DOI: 10.32517/2221-1993-2025-24-4-57-64

К. Ю. Поляков
Санкт-Петербургский государственный морской технический университет, Санкт-Петербург, Россия;
Школа № 174, Санкт-Петербург, Россия

МЕТОДЫ ОБРАБОТКИ СИМВОЛЬНЫХ СТРОК В ЗАДАЧАХ ЕГЭ ПО ИНФОРМАТИКЕ

Аннотация

В настоящей обзорной статье рассмотрены существующие подходы к решению задач на обработку символьных строк, включенных в контрольно-измерительные материалы (КИМ) Единого государственного экзамена (ЕГЭ) по информатике. Среди них можно выделить метод перебора, конечные автоматы, однопроходные алгоритмы, метод замен, динамическое программирование, регулярные выражения. Некоторые из этих инструментов недостаточно освещены в школьных учебниках информатики, хотя владение ими может существенно улучшить эффективность решения задач в условиях ЕГЭ. Для каждого из рассмотренных методов приведены характерные задачи, для решения которых этот метод эффективен. Все решения приводятся на языке программирования Python.

Статья может быть полезна для учителей информатики, а также для школьников с высоким уровнем подготовки в области программирования.

Ключевые слова: информатика, ЕГЭ, КИМ, символьные строки, перебор вариантов, однопроходные алгоритмы, конечные автоматы, метод замен, динамическое программирование, регулярные выражения.

Информация об авторе

Поляков Константин Юрьевич, доктор тех. наук, профессор кафедры судовой автоматики и измерений, факультет корабельной энергетики и автоматики, Санкт-Петербургский государственный морской технический университет, Санкт-Петербург, Россия; учитель информатики, школа № 174, Санкт-Петербург, Россия; *e-mail:* kpolyakov@mail.ru

1. Введение

В состав КИМ ЕГЭ по информатике неизменно входит задание на обработку символьных данных, которое в настоящее время имеет номер 24 [3]. Согласно спецификации, это задание высокого уровня сложности, для решения которого нужно написать программу на одном из языков программирования.

Сложность этого задания состоит в том, что возможные формулировки его условия чрезвычайно разнообразны. В этой ситуации важно не то, умеет ли ученик решать какую-то конкретную задачу, а то, какими инструментами (методами решения) он владеет и насколько удачно выбирает инструмент, оптимальный в каждом конкретном случае.

К сожалению, информации по обработке символьных строк, которая приводится в учебниках информатики [15], недостаточно для уверенного решения этого класса задач ЕГЭ. Настоящая статья пытается восполнить этот пробел, представляя читателю обзор основных методов решения задач ЕГЭ на символьные строки. Далее будут кратко рассмотрены метод перебора,

K. Yu. Polyakov
State Marine Technical University, Saint Petersburg, Russia;
School 174, Saint Petersburg, Russia

PROCESSING STRING DATA IN THE UNIFIED STATE EXAM IN INFORMATICS

Abstract

This review article considers various approaches to solving problems for processing character strings included in the materials of the Unified State Examination (USE) in informatics. Among them are the brute force method, finite automata, one-pass algorithms, substitution method, dynamic programming, regular expressions. Some of these tools are insufficiently covered in school textbooks, although mastering them can significantly improve the efficiency of solving problems on processing character strings in the conditions of USE. For each of the considered methods we present characteristic problems that can be effectively solved by this method. All solutions are given in Python.

The article can be useful for informatics teachers, as well as for pupils with an advanced level of programming skills.

Keywords: informatics, USE, strings, brute force algorithms, one-pass algorithms, finite automata, substitution method, dynamic programming, regular expressions.

Information about the author

Konstantin Yu. Polyakov, Doctor of Sciences (Engineering), Professor at the Department of Ship Automation and Measurements, Faculty of Ship Power Engineering and Automation, State Marine Technical University, Saint Petersburg, Russia; informatics teacher, School 174, Saint Petersburg, Russia; *e-mail:* kpolyakov@mail.ru

ISSN 2221-1993 • **Informatics in School** • 2025 • Volume 24 No 4

конечные автоматы, однопроходные алгоритмы, метод замен, динамическое программирование и регулярные выражения. Некоторые из этих методов подробно изложены в пособии [14], где приведено большое количество примеров решения разнообразных задач ЕГЭ.

2. Формулировка задачи

Рассмотрим типовую формулировку задания 24 из КИМ ЕГЭ по информатике [3].

Задание 24.

Текстовый файл 24.txt состоит из десятичных цифр и заглавных букв латинского алфавита. Определите в прилагаемом файле максимальное (вариант: минимальное) количество идущих подряд символов, для которых выполняется <условие>.

Подстроки, для которых выполняется <условие>, далее будем называть *правильными*.

