**Тема занятия №9: Сортировка, поиск, регулярные выражения**

**Введение**

Поиск — процесс обнаружения в некотором наборе данных таких элементов (объектов), характеристики которых (одна или несколько) соответствуют критериям поиска. Критерий поиска — это условие (ограничение), накладываемое на значения свойств (характеристик) данных.

Приведем пример: у нас есть список логинов пользователя и нужно узнать, есть ли в этом списке логин «admin». Значение критерия поиска часто называют ключом поиска (в нашем примере ключ — это строка «admin»). В этом случае критерий поиска: значение элемента равно «admin» (проверяемой характеристикой элемента является его значение).

**9.1 Линейный поиск**

**Линейный поиск** — это один из самых простых и понятных алгоритмов поиска. Мы можем думать о нем как о расширенной версии нашей собственной реализации оператора in в Python.

Суть алгоритма заключается в том*,* чтобы перебрать массив и вернуть индекс первого вхождения элемента, когда он найден:

**def LinearSearch(lys, element):**

**for i in range (len(lys)):**

**if lys[i] == element:**

**return i**

**return -1**

Итак, если мы используем функцию для вычисления:

**>>> print(LinearSearch([1,2,3,4,5,2,1], 2))**

То получим следующий результат:

**1**

Это индекс первого вхождения искомого элемента, учитывая, что нумерация элементов в Python начинается с нуля.

Временная сложность линейного поиска равна O(n). Это означает, что время, необходимое для выполнения, увеличивается с увеличением количества элементов в нашем входном списке lvs.

Линейный поиск не часто используется на практике, потому что такая же эффективность может быть достигнута с помощью встроенных методов или существующих операторов. К тому же, он не такой быстрый и эффективный, как другие алгоритмы поиска.

Линейный поиск хорошо подходит для тех случаев, когда нам нужно найти первое вхождение элемента в несортированной коллекции. Это связано с тем, что он не требует сортировки коллекции перед поиском (в отличие от большинства других алгоритмов поиска).

**9.2 Бинарный поиск**

Бинарный поиск работает по принципу [«разделяй и властвуй»](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B9_%D0%B8_%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B2%D1%83%D0%B9_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

Предполагая, что мы ищем значение val в отсортированном массиве, алгоритм сравнивает val со значением среднего элемента массива, который мы будем называть mid.

Если mid — это тот элемент, который мы ищем (в лучшем случае), мы возвращаем его индекс.

Если нет, мы определяем, в какой половине массива мы будем искать val дальше, основываясь на том, меньше или больше значение val значения mid, и отбрасываем вторую половину массива.

Затем мы рекурсивно или итеративно выполняем те же шаги, выбирая новое значение для mid, сравнивая его с val и отбрасывая половину массива на каждой итерации алгоритма.

Алгоритм бинарного поиска можно написать, как рекурсивно, так и итеративно.

Поскольку хороший алгоритм поиска должен быть максимально быстрым и точным, давайте рассмотрим итеративную реализацию бинарного поиска:

**def BinarySearch(lys, val):**

**first = 0**

**last = len(lys)-1**

**index = -1**

**while (first <= last) and (index == -1):**

**mid = (first+last)//2**

**if lys[mid] == val:**

**index = mid**

**else:**

**if val<lys[mid]:**

**last = mid -1**

**else:**

**first = mid +1**

**return index**

Если мы используем функцию для вычисления:

**>>> BinarySearch([10,20,30,40,50], 20)**

То получим следующий результат, являющийся индексом искомого значения:

**1**

На каждой итерации алгоритм выполняет одно из следующих действий:

Возврат индекса текущего элемента.

Поиск в левой половине массива.

Поиск в правой половине массива.

Мы можем выбрать только одно действие на каждой итерации. Также на каждой итерации наш массив делится на две части. Из-за этого временная сложность двоичного поиска равна O(log n).

Одним из недостатков бинарного поиска является то, что если в массиве имеется несколько вхождений элемента, он возвращает индекс не первого элемента, а ближайшего к середине:

**>>> print(BinarySearch([4,4,4,4,4], 4))**

После выполнения этого фрагмента кода будет возвращен индекс среднего элемента:

**2**

Для сравнения: выполнение линейного поиска по тому же массиву вернет индекс первого элемента:

**0**

Однако мы не можем категорически утверждать, что двоичный поиск не работает, если массив содержит дубликаты. Он может работать так же, как линейный поиск, и в некоторых случаях возвращать первое вхождение элемента. Например:

**>>> print(BinarySearch([1,2,3,4,4,4,5], 4))**

**3**

Бинарный поиск довольно часто используется на практике, потому что он эффективен и быстр по сравнению с линейным поиском. Однако у него есть некоторые недостатки, такие как зависимость от оператора //.

