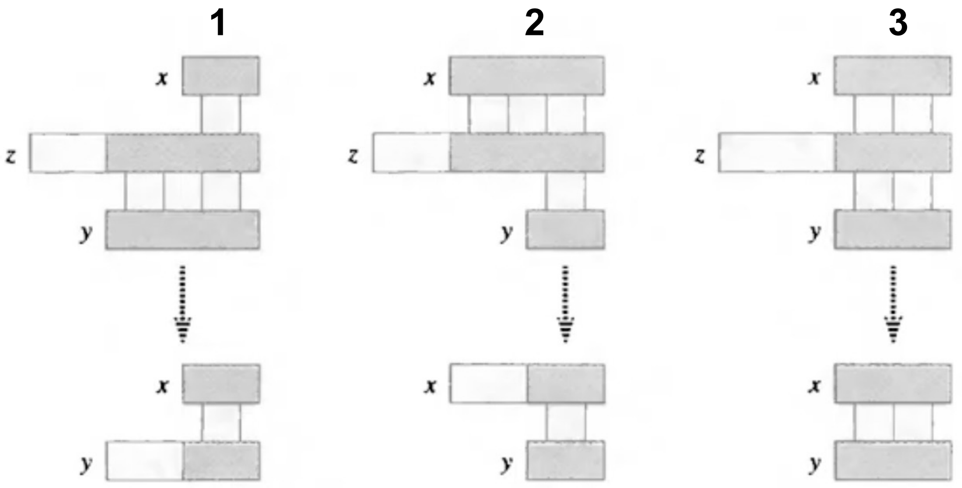
Обозначения

- $\boldsymbol{\varepsilon}$ – пустая строка
- $|\boldsymbol{x}|$ - длина строки **x**
- **xy** – конкатенация строк **x** и **y** длины $|\boldsymbol{x}| + |\boldsymbol{y}|$
- $\boldsymbol{\omega}$ префикс **x**, если **x** = $\boldsymbol{\omega}$**y**, обозначение $\boldsymbol{\omega}[\boldsymbol{x}$
- $\boldsymbol{\omega}$ суффикс **x**, если **x** = **y**$\boldsymbol{\omega}$, обозначение $\boldsymbol{\omega}]\boldsymbol{x}$

Лемма о перекрывающихся суффиксах

Пусть x, y, z – строки: x]z, y]z и если:

1. $|x| \leq |y|$, то x]y
2. $|x| \geq |y|$, то y]x
3. $|x| = |y|$, то x = y



УНИВЕРСИТЕТ ИТМО

Простейший алгоритм поиска подстрок

```
vector<int> naiveStringMatcher(t : string, p : string):
   int n = t.length
   int m = p.length
   vector<int> ans
   for i = 0 to n - m
      if t[i .. i + m - 1] == p
         ans.push_back(i)
   return ans
```

Просмотр каждого символа текста, откуда может начинаться подстрока, дает O(n-m)

Сравнение каждого символа текста с символом паттерна, что дает O(m)

Для каждого символа, являющимся потенциальным началом паттерна, будут просмотрены все символы паттерна, это и даст сложность в O(m*(n-m))

- Находит все вхождения строки **p** в **t** и возвращает массив позиций, откуда начинаются вхождения
- Алгоритм работает за $O\big(m*(n-m)\big)$ в среднем. В худшем случае $m \approx \frac{n}{2}$, что дает $O(n\text{^}2)$
- Неэффективен, так как информация о предыдущих сравнениях никак не учитывается

ITsMOre than a UNIVERSITY

Полиномиальная хеш-функция

Проблема – долгое сравнение на равенство строк

Решение - сравнение на равенство хешей этих строк

$h(x) \neq h(y) \;\; => \;\; x \neq y$

Полиномиальная хеш-функция

Проблемы:

1. Ложное совпадение строк при равенстве хешей
2. Долгий подсчет хеш-функции

Решение – полиномиальная хеш-функция

$$hash(p) = (P_0 X^{m-1} + P_1 X^{m-2} + \dots + P_{m-2} X + P_{m-1}) \bmod k$$

Константы:

- **k** – большое простое число
- **X** – взаимно простое с **k**

Полиномиальная хеш-функция

Пример: $X = 2 \quad k = 5 \quad \sum = [a \ldots z]$

hash(“computer”) $= (2 \cdot 2^7 + 14 \cdot 2^6 + 12 \cdot 2^5 + 14 \cdot 2^6 + 12 \cdot 2^5 + 15 \cdot 2^4$
$+ 20 \cdot 2^3 + 14 \cdot 2^6 + 19 \cdot 2^2 + 14 \cdot 2 + 17) \bmod 5 = \mathbf{2}$

Проблема:

- **k** – маленькое, поэтому велика вероятность совпадения хеш-кодов разных строк

Решение:

- **k** – большое простое число
- **X** – взаимно простое с **k**

Вычисление полиномиальной хеш-функции

Итеративное вычисление

- $hash(p) = (P_0X^{n-1} + P_1X^{n-2} + \dots + P_{n-2}X + P_{n-1}) \bmod k$
- $hash(p + c) = (P_0X^{n} + P_1X^{n-1} + \dots + P_{n-2}X^2 + P_{n-1}X + c) \bmod k$
  $= (hash(p) \cdot X + c) \bmod k$

Пример: X = 2, k = 5, ∑ = [a…z], T = hash

- $h(h) = 7 \bmod 5 = 2$
- $h(ha) = (2 * 2 + 0) \bmod 5 = 4$
- $h(has) = (4 * 2 + 18) \bmod 5 = 1$
- $h(hash) = (1 * 2 + 7) \bmod 5 = 4$

