**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МО ЭВМ**

# ОТЧЕТ

**по лабораторной работе №5**
**по дисциплине «Построение и анализ алгоритмов»**
**Тема: Поиск набора подстрок в строке.**

Студент гр. 1303 _____ Смирнов Д. Ю.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2023

**Цель работы.**

Изучить принцип работы алгоритма Ахо-Корасик. Решить с его помощью задачи.

**Задание 1.**

Разработайте программу, решающую задачу точного поиска набора образцов.

**Вход:**

Первая строка содержит текст ($T$, $1 \leq |T| \leq 100000$).

Вторая - число $n$ ($1 \leq n \leq 3000$), каждая следующая из $n$ строк содержит шаблон из набора $P = \{p_1, \dots, p_n\} 1 \leq |p_i| \leq 75$

Все строки содержат символы из алфавита $\{A,C,G,T,N\}$

**Выход:**

Все вхождения образцов из $P$ в $T$.

Каждое вхождение образца в текст представить в виде двух чисел - $i$ $p$

Где $i$ - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером $p$ (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

**Sample Input:**

NTAG

3

TAGT

TAG

T

**Sample Output:**

2 2

2 3

**Задание 2.**

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу *P* необходимо найти все вхождения P в текст T.

Например, образец *ab??c?* с джокером *?* встречается дважды в тексте *xabvccbababcax.*

Символ джокер не входит в алфавит, символы которого используются в *T*. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида *???* недопустимы.

Все строки содержат символы из алфавита {*A, C, G, T, N*}

**Вход:**

Текст ($T$, $1 \leq |T| \leq 100000$)

Шаблон ($P$, $1 \leq |P| \leq 40$)

Символ джокера

**Выход:**

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

**Sample Input:**

ACTANCA

A$$A$

$

**Sample Output:**

1

Вариант 5. Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вы́резать из строки поиска все найденные образцы и вывести остаток строки поиска.

**Выполнение работы.**

**Описание алгоритма Ахо-Корасик.**

Алгоритм создает префиксное дерево из букв искомых подстрок. Затем в полученном дереве ищутся суффиксные ссылки.

Суффиксная ссылка для каждой вершины u — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине u. Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет.

Побуквенно проходимся по тексту, каждую букву передавая в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по суффиксной ссылке.

Если встреченная вершина является терминальной или она имеет суффиксную ссылку на терминальную вершину, значит была встречена подстрока. Индекс этой подстроки хранится в поле *patternID* вершины.

В ответ сохранятся индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

**Сложность по времени:**

Т.к при построении префиксного дерева запускается цикл по длине каждой подстроки (суммарная длина подстрок - $n$), и из каждой вершины может исходить максимум $k$ ребер (где $k$ – размер алфавита), то построение префиксного дерева происходит за O($n*k$)

Алгоритм в цикле проходит по тексту длины s: O($s$)

Также $t$ — количество всех возможных вхождений всех строк-образцов в $s$.

Итого: O($n*k + s + t$)

**Сложность по памяти:**

Алгоритм создает префиксное дерево с $n$ вершинами, каждая вершина хранит массив вершин, инцидентных ей, размером $k$ ($k$ – размер алфавита).

Итого: O(n*k)

**Описание алгоритма для нахождения шаблонов с маской.**

Алгоритм тот же, но в качестве подстрок берутся кусочки шаблона, разделенные джокером, запоминается начальная позиции полученных подстрок в исходном шаблоне.

Создается массив $C$ длины $s$, где $s$ – длина текста, где ищется шаблон.

При нахождении подстроки, в массиве $C$ увеличивается на единицу число по индексу, соответствующему возможному началу шаблона. Индекс высчитывается по формуле: текущий индекс – индекс найденной подстроки в шаблоне.

Затем проходим по полученному массиву, каждый $i$ для которого $C[i]$ = количеству подстрок, является началом шаблона.

**Сложность по времени для модифицированного алгоритма:**

Затраты по времени такие же как в обычном алгоритме, но дополнительно проход по массиву $C$ длины $s$:

Итого: O($n*k + s + t$)

**Сложность по памяти для модифицированного алгоритма:**

Затраты по памяти такие же как в обычном алгоритме, но дополнительно создается массив $C$ длины $s$.

Затраты по памяти O($n*k + s$)

**Описание классов**

***class Node***

Класс, отвечает за хранение данных вершины в Боре.

