МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский технический университет связи и информатики»**

Кафедра

«Математическая кибернетика и информационные технологии»

Лабораторная работа №3

**«Методы поиска подстроки в строке»**

по дисциплине

**«**Структуры и алгоритмы обработки данных**»**

Выполнил:

студент гр. БСТ2004

Шадюк М.Р.

Вариант №22

Москва, 2022 г.

Оглавление

[Задание №1 3](#_Toc102095665)

[Алгоритм Кнута-Морриса-Пратта 3](#_Toc102095666)

[КОД 3](#_Toc102095667)

[Выполнение 5](#_Toc102095668)

[Упрощенный алгоритм Бойера-Мура 5](#_Toc102095669)

[КОД 5](#_Toc102095670)

[Выполнение 7](#_Toc102095671)

[Задание №2 7](#_Toc102095672)

[КОД 8](#_Toc102095673)

[Выполнение 10](#_Toc102095674)

# Задание №1

Реализовать методы поиска подстроки в строке. Добавить возможность ввода строки и подстроки с клавиатуры. Предусмотреть возможность существования пробела. Реализовать возможность выбора опции чувствительности или нечувствительности к регистру. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

## Алгоритм Кнута-Морриса-Пратта

### КОД

class search1:

    def \_\_init\_\_(self, strr):

        self.str=strr.replace(' ','') # удаление пробелов между символами

        self.strprob=strr

        self.strregistr=self.str.lower() #восприимчивость к любому регистру

    #создание префиксной функции

    def pref(self, podstr):

        length\_podstr=len(podstr)

        p=[0]\*length\_podstr

        p[0]=0

        k=0

        for q in range(1, length\_podstr):

            while k>0 and (podstr[k] is not podstr[q]):

                k=p[k]

            if podstr[k]==podstr[q]:

                k=k+1

            p[q]=k

        return p

    #поиск чувствительный к регистру

    def search(self, podstr):

        length\_str=len(self.str)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        q=0

        for i in range(0, length\_str):

            while q>0 and q<length\_podstr and (podstr[q] is not self.str[i]) :

                q=p[q]

            if podstr[q]==self.str[i]:

                q=q+1

            if q==length\_podstr:

                print("такая подстрока найдена:", i, " ", length\_podstr)

                q=p[q-1]

    #поиск чувствительный к регистру и к пробелам

    def searchregspl(self, podstr):

        length\_str=len(self.strprob)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        q=0

        for i in range(0, length\_str):

            while q>0 and q<length\_podstr and (podstr[q] is not self.strprob[i]) :

                q=p[q]

            if podstr[q]==self.strprob[i]:

                q=q+1

            if q==length\_podstr:

                print("такая подстрока найдена:", i, " ", length\_podstr)

                q=p[q-1]

     #поиск не чувствительный к регистру

    def searchreg(self, podstr):

        podstr=podstr.lower()

        length\_str=len(self.strregistr)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        q=0

        for i in range(0, length\_str):

            while q>0 and q<length\_podstr and (podstr[q] is not self.strregistr[i]) :

                q=p[q]

            if podstr[q]==self.strregistr[i]:

                q=q+1

            if q==length\_podstr:

                print("такая подстрока найдена:", i, " ", length\_podstr)

                q=p[q-1]

s=str(input("введите строку "))

a=search1(s)

m=str(input("введите подстроку "))

a.search(m)

a.searchreg(m)

a.searchregspl(m)

### Выполнение

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 – Результат работы программы

## Упрощенный алгоритм Бойера-Мура

### КОД

class search2:

    def \_\_init\_\_(self, strr):

        self.str=strr.replace(' ','') # удаление пробелов между символами

        self.strprob=strr

        self.strregistr=self.str.lower() #восприимчивость к любому регистру

    #создание префиксной функции

    def pref(self, podstr):

        length\_podstr=len(podstr)

        p=[0]\*(length\_podstr)

        k=1

        for q in range(0,length\_podstr):

            p[q]=length\_podstr-k

            k=k+1

        p[length\_podstr-1]=length\_podstr

        return p

     #поиск чувствительный к регистру

    def search(self, podstr):

        podstr=podstr.replace(' ','')

        length\_str=len(self.str)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        i=length\_podstr-1

        j=i

        st=0

        while i<length\_str:

            if self.str[i]==podstr[j]:

                if(j==0):

                    print("такая подстрока найдена", i, i+length\_podstr-1)

                    i=i+(p[length\_podstr-1]\*2)-1

                    j=length\_podstr-1

                else:

                    i=i-1

                    j=j-1

            else:

                while self.str[i] is not podstr[j]:

                    j=j-1

                    if j==-1:

                        j=length\_podstr-1

                        break

                i=i+p[j]

                j=length\_podstr-1

    # поиск не чувствительный к регистру

    def searchreg(self, podstr):

        podstr=podstr.replace(' ','')

        podstr=podstr.lower()

        length\_str=len(self.strregistr)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        i=length\_podstr-1

        j=i

        st=0

        while i<length\_str:

            if self.strregistr[i]==podstr[j]:

                if(j==0):

                    print("такая подстрока найдена", i, i+length\_podstr-1)

                    i=i+(p[length\_podstr-1]\*2)-1

                    j=length\_podstr-1

                else:

                    i=i-1

                    j=j-1

            else:

                while self.strregistr[i] is not podstr[j]:

                    j=j-1

                    if j==-1:

                        j=length\_podstr-1

                        break

                i=i+p[j]

                j=length\_podstr-1

         # поиск чувствительный к пробелам и к регистру

    def searchregspl(self, podstr):

        length\_str=len(self.strprob)

        length\_podstr=len(podstr)

        p=self.pref(podstr)

        i=length\_podstr-1

        j=i

        st=0

        while i<length\_str:

            if self.strprob[i]==podstr[j]:

                if(j==0):

                    print("такая подстрока найдена", i, i+length\_podstr-1)

                    i=i+(p[length\_podstr-1]\*2)-1

                    j=length\_podstr-1

                else:

                    i=i-1

                    j=j-1

            else:

                while self.strprob[i] is not podstr[j]:

                    j=j-1

                    if j==-1:

                        j=length\_podstr-1

                        break

                i=i+p[j]

                j=length\_podstr-1

s="sjnaaoannshhdbdi"

a=search2(s)

a.search("shhd")

a.searchreg("shhd")

a.searchregspl("shhd")

### Выполнение

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2 – Результат работы программы

# Задание №2

Написать программу, определяющую, является ли данное расположение «решаемым», то есть можно ли из него за конечное число шагов перейти к правильному. Если это возможно, то необходимо найти хотя бы одно решение - последовательность движений, после которой числа будут расположены в правильном порядке. Входные данные: массив чисел, представляющий собой расстановку в порядке «слева направо, сверху вниз». Число 0 обозначает пустое поле. Например, массив [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0] представляет собой «решенную» позицию элементов. Выходные данные: если решения нет, то функция должна вернуть пустой массив []. Если решение есть, то необходимо представить решение — для каждого шага записывается номер передвигаемого на данном шаге элемента.

### КОД

class Position:

    def \_\_init\_\_(self, position, start\_distance,finish\_distance):

        self.position = position

        self.start\_distance = start\_distance

        self.finish\_distance = finish\_distance

    def \_\_str\_\_(self):

        return '\n'.join((N\*'{:3}').format(\*[i%(N\*N) for i in self.position[i:]]) for i in range(0, N\*N, N))

    # Переопределяем метод less then для работы PriorityQueue

    def \_\_lt\_\_(self, other):

        return self.start\_distance+self.finish\_distance < other.start\_distance+other.finish\_distance

from queue import PriorityQueue

N = 4

# Генератор сдвигов

def shifts(position):

# Находим индекс нуля

    zeroPosition = position.index(0)

# Находим его позицию в поле 4х4, где  i это номер строки,а j номер столбца

    i, j = divmod(zeroPosition, N)

    displacement = []

# В зависимости от местоположения нуля, смотрим варианты, куда его можно сдвинуть

    if i > 0: displacement.append(-N)     # вверх

    if i < N - 1: displacement.append(N)  # вниз

    if j > 0: displacement.append(-1)     # влево

    if j < N - 1: displacement.append(1)  # вправо

    for offset in displacement:

# считаем индексы новых позиций нуля

        swap = zeroPosition + offset

# Выводим новое состояние поля, где 0 сдвинут в одном из возможных направлений

        yield tuple(position[swap] if x==zeroPosition else position[zeroPosition] if x==swap else position[x] for x in range(N\*N))

# Проверяет четность неправильных пар

def parityOfPairs(state):

    countOfPairs = 0

    for i in range(len(state)-1):

        if state[i] > state[i+1]:

            countOfPairs +=1

def fifteenGame(startState):

    terminalState = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)

    # Если количество неправильных пар четное, то такая позиция нерешаема

    if parityOfPairs(startState)==0:

        print("Нет решений")

    else:

    # В противном случае создаем из начальной позиции кортеж

        startState= tuple(startState)

    # создаем экзмепляр класса, с кортежем стартовой позиции в качестве параметра,длиной пути от начала до текущей точки

    # и длиной от текущей позиции до конца

        p = Position(startState, 0,0)

    # Выводим начальную позицию

        print(p)

        print()

    # Создаем экземпляр класса приоритетной очереди

        fieldStates= PriorityQueue()

    # Заносим в нее пару-это кортеж стартовой позиции, и вес данной вершины

        fieldStates.put(p)

    # Создаем множество посещенных вершин

        closePoints = set([p])

    # Создаем словарь в котором будем хранить позиции,где для каждой будет определена ее предыдущая позиция

        parents = {p.position: None}

    # Пока позиция не будет равна решению

        while p.position != terminalState:

    # Получаем приоритетную позицию

            p =fieldStates.get()

    # для каждого варианта передвижения нуля в одном состоянии поля

            for k in shifts(p.position):

                count= 0

    # Если такой вариант передвижения не находится в списке посещенных

                if k not in closePoints:

    # Расчитаем растояние до терминального состояния, это количество цифр стоящих не на своих местах

                    for m in range(len(k)):

                        if k[m] != terminalState[m]:

                            count+=1

    # То мы заносим его в очередь, с состоянием в качестве параметра, длиной пути от старта до текущего сосотояния

    # и длиной до финиша

                    fieldStates.put(Position(k, p.start\_distance +1,p.finish\_distance+count))

    # в словарь добавляем новый ключ, то есть эту позицию, его значением будет предыдущая позиция

                    parents[k] = p

    # Добавляем этот вариант перемещения в посещенный

                    closePoints.add(k)

        path = []

        x = p

        previous = p

        while p.position != startState:

    # Берем из словаря родителя текущего состояния поля

            p = parents[p.position]

    # Запоминаем индекс нуля из текущего состояния и по этому индексу находим элемент, который сдвинули в предыдущем состоянии

            number = p.position[previous.position.index(0)]

            path.append(number)

    # Предыдущее состояние делаем текущим

            previous = p

    # Разворачиваем путь, чтобы получить путь от первого до последнего сдвига

        path.reverse()

        print(path)

        print(x)

startState = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 14, 15, 13]

fifteenGame(startState)

### Выполнение

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 – Результат работы программы