Время работы:

- **O(m)** – предварительный подсчет хеша паттерна
- **O(1)** – добавление символа к паттерну и пересчет хеша

Вычисление полиномиальной хеш-функции

Пересчет хеш-функции при сдвиге на 1

- $H_i = \ hash(T[i \dots i+m])$

  $H_i = (t_i X^{m-1} + \boxed{t_{i+1} X^{m-2} + \dots + t_{i+m-2} X + t_{i+m-1}}) \ mod\ k$
- $H_{i+1} = \ hash(T[i+1 \dots i+m+1])$

  $H_{i+1} = \boxed{(t_{i+1} X^{m-1} + t_{i+2} X^{m-2} + \dots + t_{i+m-1} X} + t_{i+m}) \ mod\ k$

Сравнивая выделенные выражения, получаем:

- $H_{i+1} = (H_i X - t_i X^m + t_{i+m}) \text{ mod k}$

Время работы:

- **O(m)** – подсчет первоначального хеша, который работает за длину рассматриваемой строки.
- **O(1)** – при сдвиге умножаем имеющийся хеш на X, отнимаем код первого в строке символа и прибавляем код добавленного символа при сдвиге, берем остаток по модулю k. Эти 3 операции работают за константное время.

Алгоритм Рабина-Карпа

- **Подсчет значений хеш-функции для всех подстрок**

$BuildH(T, P)$

n = T.length

m = P.length

H[0] = hash(T[0…m]) O(m)

for i = 0 … n-m-1

    H[i+1] = $(H[i] * X - T[i] * X^m + T[i+m])\ mod\ k$ O(n-m-1)

O(1)

O(m) + O(n-m-1) = O(n)

Алгоритм Рабина-Карпа

- **Подсчет значений хеш-функции для всех подстрок**

1. Подсчет хеша первоначальной подстроки длины паттерна, работает за длину паттерна – за **O(m)**
2. Цикл просматривает все символы, с которых может начинаться паттерн, работает за **O(n-m-1)**
3. Изменение хеша работает за **O(1)**, так как для этого производятся 3 операции, работающие так же за константное время

В итоге:

- Подсчитали первоначальный хеш подстроки длины паттерна
- Основываясь на этих данных, подсчитали хеш всех остальных подстрок длины паттерна
- Имеем сложность в **O(m) + O(n-m-1) = O(n)**

Алгоритм Рабина-Карпа

- **Поиск вхождений образца P в строку T**

Rabin_Karp(T, P):

$H_p = hash(P)$ ← O(m) Хеш-функция для паттерна

$BuildH(T, P)$ ← O(n) Подсчет хешей для всех подстрочек

```
for i = 0 … n – m:
    if H[i] != H[p]:
        continue
    ok = true
    for j = 0 … m – 1:
        if T[i+j] != P[j]:
            ok = false
            break
    if ok:
        print "Образец найден по индексу" i
```

1

$H_i[\dots]\ mod\ k$
$H_p[\dots]\ mod\ k$

$O(\frac{m}{k})$

O($\frac{m(n-m)}{k}$),
k - очень большое

O($\frac{m(n-m)}{k}$) + O(m) + O(n) = O(n+m)

Алгоритм Рабина-Карпа: асимптотика

1. Какова вероятность того, что данный цикл будет выполнен зря, то есть мы увидели совпадение по значениям хешей подстроки и паттерна, начали сравнивать их посимвольно и оказалось, что какие-то символы не совпали

- Рассмотрим 2 различных полинома $\mathbf{H_i(\ldots)}$ и $\mathbf{H_p(\ldots)}$ длины m, значения хешей которых по модулю **k** равны:
  - Вероятность такого совпадения хешей равна $\frac{1}{k}$
  - При таком совпадении нам потребуется посимвольное сравнение, которое в худшем случае потребует **m** сравнений
  - В самом худшем случае такая ситуация возникает каждый раз, когда проверяется очередной символ текста, а так как мы просматриваем все символы, с которых может начинаться паттерн, то будут просмотрены **n-m** символов текста
- В итоге: вероятность того, что нам потребуется проверить **m** символов **n-m** раз равна $\frac{m(n-m)}{k}$
- Делая **k** достаточно большим, мы сведем данную вероятность к минимуму, тем самым добьемся асимптотики алгоритма в **O(n+m)**

Z-функция

- **Z-функция от строки S и позиции x -** длина максимального префикса подстроки, начинающейся с позиции x в строке S, который одновременно является и префиксом всей строки S
  $Z[i](s) = \max k \mid s[i \ \ldots \ i+k] = s[0 \ldots k]$
- Тривиальный и эффективный виды алгоритма
- Пример строки и ее Z-функции:

string	a	b	a	c	a	b	a
Z(string)	7	0	1	0	3	0	1

Z-функция: тривиальный подход

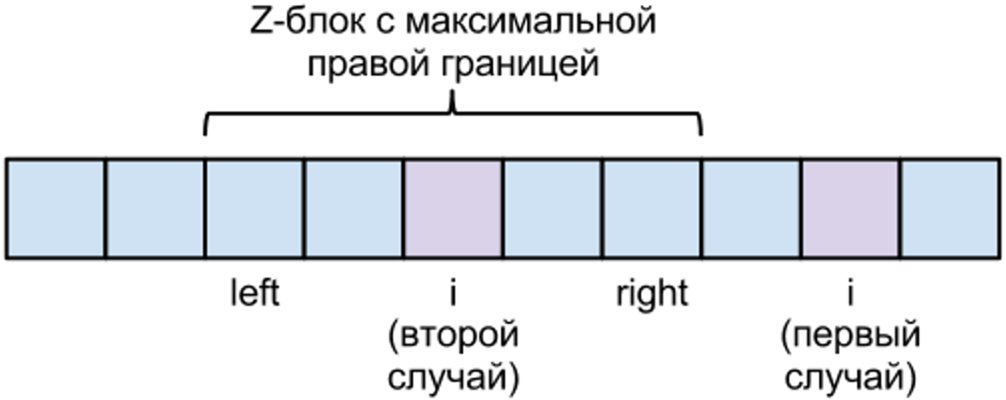
- Простая, наивная реализация, принцип которой заключается в переборе ответа для каждой i-ой позиции до тех пор, пока либо не обнаружим несовпадение, либо не дойдем до конца строки:

```
int[] zFunction(s : string):
  int[] zf = int[n]
  for i = 1 to n − 1                                      ← O(n)
    while i + zf[i] < n and s[zf[i]] == s[i + zf[i]]      ← O(n)
      zf[i]++
  return zf
```

$O(n^2)$

Z-функция: эффективный подход

- Назовем подстроку с началом в позиции i и длиной Z[i] Z-блоком. Будем поддерживать координаты left и right (изначально оба равны 0) - начало и конец соответственно - Z блока такого, что его позиция конца right максимальна (т.е. из всех обнаруживаемых блоков храним тот, что заканчивается правее всего).
  В некотором смысле, r - такая граница, что до нее строка уже была “просканирована” алгоритмом, после нее - еще нет
- Тогда, если текущий индекс, для которого мы хотим посчитать значение Z-функции - i, то мы имеем два случая:
  1) i > right, т.е. текущая позиция лежит за пределами того, что мы уже успели обработать
  2) i <= right, т.е. текущая позиция лежит внутри того, что мы уже успели обработать



Z-функция: эффективный подход

- 1) В случае, когда i > right: будем искать z[i] тривиальным методом, сравнивая символы на позициях j и i+j (т.е. просматриваем оставшуюся часть строки и первые символы). Как только на каком-то j у нас не соблюдается равенство s[j] = s[i + j], то мы можем утверждать, что z[i] = j (т.к. значение z[i] мы искали наивно), а границы теперь равны:
  left = i, right = i + j - 1

- 2) В случае, когда i <= right: мы можем использовать уже подсчитанные предыдущие значения Z-функции. Для этого заметим, что подстроки s[l..r] и s[0…r-l] совпадают, значит, для начала, в качестве z[i] мы можем взять соответствующее ему значение z[i - l]. Однако мы обязаны проверить, что это значение не является слишком большим, т.е. что оно “не вылезает” за пределы границы right. Поэтому проверяем:
  z[i] = min(right - i + 1, z[i - left])
  После этого шага мы снова действуем тривиальным методом, т.к. после границы right может обнаружиться продолжение нашего Z-блока, предугадать которое только лишь предыдущими значениями Z-функции мы не могли

Z-функция: реализация, асимптотика

- Что мы имеем в итоге? Оба случая приводят к использованию наивного метода, просто между собой они отличаются лишь начальным значением z[i], от которого мы дальше начинаем наивный поиск
- Реализация:

```
z_func (string s) {
  n = s.len
  z[n]
  left = right = 0
  for i = 1 to n - 1
    if (i <= right):
      z[i] = min(right - i + 1, z[i - left])
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
      z[i] += 1
    if (i + z[i] - 1 > right)
      left = i
      right = i + z[i] - 1
  return z
```

- Изначально z заполняем нулями, а границы делаем равными 0. Внутри цикла определяем сначала начальное значение z[i]: либо остается 0, либо вычисляется через уже известные. Затем запускаем тривиальный алгоритм, после которого двигаем границы, если это требуется (если был найден новый Z-блок)
- Асимптотика: O(n)
- Каждая итерация в цикле while увеличивает правую границу на 1. Так как в итоге right не может оказаться больше n - 1, получаем, что всего while сделает не более n - 1 итераций. Остальные операции - константные, выполняемые O(n) раз, значит в итоге получаем O(n)

Префикс-функция

- **Префикс-функция** – массив длины максимального собственного (не совпадающего со строкой) префикса подстроки, являющимся ее суффиксами
- Пример:
  - pref(”ABABAB”) = “ABAB”
  - pref(“ABCAAB”) = “AB”
  - pref(“AB”) = “”
- pi[i] = $|pref(P[0 \ldots i-1])|$
- Пример (итеративно рассматривается в приложенном файле):

P		A	B	A	C	A	B	A	B	A	C	B
pi	-	0	0	1	0	1	2	3	2	3	4	0

Вычисление и анализ сложности

BuildP(T)

```
i = 0, j = 0
while i < n:
    if T[i] = P[j]:
        pi[i+1] = j + 1
        i++, j++
    else:
        if j > 0:
            j = pi[j]
        else:
            pi[i+1] = 0
            i++
```

Анализ цикла while на следующем слайде

Анализ переменных цикла while:

1. Либо увеличились i и j на единицу
2. Либо уменьшилось j минимум на единицу
3. Либо увеличилось i на единицу

- Асимптотика: O(n)