**Регулярные выражения**

**Введение**

Регулярное выражение — это строка, задающая шаблон поиска подстрок в тексте. Одному шаблону может соответствовать много разных строчек. Термин «Регулярные выражения» является переводом английского словосочетания «Regular expressions». Перевод не очень точно отражает смысл, правильнее было бы «шаблонные выражения». Регулярное выражение, или коротко «регулярка», состоит из обычных символов и специальных командных последовательностей. Например, \d задаёт любую цифру, а \d+ — задает любую последовательность из одной или более цифр. Работа с регулярками реализована во всех современных языках программирования.

**Основы регулярных выражений**

Регулярками называются шаблоны, которые используются для поиска соответствующего фрагмента текста и сопоставления символов.

Грубо говоря, у нас есть input-поле, в которое должен вводиться email-адрес. Но пока мы не зададим проверку валидности введённого email-адреса, в этой строке может оказаться совершенно любой набор символов, а нам это не нужно.

Чтобы выявить ошибку при вводе некорректного адреса электронной почты, можно использовать следующее регулярное выражение:

**r'^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)+$'**

По сути, наш шаблон — это набор символов, который проверяет строку на соответствие заданному правилу.

**Синтаксис RegEx**

Синтаксис у регулярок необычный. Любая строка (в которой нет символов .^$\*+?{}[]\|()) сама по себе является регулярным выражением. Так, выражению Хаха будет соответствовать строка “Хаха” и только она. Регулярные выражения являются регистрозависимыми, поэтому строка “хаха” (с маленькой буквы) уже не будет соответствовать выражению выше. Подобно строкам в языке Python, регулярные выражения имеют спецсимволы .^$\*+?{}[]\|(), которые в регулярках являются управляющими конструкциями. Для написания их просто как символов требуется их экранировать, для чего нужно поставить перед ними знак \. Так же, как и в питоне, в регулярных выражениях выражение \n соответствует концу строки, а \t — табуляции.

Символы могут быть как буквами или цифрами, так и метасимволами, которые задают шаблон строки:

Также есть дополнительные конструкции, которые позволяют сокращать регулярные выражения:

\d — соответствует любой одной цифре и заменяет собой выражение [0-9];

\D — исключает все цифры и заменяет [^0-9];

\w — заменяет любую цифру, букву, а также знак нижнего подчёркивания;

\W — любой символ кроме латиницы, цифр или нижнего подчёркивания;

\s — соответствует любому пробельному символу;

\S — описывает любой непробельный символ.

Для чего используются регулярные выражения:

* для определения нужного формата, например телефонного номера или email-адреса;
* для разбивки строк на подстроки;
* для поиска, замены и извлечения символов;
* для быстрого выполнения нетривиальных операций.

Синтаксис таких выражений в основном стандартизирован, так что вам следует понять их лишь раз, чтобы использовать в любом языке программирования.

Но не стоит забывать, что регулярные выражения не всегда оптимальны, и для простых операций часто достаточно встроенных в Python функций.

Примеры регулярных выражений

Регулярка Её смысл

simple text В точности текст «simple text»

\d{5} Последовательности из 5 цифр

\d означает любую цифру

{5} — ровно 5 раз

\d\d/\d\d/\d{4} Даты в формате ДД/ММ/ГГГГ

(и прочие куски, на них похожие, например, 98/76/5432)

\b\w{3}\b Слова в точности из трёх букв

\b означает границу слова

(с одной стороны буква, а с другой — нет)

\w — любая буква,

{3} — ровно три раза

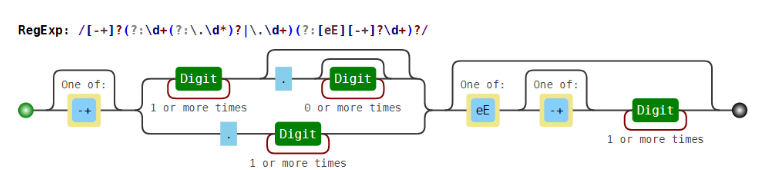
[-+]?\d+ Целое число, например, 7, +17, -42, 0013 (возможны ведущие нули)

[-+]? — либо -, либо +, либо пусто

\d+ — последовательность из 1 или более цифр

[-+]?(?:\d+(?:\.\d\*)?|\.\d+)(?:[eE][-+]?\d+)? Действительное число, возможно в экспоненциальной записи

Например, 0.2, +5.45, -.4, 6e23, -3.17E-14.