Как правило, файл не содержит символов перехода на новую строку (в программах его обозначают как «\n»), поэтому все его содержимое можно воспринимать как одну символьную строку. Далее во всех фрагментах программ будем использовать язык программирования Python.

Чтобы прочитать строку из файла в оперативную память, в программе нужно открыть файл на чтение, выполнить непосредственно чтение и после этого закрыть файл:

```
F = open("24.txt")
s = F.readline() # F.read()
F.close()
```

Поскольку файл фактически содержит только одну строку, чтение можно выполнить либо методом *readline*, либо методом *read* (на выбор). В языке Python удобно использовать менеджер контекста *with*, который автоматически закроет файл после выхода программы из его области действия:

```
with open("24.txt") as F:
  s = F.readline()
```

На ЕГЭ допустим и такой вариант:

```
s = open("24.txt").readline()
```

При этом файл автоматически не закрывается, но в наиболее распространенной реализации CPython он будет закрыт сборщиком мусора.

Далее везде будем предполагать, что переменная *s* содержит строку, прочитанную из файла. Иногда в конце файла записан одиночный символ «\n», который может повлиять на результат обработки строки. Чтобы удалить его, достаточно сразу после чтения вызвать метод *rstrip*:

```
s = F.readline().rstrip()
```

Предположим, что условие, которому должны соответствовать правильные подстроки, задано в виде функции:

```
def valid( s: str ) -> bool:
  ...
  return <условие>
```

Эта функция принимает один параметр (символьную строку) и возвращает логическое значение: *True*, если переданная подстрока соответствует заданному в задаче условию, и *False* в противном случае.

3. Полный перебор

Простейший метод решения сформулированной задачи — перебрать все возможные подстроки исходной строки *s*, проверяя каждую из них на соответствие заданному условию с помощью функции *valid*. В переменных *L* и *R* будем хранить соответственно индекс левой границы рабочей подстроки *sub* и индекс первого символа справа от этой подстроки (рис. 1).

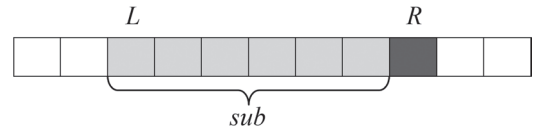

Рис. 1

Полный перебор всех возможных подстрок строки *s* выполняется с помощью вложенного цикла:

```
N = len(s)
maxLen = 0
for L in range(N):
  for R in range(L+1, N+1):
    sub = s[L:R]
    if valid(sub):
      curLen = R - L
      if curLen > maxLen:
        maxLen = curLen
        sMax = sub
print( maxLen, sMax )
```

Если подстрока *s*[*L*:*R*] удовлетворяет условию задачи и функция *valid* вернула результат *True*, определяем длину подстроки (переменная *curLen*) и, если она больше, чем длина *maxLen* предыдущей лучшей подстроки, обновляем значение *maxLen* и запоминаем новую лучшую строку в переменной *sMax*.

Реализованный алгоритм полного перебора имеет квадратичную сложность, т. е. количество операций растет пропорционально квадрату длины строки. Приведенная программа хорошо работает на коротких строках (длиной примерно до 1000 символов), но для заданий ЕГЭ, где строки содержат несколько миллионов символов, дождаться результата, скорее всего, не удастся. Тем не менее можно оптимизировать алгоритм, используя следующие соображения:

1) если мы уже нашли подходящую цепочку длиной *maxLen*, далее нет смысла проверять более короткие цепочки, поэтому перебор по *R* во вложенном цикле можно начать со значения $L+maxLen+1$, а не с $L+1$;
2) часто бывает так, что удлинение цепочки уже не может сделать ее правильной (например, если мы встретили недопустимый символ); в этом случае вложенный цикл (по переменной *R*) нужно завершить и сразу перейти к следующему значению *L*.

ISSN 2221-1993 • Информатика в школе • 2025 • Том 24 № 4

Таким образом, в программу требуется внести изменения, выделенные фоном:

```
for L in range(N):
  for R in range(L+ maxLen +1, N+1):
    sub = s[L:R]
    if valid(sub):
      ...
    elif cantBeValid(sub):
      break
```

Функция *cantBeValid* — это логическая функция, которая возвращает *True*, если цепочка не может стать правильной в результате удлинения, и *False* в противном случае. Ее содержание зависит от конкретной решаемой задачи.

Пусть, например, требуется найти наибольшую длину подстроки, содержащей ровно 100 букв A. В этом случае функции *valid* и *cantBeValid* запишутся следующим образом:

```
def valid( s: str ) -> bool:
  return s.count('A') == 100
def cantBeValid( s: str ) -> bool:
  return s.count('A') > 100
```

Для строки длиной 1 млн символов программа с оптимизацией на компьютере автора выполняет полный перебор за несколько секунд, что вполне приемлемо.