Вычисление и анализ сложности

- В цикле **while** у нас нет ситуации, при которой бы не изменилось значение **i** или **j**. На каждой итерации цикла у нас как минимум меняется значение одного из итераторов
- Так как **i** не уменьшается, то максимальное количество раз, которое мы можем увеличить **i** равняется **n**
- Из этого следует, что затормозить цикл может только **j**
- Максимально **j** может уменьшаться на длину подстроки. Поэтому самый худший случай – уменьшение **j** на **1**
- Цикл while либо увеличивается **i** на **1**, либо оставляет неизменным столько раз, сколько было до этого шага совпадений, где за количество совпадений и отвечает **j**
- Пусть на какой-то итерации было **k** совпадений, тогда **i** увеличилось на **k**, **j** стало равно **k**
- Так как мы рассматриваем худший случай, то после этой итерации у нас не будет совпадений по префиксу и суффиксу **k** итераций
- Тогда **j** уменьшиться на **k**, а **i** останется неизменным
- В результате рассматрения данной подстроки за **k** итераций **i** увеличилось на **k**, затем за **k** итераций **j** уменьшилось на **k** и стало равно **0**
- Тогда на данную подстроку ушло **2k** итераций
- Так как **i** у нас ограничено **n**, то максимум **i** увеличиться на **n**, а так как в худшем случае **i** остается неизменным столько раз, сколько до этого было совпадений, то **j** уменьшится на это же количество, то есть на длину рассматриваемой подстроки => в сумме **j** будет уменьшаться **n** раз
- А значит всего итераций в худшем случае будет ≤ **2n** и следовательно сложность данного цикла **O(2n) = O(n)**

Алгоритм Кнута-Морриса-Пратта

KMP_Matcher(T, P):

```
BuildP(P)
i = 0, j = 0
n = T.length, m = P.length
while i < n && j < m:
    if T[i] = P[j]:
        i++, j++
    else:
        if j > 0:
            j = pi[j]
        else:
            i++
if j == m:
    return i - m
else:
    return -1
```

Предподсчет префикс-функции для паттерна
Работает за длину паттерна: O(m)

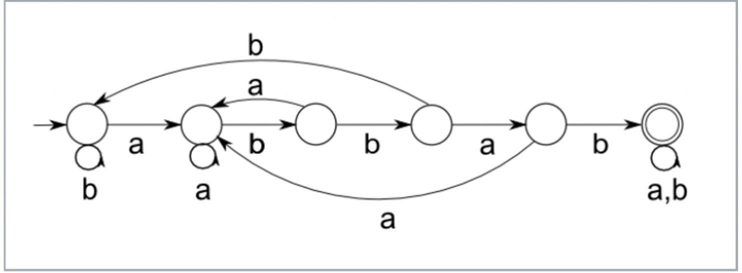
Нахождение паттерна в тексте
Работает за O(n)

Итоговая асимптотика:
O(n+m)

Конечный автомат: определение

Конечный автомат **M**:

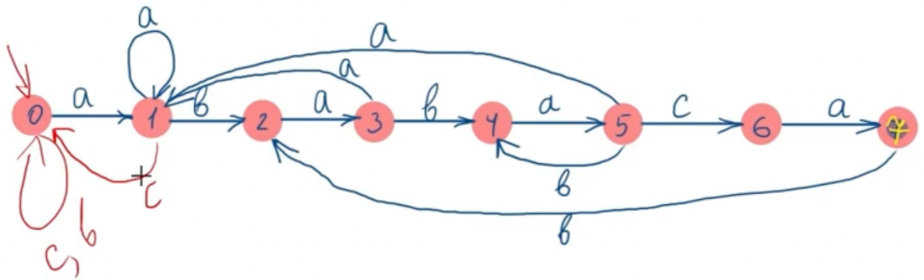
Пример



- **Q** - конечное множество состояний
- $\mathbf{q_0} \in \mathbf{Q}$ - начальное состояние
- $\mathbf{A} \subseteq \mathbf{Q}$ - множество допускающих состояний
- **Σ** - конечный входной алфавит
- **δ** - функция переходов автомата: **Q** x **Σ** –> **Q** (производит переход из одного состояния в другое засчет перехода по соответствующему символу из **Σ**)
- → - начальное состояние
- - завершающее(допускающее) состояние
- Как работает: начинает в $\mathbf{q_0}$ и считывает символы по одному

Конечный автомат: определение

- **ф** - функция конечного состояния **Σ^*–> Q:**
  ф(*ω*) состояние, в котором оказывается автомат **М** после сканирования ***ω*,**
  ***ω* -** слово
- **М** принимает ***ω*,** если **ф(*ω*) ∈ A**



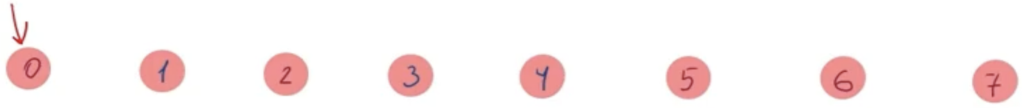
*из всех вершин есть возможность перейти по ВСЕМ символам алфавита. Здесь для уменьшения объема графа не показаны переходы по каждому неучтенному симоволу в начальное состояние

Пример создания автомата для строки-образца

- **P =** ababaca
- **Префикс функция:** π =

	a	b	a	b	a	c	a
-	0	0	1	2	3	0	1
0	1	2	3	4	5	6	7

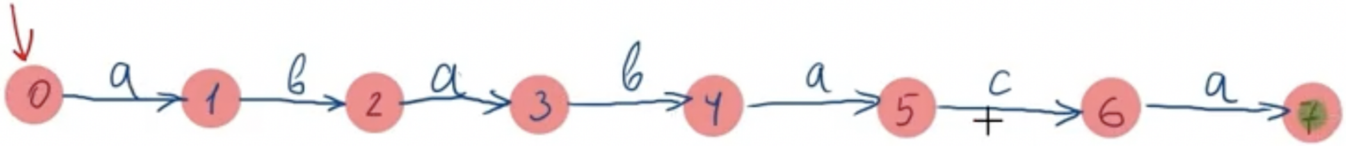
- Первоначальное представление автомата:



- Теперь необходимо подсчитать функции переходов

Подсчет функции переходов

- Перед подсчетом функции переходов для каждого состояния, заметим, что у нас уже существует идеальный сценарий, который заключается в следующем:
  - Видим **a** - идем в состояние **1**
  - Видим **b** - идем в состояние **2**
  - Видим **a** - идем в состояние **3**
  - Видим **b** - идем в состояние **4**
  - Видим **a** - идем в состояние **5**
  - Видим **c** - идем в состояние **6**
  - Видим **a** - идем в состояние **7**



УНИВЕРСИТЕТ ИТМО

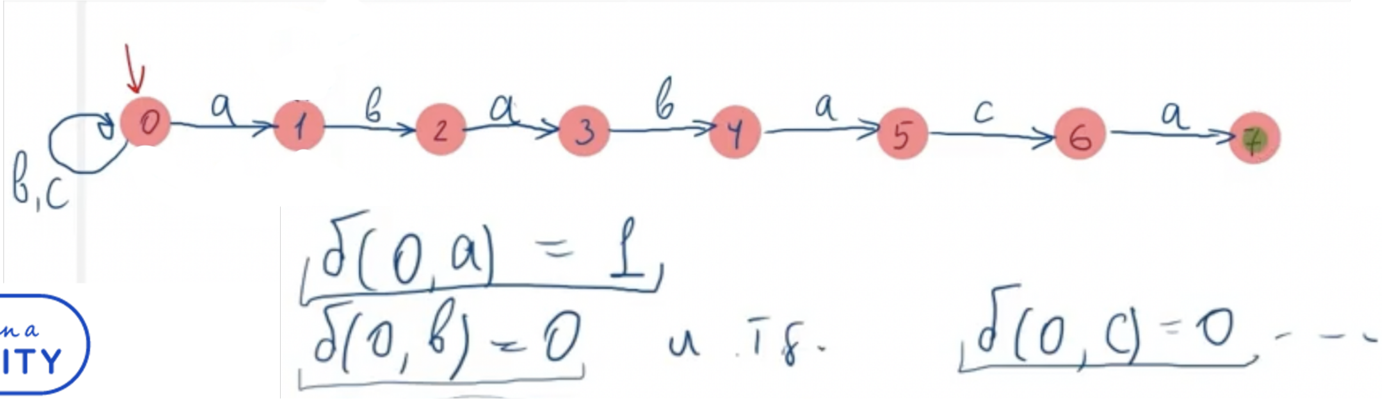
Подсчет функции переходов

- Предположим, что случилась ситуация, когда мы не увидели символ, позволяющий перейти в следующее состояния, тогда потребуется какой-то другой переход
- Данный переход рассчитывается с помощью префикс-функции, которая была посчитана ранее
- Для расчета требуется просмотреть значение префикс-функции под индексом, который соответствует состоянию, из которого должен произойти переход
- В ходе данного перехода автомат может перейти в любое состояние, предшествующего тому, из которого происходил данный переход
- Значение префикс-функции позволяет в подобных ситуациях не возвращаться каждый раз в нулевое состояния, а переходить в состояние, несущее уже некоторую информацию о части паттерна, который необходимо найти в тексте

ITsMOre than a UNIVERSITY

Подсчет функции переходов: 0 состояние

- Первоначально нам известно, что из **0** состояния мы можем попасть в **1** состояние пройдя по символу **a**
- Пройдя по символам **b** или **c** автомат останется в начальной(нулевом) состоянии
- Данную информацию мы берем за базу, благодаря которой сможем подсчитать функции переходов для остальных состояний

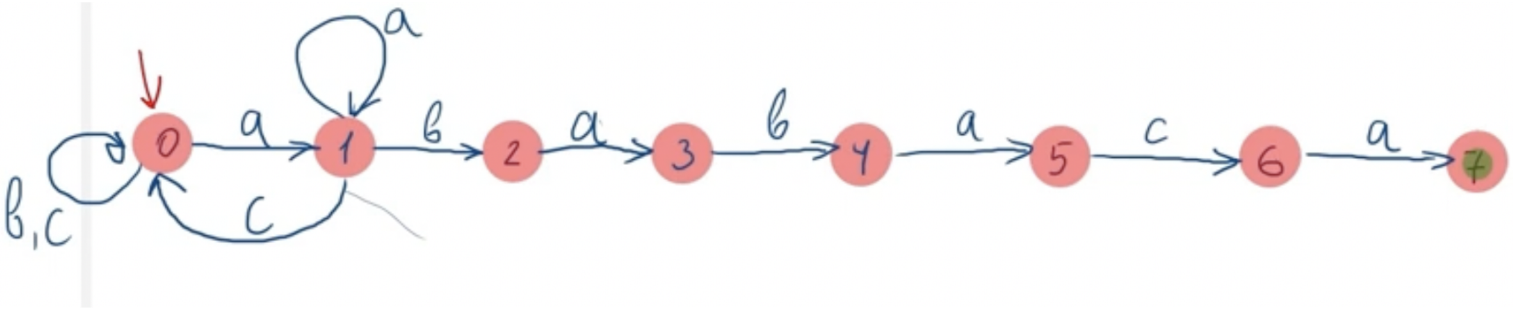


Подсчет функции переходов: 1 состояние

- Переход из состояния **1** в состояние **2** по символу **b** нам известен: $\delta(1, b) = 2$
- Требуется подсчитать функции переходов для символа **a** и **c**:

$$\delta(1, a) = \delta(\pi_q(1), a) = \delta(0, a) = 1$$

$$\delta(1, c) = \delta(\pi_q(1), c) = \delta(0, c) = 0$$

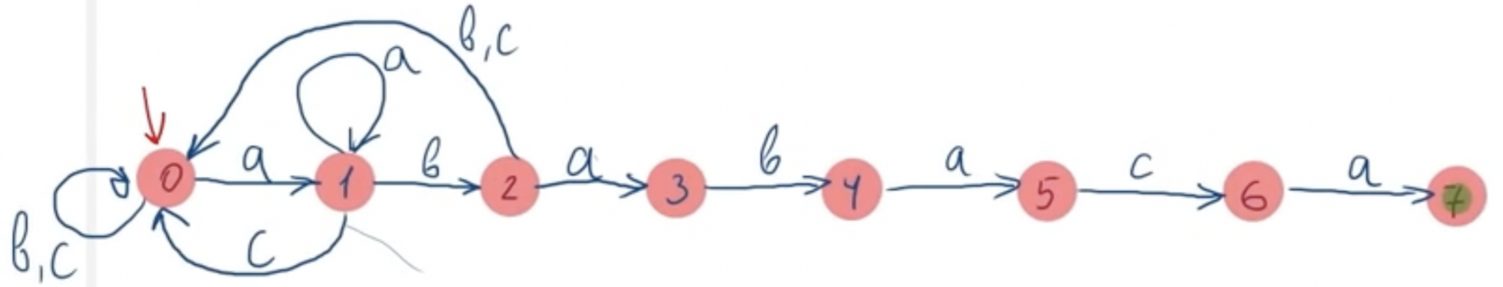


Подсчет функции переходов: 2 состояние

- Переход из состояния **2** в состояние **3** по символу **a** нам известен: $\delta(2,a) = 3$
- Требуется подсчитать функции переходов для символа **b** и **c**:

$$\delta(2,b) = (\pi_q(2), b) = \delta(0,b) = 0$$

$$\delta(2,c) = (\pi_q(2), c) = \delta(0,c) = 0$$

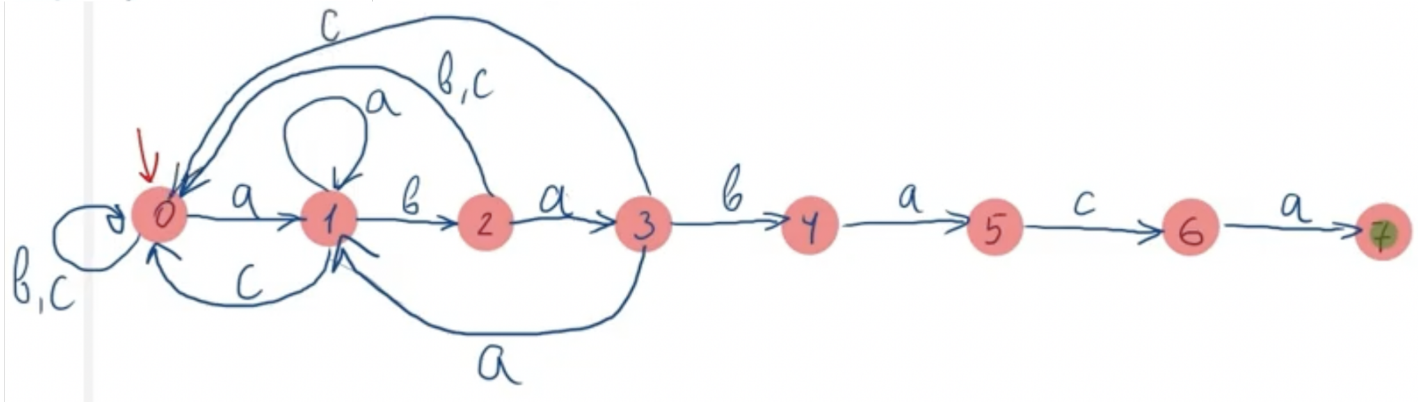


Подсчет функции переходов: 3 состояние

- Переход из состояния **3** в состояние **4** по символу **b** нам известен: $\delta(3, b) = 4$
- Требуется подсчитать функции переходов для символа **b** и **c**:

$$\delta(3, a) = \delta(1, a) = 1$$
$$\delta(3, c) = \delta(1, c) = 0$$

После 1 знака равенства было сразу подставлено значение префикс-функции под индексом 3