**Регулярные выражения в Python**

В Python для работы с регулярками есть модуль re. Его нужно просто импортировать:

import re

А вот наиболее популярные методы, которые предоставляет модуль:

re.match()

re.search()

re.findall()

re.split()

re.sub()

re.compile()

Рассмотрим подробнее по отдельности:

e.match(pattern, string)

Этот метод ищет по заданному шаблону в начале строки. Например, если мы вызовем метод match() на строке «AV Analytics AV» с шаблоном «AV», то он завершится успешно. Но если мы будем искать «Analytics», то результат будет отрицательный:

**import re**

**result = re.match(r'AV', 'AV Analytics Vidhya AV')**

**print result**

**Результат:**

**<\_sre.SRE\_Match object at 0x0000000009BE4370>**

Искомая подстрока найдена. Чтобы вывести её содержимое, применим метод group() (мы используем «r» перед строкой шаблона, чтобы показать, что это «сырая» строка в Python):

**result = re.match(r'AV', 'AV Analytics Vidhya AV')**

**print result.group(0)**

**Результат:**

**AV**

Теперь попробуем найти «Analytics» в данной строке. Поскольку строка начинается на «AV», метод вернет None:

**result = re.match(r'Analytics', 'AV Analytics Vidhya AV')**

**print result**

**Результат:**

**None**

Также есть методы start() и end() для того, чтобы узнать начальную и конечную позицию найденной строки.

**result = re.match(r'AV', 'AV Analytics Vidhya AV')**

**print result.start()**

**print result.end()**

**Результат:**

**0**

**2**

Эти методы иногда очень полезны для работы со строками.

re.search(pattern, string)

Метод похож на match(), но ищет не только в начале строки. В отличие от предыдущего, search() вернёт объект, если мы попытаемся найти «Analytics»:

**result = re.search(r'Analytics', 'AV Analytics Vidhya AV')**

**print result.group(0)**

**Результат:**

**Analytics**

Метод search() ищет по всей строке, но возвращает только первое найденное совпадение.

**re.findall(pattern, string)**

Возвращает список всех найденных совпадений. У метода findall() нет ограничений на поиск в начале или конце строки. Если мы будем искать «AV» в нашей строке, он вернет все вхождения «AV». Для поиска рекомендуется использовать именно findall(), так как он может работать и как re.search(), и как re.match().

**result = re.findall(r'AV', 'AV Analytics Vidhya AV')**

**print result**

Результат:

**['AV', 'AV']**

**re.split(pattern, string, [maxsplit=0])**

Этот метод разделяет строку по заданному шаблону.

**result = re.split(r'y', 'Analytics')**

**print result**

**Результат:**

**['Anal', 'tics']**

В примере мы разделили слово «Analytics» по букве «y». Метод split() принимает также аргумент maxsplit со значением по умолчанию, равным 0. В данном случае он разделит строку столько раз, сколько возможно, но если указать этот аргумент, то разделение будет произведено не более указанного количества раз. Давайте посмотрим на примеры Python RegEx:

**result = re.split(r'i', 'Analytics Vidhya')**

**print result**

**Результат:**

**['Analyt', 'cs V', 'dhya'] # все возможные участки.**

**result = re.split(r'i', 'Analytics Vidhya',maxsplit=1)**

**print result**

**Результат:**

**['Analyt', 'cs Vidhya']**

Мы установили параметр maxsplit равным 1, и в результате строка была разделена на две части вместо трех.

**re.sub(pattern, repl, string)**

Ищет шаблон в строке и заменяет его на указанную подстроку. Если шаблон не найден, строка остается неизменной.

**result = re.sub(r'India', 'the World', 'AV is largest Analytics community of India')**

**print result**

**Результат:**

**'AV is largest Analytics community of the World'**

**re.compile(pattern, repl, string)**

Мы можем собрать регулярное выражение в отдельный объект, который может быть использован для поиска. Это также избавляет от переписывания одного и того же выражения.

**pattern = re.compile('AV')**

**result = pattern.findall('AV Analytics Vidhya AV')**

**print result**

**result2 = pattern.findall('AV is largest analytics community of India')**

**print result2**

**Результат:**

**['AV', 'AV']**

**['AV']**

До сих пор мы рассматривали поиск определенной последовательности символов. Но что, если у нас нет определенного шаблона, и нам надо вернуть набор символов из строки, отвечающий определенным правилам? Такая задача часто стоит при извлечении информации из строк. Это можно сделать, написав выражение с использованием специальных символов. Вот наиболее часто используемые из них:

Оператор Описание

. Один любой символ, кроме новой строки \n.

? 0 или 1 вхождение шаблона слева

+ 1 и более вхождений шаблона слева

\* 0 и более вхождений шаблона слева

\w Любая цифра или буква (\W — все, кроме буквы или цифры)

\d Любая цифра [0-9] (\D — все, кроме цифры)

\s Любой пробельный символ (\S — любой непробельный символ)

\b Граница слова

[..] Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)

\ Экранирование специальных символов (\. означает точку или \+ — знак «плюс»)

^ и $ Начало и конец строки соответственно

{n,m} От n до m вхождений ({,m} — от 0 до m)

a|b Соответствует a или b

() Группирует выражение и возвращает найденный текст

\t, \n, \r Символ табуляции, новой строки и возврата каретки соответственно