Программа легко модифицируется, если нужно найти цепочку не максимальной, а минимальной длины. Например, может потребоваться найти *наименьшую* длину подстроки, содержащей ровно 100 букв A. Вместо переменной *maxLen* будем использовать переменную *minLen*, в которой хранится минимальная длина уже найденной подходящей цепочки.

Сначала задается некоторое начальное значение *minLen*, такое, что предполагаемый ответ меньше этого значения. Перебор значений *R* лучше организовать в порядке убывания:

```
minLen = 250
for L in range(N):
  print( L ) # для информации
  for R in range( L+minLen-1, L, -1 ):
    ...
```

Чтобы убедиться в том, что программа действительно работает, и примерно оценить время ее выполнения, полезно между заголовками двух циклов добавить команду для вывода на экран текущего значения *L*.

Может случиться так, что в результате выполнения программы ни для одной из рассмотренных подстрок условие не выполнилось, тогда начальное значение *minLen* придется увеличить и запустить программу еще раз. Нежелательно задавать большое начальное значение *minLen*, например, равное длине строки *N* или бесконечному значению *float*("*inf*"), потому что при этом время вычислений может стать недопустимо большим (получится алгоритм квадратичной сложности).

Другие примеры решения задач ЕГЭ с помощью оптимизированного перебора можно найти в [9].

4. Однопроходные алгоритмы и конечные автоматы

Для многих задач удается найти эффективный однопроходный алгоритм, который один раз перебирает все символы строки и таким образом находит ответ за время, линейно зависящее от длины строки.

Пусть *count* — это количество символов в правильной подстроке, которая заканчивается последним уже рассмотренным символом. Теперь возьмем следующий символ, обозначенный на рисунке 2 как *c*, и проверим, останется ли правильной цепочка с добавленным символом *c*.

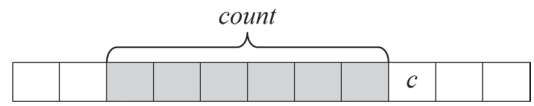

Рис. 2

Если добавленный символ *c* не нарушил правильность цепочки, увеличиваем счетчик символов подстроки *count* и, если получилась новая самая длинная подходящая цепочка, обновляем значение переменной *maxLen*. Если добавление нового символа *c* сделало цепочку неправильной, нужно присвоить переменной *count* значение 0 или 1 в зависимости от того, может ли символ *c* начать новую правильную цепочку.

Приведем фрагмент программы, которая реализует однопроходный алгоритм:

```
maxLen = 0
count = 0
for c in s:
  if <цепочка продолжается>:
    count += 1
    maxLen = max( count, maxLen )
  else:
    count = 0 # или 1
print( maxLen )
```

Иногда для определения правильности новой цепочки недостаточно рассмотреть только один символ *c*, а требуется дополнительная информация о предшествующих символах. В этом случае удобно применить конечный автомат (КА) [13], который переходит из одного состояния в другое в зависимости от полученного символа.

Для примера рассмотрим такую задачу: определить максимальное количество идущих подряд пар символов AB. КА для решения этой задачи имеет два состояния:

0 — ожидание символа A;

1 — ожидание символа B.

В начальный момент КА находится в состоянии 0. Схема, описывающая переходы и соответствующие изменения счетчика *count*, показана на рисунке 3.

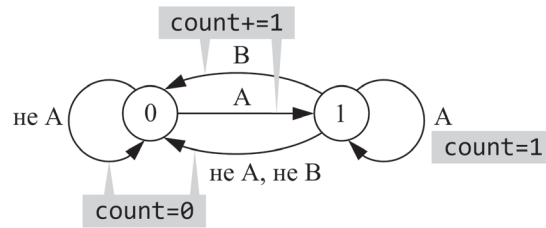

Рис. 3

Конечный автомат очень просто программируется с помощью вложенных условных операторов, хотя программа получается достаточно длинная:

```
state = 0
count = maxLen = 0
for c in s:
  if state == 0:
    if c == 'A':
      count += 1
      state = 1
    else:
      count = 0
  else: # state = 1
    if c == 'B':
      count += 1
      state = 0
    else:
      if c == 'A':
        count = 1
      else:
        count = 0
        state = 0
  maxLen = max( count, maxLen )
print( maxLen // 2 )
```

Эту программу можно упростить, учитывая, что переменная *state* в любой момент совпадает с остатком от деления значения переменной *count* на 2. Поэтому условие в первом операторе *if* заменяется так:

```
    if count % 2 == 0:
      ...
```

Теперь можно удалить из программы все строки, где используется переменная *state*, и таким образом получается обычный однопроходный алгоритм.

5. Замены и метод *split*

Замены символов (или сочетаний символов) часто позволяют свести исходную задачу к другой, уже известной задаче, для которой есть готовое решение [1].