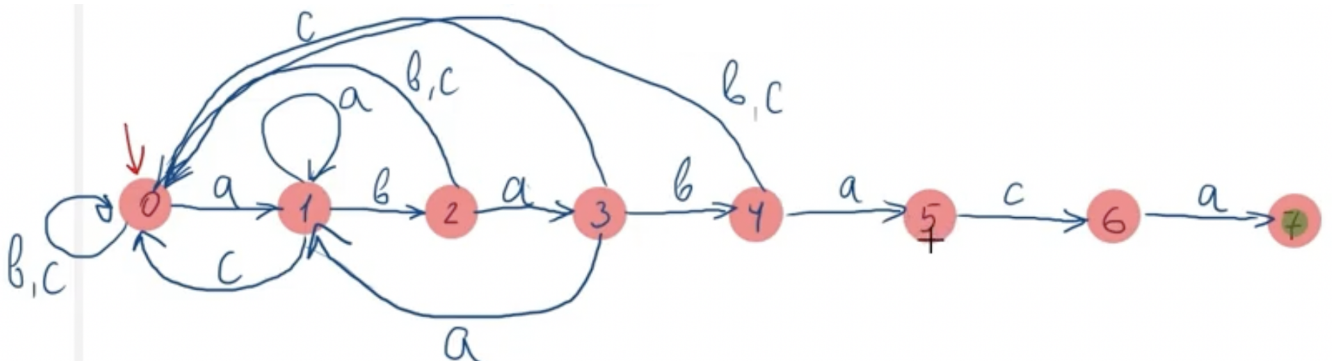


Подсчет функции переходов: 4 состояние

- Переход из состояния **4** в состояние **5** по символу **a** нам известен: **δ(4, a) = 5**
- Требуется подсчитать функции переходов для символа **b** и **c**:
- **δ(4, b) = δ(2, b) = 0**
- **δ(4, c) = δ(2, c) = 0**

Аналогичные действия:

- Берем значение префикс-функции под индексом текущего состояния
- Все переходы для состояния равного значению префикс-функции, взятой выше, уже найдены

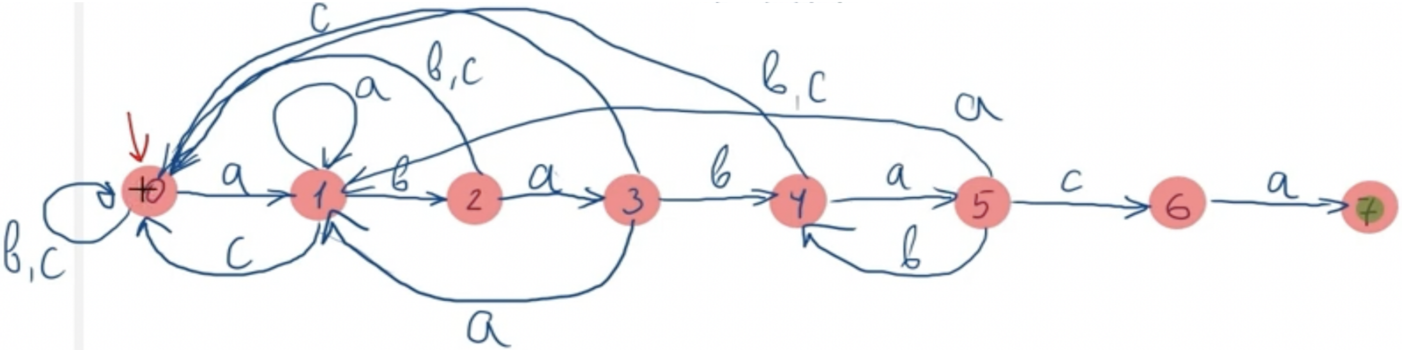


Подсчет функции переходов: 5 состояние

- Переход из состояния **5** в состояние **6** по символу **c** нам известен: **δ(5, c)** = **6**
- Требуется подсчитать функции переходов для символа **a** и **b**:
- **δ(5, a) = δ(3, a) = 1**
- **δ(5, b) = δ(3, b) = 4**

Аналогичные действия:

- Берем значение префикс-функции под индексом текущего состояния
- Все переходы для состояния равного значению префикс-функции, взятой выше, уже найдены

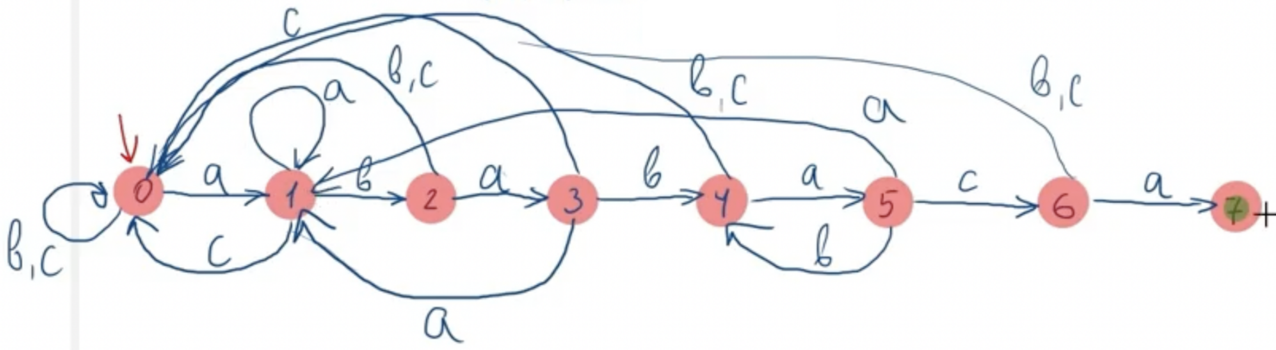


Подсчет функции переходов: 6 состояние

- Переход из состояния **6** в состояние **7** по символу **a** нам известен: **δ(6, a)** = **7**
- Требуется подсчитать функции переходов для символа **b** и **c**:
- **δ(6, b) = δ(0, b) = 0**
- **δ(6, c) = δ(0, c) = 0**

Аналогичные действия:

- Берем значение префикс-функции под индексом текущего состояния
- Все переходы для состояния равного значению префикс-функции, взятой выше, уже найдены

