Лабораторная №3. ***Работа со строками***

Строки в языке Java

Класс String является основным классом, предназначенным для хранения и обработки строк символов. Для создания экземпляров класса String может быть использован один из следующих **конструкторов**:

```
String()
String(String str)
String(StringBuffer strbuf)
String(char[] arr)
String(char[] arr, int first, int count)
```

Первый из них создаёт пустую строку, второй и третий копируют содержимое объектов классов String и StringBuffer в созданный объект. Последние два конструктора позволяют создать строку на основе символьного массива или его части. Кроме того, любая объектная ссылка типа String может быть проинициализирована посредством присвоения ей **строкового литерала**, например:

```
String filename = "data.txt";
```

Особенностью класса String является то, что экземпляры этого класса **не могут быть изменены** после их создания. Однако это не создаёт ограничений для их использования, поскольку все методы, которые должны были бы изменять строку, просто создают **новую модифицированную строку**, оставляя исходную без изменений. Поясним работу этого механизма на примере:

```
String s = "abcd";
s = s.toUpperCase();
```

Здесь метод toUpperCase() создаёт новую строку, содержащую последовательность символов "ABCD", и возвращает ссылку на эту строку, которая присваивается переменной s, старое значение переменной теряется. Исходная строка остаётся в неизменном виде и, поскольку на неё больше не осталось объектных ссылок, будет удалена

Основные методы класса String

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char[] toCharArray()	Получение строки в виде символьного массива
boolean equals(String str)	Сравнение строк на равенство
boolean equalsIgnoreCase(String str)	Сравнение строк без учета регистра
int compareTo(String str)	Лексикографическое сравнение строк
int compareToIgnoreCase(String str)	Лексикографическое сравнение строк без учета регистра
boolean startsWith(String prefix)	Проверка, начинается ли строка с заданной подстроки
boolean endsWith(String suffix)	Проверка, заканчивается ли строка заданной подстрокой
int indexOf(String subStr) **int** indexOf(String subStr, **int** fromIndex)	Поиск первого вхождения подстроки в строке с начала строки/с заданной позиции
int lastIndexOf(String subStr) **int** lastIndexOf(String subStr, **int** fromIndex)	Поиск последнего вхождения подстроки в строке с начала строки/с заданной позиции

String substring(**int** beginIndex, **int** endIndex)	Получение подстроки (символ endIndex не входит в подстроку!)
String substring(**int** beginIndex)	Получение хвоста строки
String concat(String str)	Конкатенация строк
String toUpperCase() String toLowerCase()	Преобразование строки к верхнему/нижнему регистру
String trim()	Удаление ведущих и завершающих пробелов в строке
String replace(String target, String replacement)	Замена подстроки другой строкой
boolean matches(String regex)	Проверка строки на соответствие регулярному выражению
String replaceFirst(String regex, String replacement) String replaceAll(String regex, String replacement)	Замена первой подстроки/всех подстрок, соответствующих регулярному выражению, заданной подстрокой
String[] split(String regex)	Разбиение строки на подстроки (разделители задаются регулярным выражением)

Преобразование к строке

- Класс String является в некотором смысле исключительным классом в Java, поскольку любой тип данных может быть преобразован к нему.

- Для примитивных типов такое преобразование даёт их естественное строковое представление, для объектов вызывается метод toString(), определённый в классе Object и, следовательно, присутствующий в любом классе Java.

Конкатенация строк

- Для строк определена операция **конкатенации**, обозначаемая знаком +.
- Это бинарная операция, один из аргументов которой должен иметь тип String. Она осуществляет **автоматическое преобразование** другого аргумента к типу String (если это необходимо) и слияние полученных строк. Это единственный случай, когда преобразование к строке осуществляется неявно.
- Существует также операция **конкатенации с присваиванием +=**, первый аргумент которой должен иметь тип String, а второй может быть произвольным. При выполнении операции он будет преобразован к типу String

Задание

Реализовать алгоритм поиска подстрок с помощью конечного автомата

Постановка задачи

- Пусть у нас есть две строки: текстовая строка T и шаблонная строка P. Надо **найти все вхождения P в T**.
- Если текст и шаблон состоят из n и m символов соответственно, то $m \le n$.
- Решением будут все **величины сдвигов** P относительно начала T, отмечающие, где шаблон располагается в тексте: шаблон P встречается в тексте со сдвигом s, если подстрока T, которая начинается с s+1, в точности такая же, как и шаблон P.
- Минимально возможный сдвиг — нулевой. Так как шаблон не должен выходить за пределы текста,