Пусть нужно найти цепочку максимальной длины, составленную только из пар AB. Заменяем с помощью метода *replace* все такие пары на символ, отсутствующий в строке, например, на звездочку:

```
s = s.replace("AB", '*')
```

Теперь остается найти наибольшую длину цепочки из звездочек.

Предположим, что файл состоит только из латинских букв A, B, C, D, E, F и требуется найти цепочку максимальной длины, составленную только из пар «согласная+гласная». В этом случае можно заменить все согласные на символ «s», все гласные — на символ «g», а затем все пары «sg» — на звездочку. Таким образом, снова получаем уже известную задачу — найти наибольшую длину цепочки из звездочек:

```
for c in "BCDF": s = s.replace(c, 's')
for c in "AE": s = s.replace(c, 'g')
s = s.replace("sg", '*')
```

С помощью замен можно добиться того, чтобы в исходной строке остались только правильные подстроки, разделенные пробелами [7, 11]. Пусть нужно определить наибольшую длину цепочки, в которой нет двух соседних символов P [6]. Используя метод *replace*, вставим пробел между каждой парой символов P:

```
s = s.replace("PP", "P P")
s = s.replace("PP", "P P")
```

Метод *replace* нужно вызвать дважды с теми же параметрами из-за того, что после первого разбиения групп, состоящих более чем из двух символов P, останутся неразбитые пары. Например, строка PPPPP превратится в P PP PP. Второй вызов метода *replace* разобьет пробелами все оставшиеся пары.

Теперь можно разделить строку на правильные подстроки методом *split* и затем с помощью функции *max* найти подстроку максимальной длины:

```
sMax = max( s.split(), key=len )
print( len(sMax), sMax )
```

Именованный аргумент *key* при вызове функции *max* задает функцию, которая используется при сравнении строк. Здесь в качестве такой функции указана функция *len*, поэтому в результате получится строка максимальной длины.

Если, например, требуется найти наибольшее число, записанное в шестнадцатеричной системе счисления, нужно создать новую функцию с одним аргументом, которая находит значение числа по его шестнадцатеричной записи:

```
def int16( s: str ) -> int:
  return int( s, 16 )
```

Имя этой функции указывается при вызове функции *max* в качестве аргумента *key*:

```
sMax = max( s.split(), key=int16 )
print( int16(sMax), sMax )
```

Такой подход позволяет довольно просто решать весьма сложные задачи. Рассмотрим, например, задачу 24 из [4]. Текстовый файл состоит из цифр 0, 6, 7, 8, 9 и знаков арифметических операций «–» и «*» (вычитание и умножение). Нужно определить максимальное количество символов в непрерывной последовательности, которая является корректным арифметическим выражением с целыми неотрицательными числами. В этом выражении никакие два знака арифметических операций не стоят рядом, в записи чисел отсутствуют незначащие (ведущие) нули и число 0 не имеет знака.

Для того чтобы не обрабатывать особым образом граничные случаи — ведущие нули в самом начале строки и знак операции в конце, добавим к исходной строке пробелы с обеих сторон:

```
s = ' ' + s + ' '
```

Далее выполним серию замен, удаляя (или разбивая пробелами) все запрещенные комбинации символов:

```
for c in "6789":                # 1
  s = s.replace(c, '1')         # 2
s = s.replace('-', '*')         # 3
s = s.replace("**", ' ')        # 4
```

ISSN 2221-1993 • **Информатика в школе** • 2025 • Том 24 № 4

```
s = s.replace(" *", ' ')        # 5
s = s.replace("* ", ' ')        # 6
s = s.replace("*00", "*0 0")    # 7
s = s.replace("*01", "*0 1")    # 8
while " 00" in s:               # 9
  s = s.replace(" 00", " 0")    # 10
s = s.replace(" 01", " 1")      # 11
```

Для удобства объяснения строки программы пронумерованы. Сначала (в строках 1, 2) все цифры (6, 7, 8 и 9) заменяются на цифру 1, поскольку нам важно только то, что в этих позициях стоят ненулевые цифры (вычислять значения выражений не требуется). Затем (строка 3) знаки «минус» заменяются на умножение. Далее (строки 4–6) заменяются на пробелы все парные знаки, а также знаки, стоящие перед пробелом и после пробела. В строках 7, 8 нули, стоящие после знака операции, отделяются от следующих за ними цифр. Цикл в строках 9, 10 удаляет ведущие нули (их может быть много, поэтому цикл необходим). После этого у чисел может остаться один незначащий ноль (после пробела), который удаляется заменой в строке 11. После выполнения всех этих замен строка *s* будет содержать только правильные подстроки, разделенные пробелами.