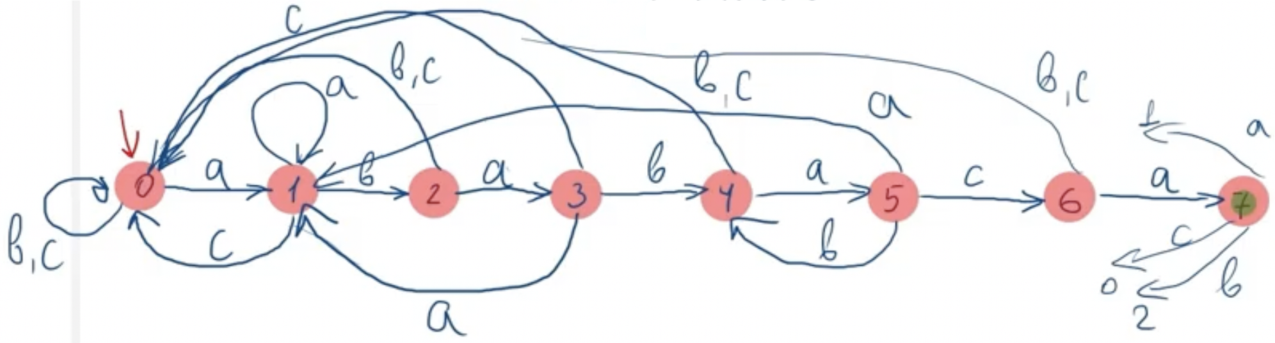


Подсчет функции переходов: 7 состояние

- **7** состояние - допускающее состояние
- Переходы такие же, как и у **1** состояния
- **δ(7, a) = δ(1, a) = 1**
- **δ(7, b) = δ(1, b) = 2**
- **δ(7, c) = δ(7, c) = 0**

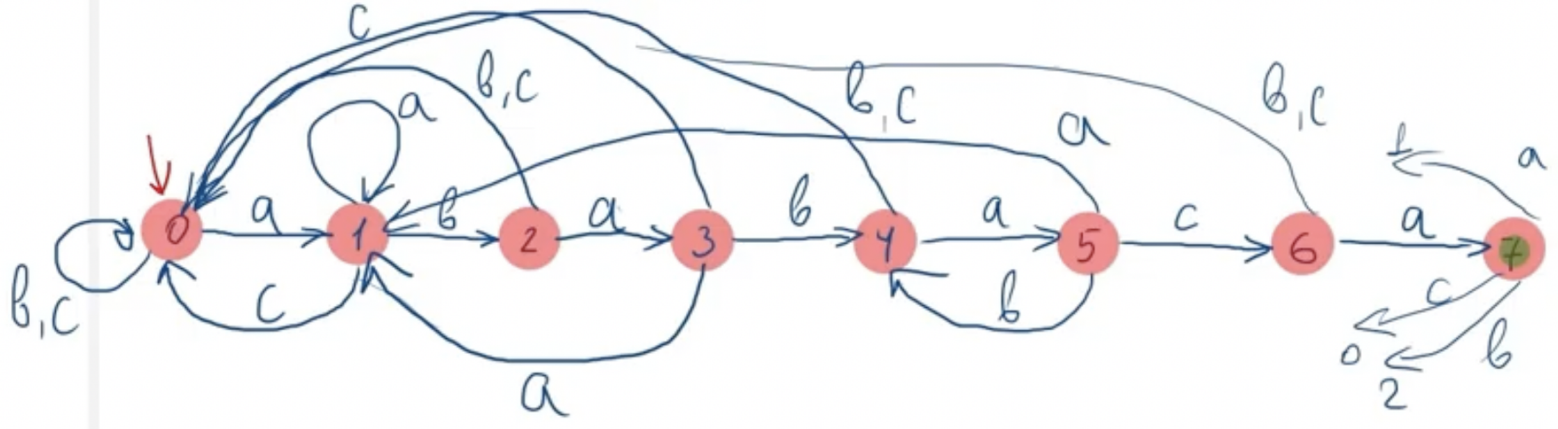
Аналогичные действия:

- Берем значение префикс-функции под индексом текущего состояния
- Все переходы для состояния равного значению префикс-функции, взятой выше, уже найдены



Автомат для строки-образца

- **P** = ababaca



Построение автомата для строки-образца: асимптотика

- Так как мы работаем с паттерном, то нам необходимо просмотреть каждый его символ: это занимает **O(m)**, **m** – длина паттерна
- Для каждого **i-го** символа паттерна **(i=0…m-1)** мы считаем функцию переходов
- Количество подсчетов для очередного символа зависит от размерности алфавита, так как для каждого состояния нам необходимо подсчитать функцию переходов по каждому символу
- В итоге: просмотрели все символы паттерна и для каждого подсчитали функции переходов
- Получаем **O(m**|∑|)

Поиск подстрок с помощью автомата

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n = T.length
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Образец найден со сдвигом" i − m
```

Функция перехода, которая возвращает номер состояния, в которое можно перейти по данному символу

Время работы:

- Просмотр всех символов текста – **O(n)**
- Подсчет функций переходов - **O(m|∑|)**
- В итоге имеем: **O(n + m|∑|)**

Поиск подстрок с помощью автомата

- Заходим в атомат по первому символу
- Пробуем по этому символу куда-то пойти, то есть совершить переход в другое состояние
- Если в результате перехода мы попали в допускающее состояния, то образец был найден со сдвигом
- Если мы не попали в допускающее состояние, то переходим к следующему символу текста и производим переход от него, проверяя возможность попадания в допускающее состояние
- Благодаря префикс-функции не придется каждый раз при непопадании в допускающее состояние перепроверять паттерн заново. Проверка начнется с состояния, соответствующего значению префикс-функции

Спасибо за внимание!

www.ifmo.ru

ITsMOre than a UNIVERSITY