Конечный автомат

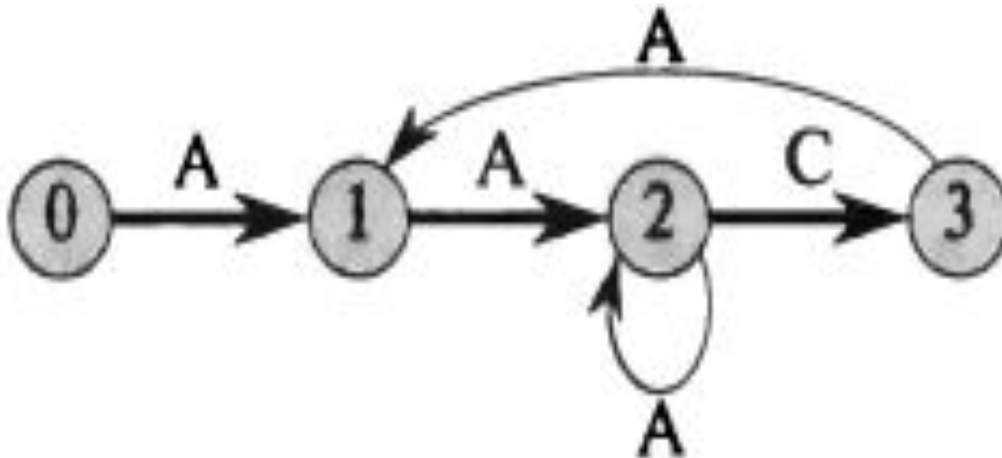
- КА — это набор некоторых состояний, а путь от состояния к состоянию основан на последовательности входных символов.

- КА начинает работу с определенного состояния и по одному получает входные символы. Основываясь на состоянии, в котором он находится, и полученном символе, конечный автомат переходит в новое состояние.

КА в случае поиска подстрок

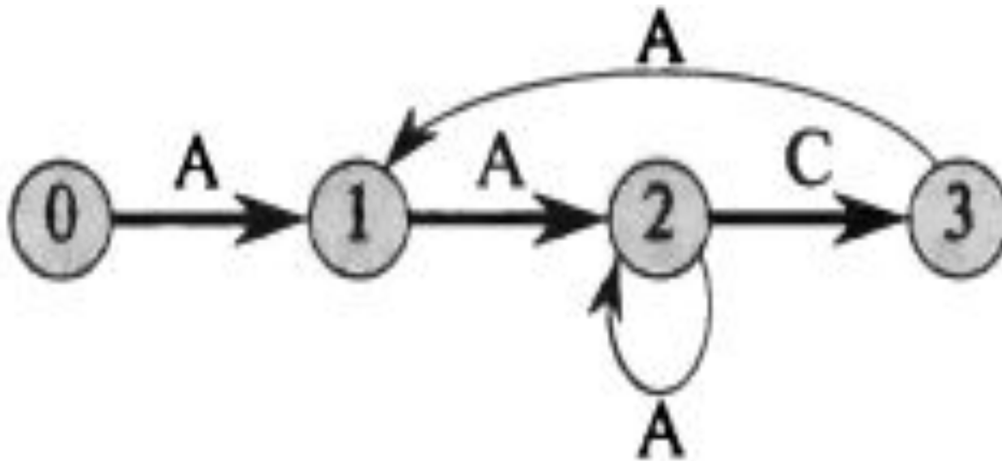
- **Входная последовательность**: символы текста T.
- **Число состояний КА**: m+1 состояние (на 1 больше, чем количество символов в шаблоне P), пронумерованное от 0 до m.
- КА **начинает** работу из состояния 0.
- Когда он находится в **состоянии k**, k последних считанных символов текста соответствуют первым k символам шаблона.
- Всякий раз, когда КА входит в **состояние m**, он встретил в тексте весь шаблон.
- КА хранит **таблицу next-state**, которая индексируется всеми состояниями и всеми возможными входными символами. Значение next-state[s,a] представляет собой номер состояния, в которое перейдет КА, если в настоящее время он находится в состоянии s и получил из текста символ a.