6. Метод двух указателей

Метод двух указателей — стандартный инструмент при обработке числовых массивов данных. В [2] и [16] было предложено использовать его при обработке символьных строк в задачах ЕГЭ.

Пусть начальный и конечный символы рабочей подстроки имеют индексы L и R. В основном цикле будем перебирать все возможные значения R от 0 до $N-1$. Для каждого из них определяем наилучшее (минимальное или максимальное) значение L, при котором рабочая подстрока $s[L:R+1]$ удовлетворяет условию задачи.

Когда R на очередной итерации цикла увеличивается на 1, условие правильности подстроки может нарушиться. Тогда нужно увеличить L, сдвинув левую границу подстроки вправо так, чтобы строка снова стала правильной.

Рассмотрим следующую задачу [5]: определить максимальную длину подстроки, содержащей ровно 100 букв Т.

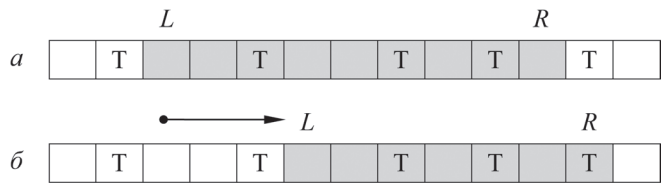

Рис. 4

На рисунке 4 показано, как работает метод двух указателей при поиске самой длинной подстроки, содержащей три символа Т. На рисунке 4, *а* фоном выделена правильная подстрока, которая заканчивается символом с индексом R. После очередного увеличения R в рабочую подстроку входит четвертая буква Т, так что подстрока становится неправильной. Поэтому нужно увеличить индекс L так, чтобы одна буква Т вышла из рабочей подстроки (рис. 4, *б*).

Будем хранить в переменной *countT* количество букв Т в рабочей подстроке. Это значение будем передавать в функцию *valid*, которая выглядит так:

```
def valid( countT: int ) -> bool:
  return countT == 100
```

Кроме того, введем логическую функцию *tooLong*, которая будет возвращать значение *True*, если строка слишком длинная (т. е. содержит больше 100 букв Т), и *False* в противном случае:

```
def tooLong( countT: int ) -> bool:
  return countT > 100
```

Приведем программу для решения рассматриваемой задачи методом двух указателей:

```
N = len(s)
maxLen = 0
L = countT = 0
for R in range(N):
  countT += (s[R] == 'T')            # 1
  while tooLong( countT ):           # 2
    countT -= (s[L] == 'T')          # 3
    L += 1                           # 4
  if valid( countT ):                # 5
    maxLen = max( R-L+1, maxLen )    # 6
print( maxLen )
```

В строке 1 для нового значения R проверяется символ $s[R]$: если это буква Т, счетчик *countT* увеличивается на 1, а если нет, то значение *countT* не изменяется. Значение выражения (s[R] == 'T') — логическое, значение *False* при суммировании автоматически преобразуется в 0, а значение *True* — в 1. Цикл в строках 2–4 «подтягивает» левую границу L рабочей подстроки так, чтобы эта подстрока стала правильной. На каждой итерации цикла символ $s[L]$ выходит из рабочей подстроки. Если это буква Т, значение счетчика *countT* уменьшается на 1 (строка 3). Перед обновлением значения *maxLen* в строке 6 проверяем, что рабочая подстрока действительно правильная (строка 5).

Пусть теперь требуется найти подстроку *минимальной* длины, содержащую ровно 100 букв Т. Разница состоит только в том, что при «подтягивании» левой границы строки нужно останавливаться непосредственно на следующей букве Т (рис. 5). При этом длина подходящей подстроки, заканчивающейся символом $s[R]$, будет минимальной.

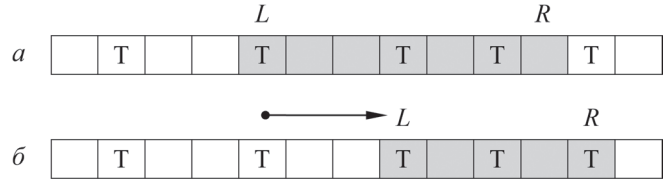

Рис. 5

Условие в функции *tooLong* усложнится:

```
def tooLong( c: str, countT: int ) -> bool:
  return countT > 100 or \
         (countT == 100 and c != 'T')
```

ISSN 2221-1993 • **Informatics in School** • 2025 • Volume 24 No 4

Вторая часть условия (она выделена фоном) означает, что нужно сокращать строку до тех пор, пока символ на левой границе строки не совпадет с буквой T.

Функции теперь нужно передавать два аргумента — не только значение счетчика *countT*, но и текущий символ *s*[*L*]:

```
while tooLong( s[L], countT ) :
  ...
```

В [10] и [16] можно познакомиться с другими примерами использования метода двух указателей при решении задач ЕГЭ по информатике.

7. Динамическое программирование

Еще один полезный прием связан с использованием динамического программирования. Иногда такой метод не только оказывается самым эффективным (алгоритм имеет линейную сложность), но и проще всего реализуется.

Рассмотрим следующую задачу (автор — Е. Джобс): найти максимальную длину подстроки, составленной только из фрагментов XY, YZ и YZZ, соединенных в произвольном порядке.

Сложность этой задачи состоит в том, что заданные фрагменты имеют пересечения: 1) фрагмент XY заканчивается на букву Y, с которой начинаются остальные два фрагмента; 2) фрагменты YZ и YZZ начинаются с одной и той же последовательности символов. Применить метод замен в этом случае крайне сложно. Пусть, например, нам дана строка XYZYZZ. Заменив XY на звездочку, мы потеряем лучшее решение — строку YZYZZ.

Решим задачу, используя динамическое программирование. Выделим дополнительный массив* *dp*, имеющий то же количество элементов, что и исходная строка *s*. Сначала все элементы массива *dp* имеют нулевые значения:

```
N = len(s)
dp = [0]*N
```

Далее заполняем массив *dp*, начиная с меньших индексов. Значение *dp*[*i*] после выполнения программы должно содержать максимальную длину правильной цепочки, которая заканчивается на символе *s*[*i*] исходной строки:

```
for i in range(N):
  if s[i-1:i+1] in ["XY", "YZ"]:      # 1
    dp[i] = 2 + dp[i-2]               # 2
  if s[i-2:i+1] in ["YZZ"]:           # 3
    dp[i] = max( dp[i], 3+dp[i-3] )   # 4
```

Для каждого индекса *i* проверяем цепочку из двух символов, которая заканчивается на символе *s*[*i*] — это срез *s*[*i*–1:*i*+1] (строки 1, 2). Если этот срез совпадает с одним из заданных фрагментов, добавляем 2 (длину найденного фрагмента) к значению *dp*[*i*–2]. Так мы учли вариант, когда последний фрагмент в правильной цепочке состоит из двух символов. Но может случиться так, что правильная цепочка заканчивается фрагментом из трех символов — среди заданных фрагментов есть один такой. Аналогичная проверка выполнена в строках 3, 4. Функция *max* используется для того, чтобы выбрать лучший из всех возможных вариантов.

Рассмотрим граничные случаи, когда $i < 2$ и при обращениях к массиву *dp* и строке *s* появляются отрицательные индексы. В языке Python *dp*[–1] и *dp*[–2] обозначают соответственно последний и предпоследний элементы списка, которые сначала равны нулю. Кроме того, срезы *s*[–1:1], *s*[–2:1] и *s*[–1:2] — это пустые строки, так что условия в обоих условных операторах не могут оказаться истинными. Поэтому такие граничные случаи при программировании на Python можно не обрабатывать особым образом.

Продемонстрируем работу программы на примере строки XYYYZXYYZZX (рис. 6).

	0	1	2	3	4	5	6	7	8	9	10
s	X	Y	Y	Y	Z	X	Y	Y	Z	Z	X
dp	0	2	0	0	2	0	4	0	6	7	0

Рис. 6

При заполнении массива *dp* будут выполнены следующие присваивания:

```
dp[1] = 2 + dp[-1]
dp[4] = 2 + dp[2]
dp[6] = 2 + dp[4]
dp[8] = 2 + dp[6]
dp[9] = 3 + dp[6]
```

Ответ к задаче — это наибольшее значение в массиве *dp*, совпадающее с максимальной длиной подходящей цепочки символов:

```
print( max(dp) )
```

8. Регулярные выражения

Регулярные выражения — это инструмент для поиска подстрок, соответствующих шаблонам, записанным на специальном языке. Для работы с регулярными выражениями в стандартной библиотеке Python существует модуль *re*, из которого чаще всего нужна только одна функция *findall*. Ее нужно импортировать в начале программы:

```
from re import findall
```

Эта функция возвращает список подстрок исходной строки, соответствующих заданному шаблону. Из этих подстрок затем нужно выбрать лучшую подстроку в каком-то смысле, например, подстроку максимальной длины:

```
pattern = <нужный шаблон>
parts = findall( pattern, s )
sMax = max( parts, key=len )
print( len(sMax), sMax )
```

Приведем несколько характерных шаблонов, которые можно применить при составлении регулярных выражений в задачах ЕГЭ.

В шаблонах используются знаки «+» и «*», обозначающие количество повторений (их называют *квантифи-*

* В языке Python это будет список, но мы будем использовать его как обычный массив в классических языках программирования.

ISSN 2221-1993 • **Информатика в школе** • 2025 • Том 24 № 4

каторами): «+» обозначает одно или более повторений, а «*» — ноль или более повторений. Например, шаблон для строки, состоящей только из букв A (хотя бы из одной), запишется как «A+».