Пример



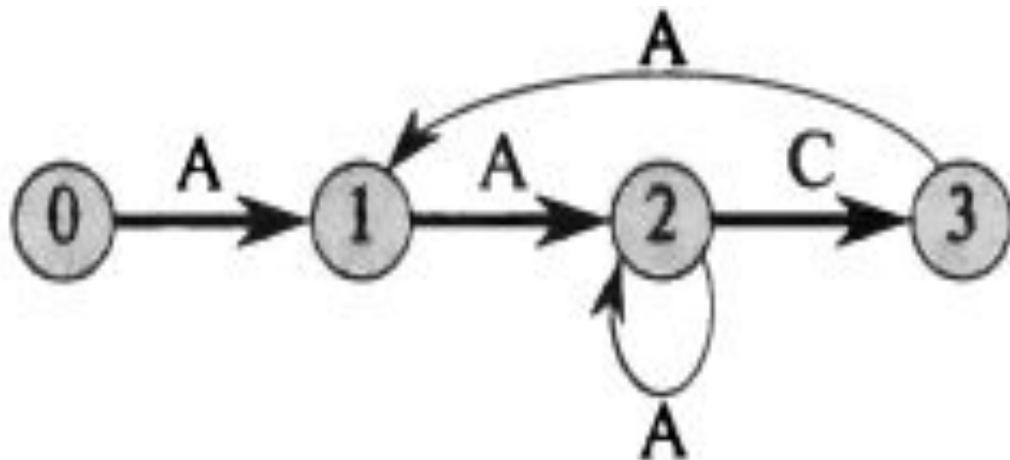
Входной текст: GTAACAGTAAACG.
Шаблон: AAC.

- Круги представляют состояния.
- Помеченные символами стрелки показывают, как КА переходит из одного состояния в другое при получении входных символов.
- Выделенный толстыми стрелками "позвоночник" при прочтении слева направо дает шаблон AAC.



Входной текст: GTAACAGTAAACG.
Шаблон: AAC.

- КА перемещается на одно состояние вправо для каждого символа, который соответствует шаблону, а для каждого символа, который ему не соответствует, он переходит влево или остается в том же состоянии.
- Всякий раз, когда в тексте встречается шаблон, КА перемещается вправо по одному состоянию для каждого символа, пока не достигнет последнего состояния, где он объявляет, что найдено вхождение шаблона в текст.
- Если стрелка отсутствует (например, стрелки, помеченные T), соответствующий переход ведет в состояние 0.



Входной текст: GTAACAGTAAACG.
Шаблон: AAC.

Таблица next-state:

Состояние	Символ A	C	G	T
0	1	0	0	0
1	2	0	0	0
2	2	3	0	0
3	1	0	0	0

Перемещение КА по состояниям при считывании символов из входного текста:

Состояние	0	0	0	1	2	3	1	0	0	1	2	2	3	0
Символ	G	T	A	A	C	A	G	T	A	A	A	C	G	

Процедура поиска подстрок

Процедура String-Matcher(T, next-state, m, n).

Вход:

- T, n – строка текста и ее длина,
- next-state – таблица переходов между состояниями, построенная для заданного шаблона,
- m – длина шаблона. Строки таблицы next-state индексированы от 0 до m, а столбцы индексированы символами, которые могут встретиться в тексте.

Выход: выводит все величины сдвигов, при которых в тексте встречается искомый шаблон.

Шаги процедуры:

1. Установить переменную state равной нулю.
2. Для i = 1 до n:
   A. Установить значение state равным next-state[state, t_i].
   B. Если state = m, вывести сообщение "Шаблон найден со сдвигом " i-m.

Несколько определений

- **Префикс** P_i шаблона P представляет собой подстроку, состоящую из первых i символов P.
- Определим **суффикс** шаблона как подстроку символов с конца P. Например, AGA — суффикс шаблона ACACAGA.

- Определим **конкатенацию** строки X и символа a как новую строку, получающуюся путем добавления a к концу X, и будем обозначать ее как Xa.

Определение нового состояния

- Если в состоянии k мы считали из текста префикс P_k, т.е. последние k считанных символов текста совпадают с первыми k символами шаблона.
- Когда мы считываем следующий символ, скажем, a, мы считываем из текста строку $P_k a$ (конкатенация P_k с a).
- Префикс P, считанный нами в этот момент, находится в конце $P_k a$, т.е. префикс P должен являться суффиксом $P_k a$. Длина этого префикса и является номером следующего состояния.
- Когда наидлиннейший префикс P, который одновременно является суффиксом $P_k a$, оказывается пустой строкой, мы устанавливаем next-state[k,a] = 0.

Алгоритм заполнения таблицы next-state

1. Образуем строку P_k a .
2. Устанавливаем i равным меньшему из значений k+1 (длина P_k a) и m (длина P).
3. Пока P_i не является суффиксом P_ka, выполняем следующее действие:
   A. Устанавливаем i равным i-1.

Найденное значение i и будет номером состояния КА, записываемым в ячейку next-state[k,a].

Задание

Реализовать алгоритм поиска подстрок с помощью конечного автомата:

- Создать строку и шаблон, записанные известным ограниченным числом символов,
- Создать таблицу next-state для данного шаблона,
- Реализовать алгоритм поиска подстрок с помощью найденной таблицы next-state.