Если необходимо повторить не один символ, а несколько символов в определенной последовательности, нужно составить группу, взяв ее в круглые скобки. Например, при поиске цепочек, состоящих из пар AB, используется шаблон «(?:AB)+». Кроме скобок здесь добавлены знак вопроса и двоеточие — они обозначают *незахватывающую группу*. Без них функция *findall* для каждой найденной подстроки вернет только одну пару AB, и мы не сможем определить, какая подстрока длиннее и какова ее длина.

Теперь будем искать цепочку, состоящую из пар «согласная+гласная» при условии, что в исходной строке встречаются только буквы A, B, C, D, E и F. В этом случае шаблон выглядит следующим образом: «(?:[BCDF][AE])+». Квадратные скобки обозначают, что на этом месте может быть любой из символов, перечисленных внутри скобок.

Для символов, которые расположены подряд в кодовой таблице, можно использовать диапазоны, разделяя знаком «-» начальный и конечный символ. Например, шаблон «[A-Z][a-z]*» определяет слова, начинающиеся с заглавной буквы, а шаблон «[1-9AB][0-9AB]*» – натуральное число, записанное в двенадцатеричной системе счисления без ведущих (незначащих) нулей.

Пусть нужно найти наибольшее четное число, записанное в четырнадцатеричной системе счисления. Поскольку основание системы четно, для четности числа достаточно, чтобы его последняя цифра имела четное значение, т. е. это должна быть цифра из набора 0, 2, 4, 6, 8, A (10) и C (12). Составим шаблон и найдем все подходящие подстроки:

```
pattern = "[0-9A-D]*[02468AC]"
parts = findall( pattern, s )
```

Теперь нужно выбрать из списка *parts* подстроку с наибольшим числовым значением. Для этого добавим в программу функцию, вычисляющую это значение с помощью стандартной функции *int*:

```
def int14( s ):
  return int( s, 14 )
```

Теперь можно использовать эту функцию в качестве аргумента *key* при вызове функции *max*:

```
sMax = max( parts, key=int14 )
print( int14(sMax), sMax )
```

Иногда нужно использовать операцию ИЛИ не для отдельных символов, а для целых фрагментов. В этом случае применяется знак «|». Например, шаблон «[1-9]\d*|0» задает натуральное число без ведущих нулей или число 0. Здесь комбинация «\d» обозначает любую цифру (от 0 до 9). При записи такого выражения в Python перед символьной строкой нужно поставить букву «r» (от *англ.* raw — сырой), которая говорит о том, что символ «\» в этой строке не имеет специального значения, как, например, в комбинации «\n»:

```
pattern = r"[1-9]\d*|0"
```

Пусть требуется найти правильное арифметическое выражение с неотрицательными числами и знаками «-» и «*» [4]. Такое выражение можно представить как число, за которым, возможно, следуют фрагменты вида «знак + число». Составим сначала шаблон для одного числа, которое записано в десятичной системе:

```
number = r"(?:[1-9]\d*|0)"
```

Теперь строим полный шаблон для всего выражения, подставляя фрагмент *number* дважды с помощью f-строки:

```
pattern = fr"{number}(?:[-*]{number})*"
```

Этот шаблон означает, что сначала записано число, а затем произвольное (возможно, нулевое) количество групп «знак + число».

Последний пример — шаблон для арифметического выражения, содержащего только неотрицательные числа и знаки «+» и «*», *значение которого равно нулю*. Так как скобки не используются, сначала выполняются все операции умножения, а затем — все операции сложения. Поэтому такое выражение представляет собой сумму произведений. Для того чтобы его значение было равно нулю, каждое произведение должно быть равно нулю, т. е. должно содержать число 0, которое слева и справа может быть умножено на любые другие числа. Шаблон в этом случае удобно строить в три шага [8]:

```
number = r"(?:[1-9]\d*|0)"
prod = fr"(?:{number}\*)*0(?:\*{number})*"
pattern = fr"{prod}(?:\+{prod})*"
```

Здесь *number* — это уже известный шаблон для одного числа, *prod* — шаблон для произведения, равного нулю (содержащего множитель 0), и *pattern* — полный шаблон для всего выражения. Обратите внимание на знаки «\» перед знаками арифметических операций «+» и «*»: они означают, что символы «+» и «*» здесь являются не квантификаторами, а обычными символами.

В некоторых задачах для получения правильного результата нужно повторять поиск, начиная с каждого символа строки. Вернемся к уже рассмотренной задаче: найти максимальную длину подстроки, составленной из фрагментов XY, YZ и YZZ, соединенных в произвольном порядке. Пусть исходная строка имеет вид XYZZXYXY, так что решением будет подстрока YZZXYXY длиной семь символов. При использовании регулярного выражения «(?:YZZ|XY|YZ)+» функция *findall* возвращает список из двух строк, среди которых нет правильного решения:

```
["XY", "XYXY"]
```

Дело в том, что после неудачи с продолжением шаблона на первой букве Z функция *findall* начинает искать следующее возможное совпадение с шаблоном именно с этой буквы Z, а не с предыдущей буквы Y. В результате правильное решение теряется.

Чтобы исправить ситуацию, нужно применить конструкцию «(?=)», которая требует повторять поиск шаблона, начиная с *каждого* символа строки:

```
expr = "(?:YZZ|XY|YZ)+"
pattern = f"(?=({expr}))"
```

Обратите внимание на скобки, выделенные фоном. Это захватывающая группа, определяющая, что нам требуются все найденные подстроки целиком. Если эти скобки опустить, функция *findall* вернет список пустых строк.

Необходимо учитывать, что использование конструкции «(?=)» значительно замедляет работу программы, и поэтому стоит применять ее только тогда, когда она действительно необходима.

Другие примеры использования регулярных выражений для решения задач ЕГЭ по информатике можно найти в [8] и [12]. Заметим, что этот метод подходит не для всех задач. Например, найти максимальную длину подстроки, в которой ровно 100 букв Т [5], с помощью регулярного выражения в принципе возможно, но шаблон получается очень сложным и программа работает несколько минут [8].

9. Выводы

В настоящей статье приведен обзор методов решения задач на обработку символьных строк, которые предлагаются в КИМ ЕГЭ по информатике. Каждый из них имеет свою сферу применимости, поэтому наиболее важный момент при подготовке к экзамену — научить школьников выбирать метод, который быстрее всего приведет к правильному ответу. Не исключено, что в будущем появятся новые формулировки заданий, и тогда владение разнообразными инструментами (а не выученные решения конкретных задач) позволит учащимся успешно справиться с неожиданной ситуацией во время экзамена.

Список источников

1. *Воропаев И.* (PRO100-ЕГЭ) Мега-простой шаблон для № 24. https://youtu.be/FpJOLBRT3Q8

2. *Воропаев И.* СтатГрад № 1 ЕГЭ по информатике 24.10.2023 | Разбор всего варианта. https://youtu.be/BzVI_V02CQU

3. Демонстрационный вариант контрольных измерительных материалов единого государственного экзамена 2026 года по информатике. https://doc.fipi.ru/ege/demoversii-specifikacii-kodifikatory/2026/inf_11_2026.zip

4. Демонстрационный вариант контрольных измерительных материалов единого государственного экзамена 2025 года по информатике. https://doc.fipi.ru/ege/demoversii-specifikacii-kodifikatory/2025/inf_11_2025.zip

5. Демонстрационный вариант контрольных измерительных материалов единого государственного экзамена 2024 года по информатике. https://doc.fipi.ru/ege/demoversii-specifikacii-kodifikatory/2024/inf_11_2024.zip

6. Демонстрационный вариант контрольных измерительных материалов единого государственного экзамена 2022 года по информатике. https://doc.fipi.ru/ege/demoversii-specifikacii-kodifikatory/2022/inf_11_2022.zip

7. *Джобс Е.* Задание № 24 через разбивку на слова | Информатика ЕГЭ-2023. https://vk.com/video-184870282_456257107

8. *Джобс Е.* Регулярные выражения для 24 задания. https://vk.com/video-184870282_456257255

9. *Кабанов А. М.* Задание 24 (двойной цикл) | КЕГЭ по информатике 2025. https://rutube.ru/video/986acace6e6505761b3ea022a959d776

10. *Кабанов А. М.* Задание 24 (метод двух указателей) | КЕГЭ по информатике 2024. https://vk.com/video-205865487_456239725

11. *Кабанов А. М.* Задание 24 (разбиение строки) | КЕГЭ по информатике 2024. https://vk.com/video-205865487_456239723

12. *Кабанов А. М.* Задание 24 (регулярные выражения) | КЕГЭ по информатике 2025. https://vk.com/video-205865487_456240469

13. *Малявко А. А.* Формальные языки и компиляторы: учебное пособие. Новосибирск: НГТУ, 2014. 431 с.

14. *Поляков К. Ю.* 100 баллов по информатике. Решаем задачи ЕГЭ на языке Python. 3-е изд. М: Лаборатория знаний, 2025. 400 с.

15. *Поляков К. Ю., Еремин Е. А.* Информатика. 10 класс (базовый и углубленный уровни): в 2 ч.: учебник. Ч. 1. М.: БИНОМ. Лаборатория знаний, 2023. 352 с.

16. *Шастин Л. Д.* Задание 24 — Метод двух указателей | Информатика ЕГЭ 2024. https:// youtu.be/mr7m-2F_UsM