Поля класса:

- *prefix* – имя вершины
- *moves* – словарь инцидентных ребер вершины
- *suffixLink* – суффиксная ссылка для вершины
- *terminalFlag* – флаг терминальной вершины
- *patternsID* – номер шаблона

Методы класса:

- *setTerminal(self, patternID)* – Метод класса делает вершину терминальной и устанавливает в неё номер паттерна. Аргументы: patternID – номер шаблона. Метод ничего не возвращает.

**Описание переменных и функций**

Глобальная переменная *DEBUG* – отвечает за переключение дополнительного вывода.

**Функции:**

- *buildTrie(patterns)* – Функция строит префиксное дерево из шаблонов. Аргументы: *patterns* – список шаблонов. Функция возвращает корень префиксного дерева.
- *setSuffixLinks(root)* – Функция для каждой вершины префиксного дерева устанавливает суффиксную ссылку. Аргументы: *root* – корень префиксного дерева. Функция возвращает корень автомата.
- *algorithmAho(text, patterns)* – Функция реализует описанный выше алгоритм Ахо-Корасика. Аргументы функции: *text* – строка в которой ищут вхождения шаблонов, *patterns* – список шаблонов. Функция возвращает список из кортежей вида (индекс вхождения шаблона в текст, индекса шаблона в *patterns*).

- *findMaxEdges(root)* – Функция находит все вершины имеющие максимальное количесиво ребер в префиксном дереве. Аргументы: root – корень префиксного дерева. Функция ничего не возвращает.
- *treeDFS(root)* – Функция выводит автомат в консоль. Аргументы: root – корень автомата. Функция ничего не возвращает.
- *createSubsAndStarts(pattern, joker*) – Функция для второго задания, делит строку шаблон по символу “джокеру”, находит для каждой подстроки индекс её начала в шаблоне. Аргументы: *pattern* – шаблон с символами “джокерами”, *joker* – символ “джокер”. Функция возвращает кортеж из подстрок и список из кортежей вида (подстрока, индекс её в хождения в шаблон).
- solve(text, pattern, joker) – Функция реализует описанный выше алгоритм для нахождения шаблонов с маской в строке. Аргументы: text – строка в которой ищут шаблон, pattern – шаблон, joker – символ “джокер”. Функция возвращает список из индексов вхождения *pattern* в *text*.

**Тестирование.**

Результаты тестирования двух алгоритмов приведены в таблице 1. Пример дополнительного вывода, представлен на рисунка 1 и 2.

Таблица 1 – Тестирование алгоритмов

| Тестирование алгоритма 1 | | | |
|---|---|---|---|
| № | Входные данные | Выходные данные | Комментарий |
| 1 | NTAG<br>3<br>TAGT<br>TAG<br>T | 2 2<br>2 3 | Верно |
| 2 | 123321<br>2<br>23<br>21 | 2 1<br>5 2 | Верно |
| 3 | BANGBANG<br>2<br>ANG<br>GBA | 2 1<br>4 2<br>6 1 | Верно |
| Тестирование алгоритма 2 | | | |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | ACTANCA<br>A$$A$<br>$ | 1 | Верно |
| 2 | ACACAC<br>AC$$<br>$ | 1<br>3 | Верно |
| 3 | ACACAA<br>AC$A<br>$ | 3 | Верно |

```
NTAG
3
TAGT
TAG
T


BUILDING A TRIE
Go from the ROOT
Now add pattern[TAGT] to trie
Add new node T to node
Add new node TA to node T
Add new node TAG to node TA
Add new node TAGT to node TAG
STATE[TAGT] has become terminal
Go from the ROOT
Now add pattern[TAG] to trie
Go to node T
Go to node TA
Go to node TAG
STATE[TAG] has become terminal
Go from the ROOT
Now add pattern[T] to trie
Go to node T
STATE[T] has become terminal


Install suffix links in the trie
Add suffix link to ROOT from T
Add suffix link FROM:TA TO: root
Add suffix link FROM:TAG TO: root
Add suffix link FROM:TAGT TO: T


THE AUTOMATON
NODE[root] - TERMINAL?[False] - SUFFIXLINK TO []
    NODE[T] - TERMINAL?[True] - SUFFIXLINK TO [root]
        NODE[TA] - TERMINAL?[False] - SUFFIXLINK TO [root]
            NODE[TAG] - TERMINAL?[True] - SUFFIXLINK TO [root]
                NODE[TAGT] - TERMINAL?[True] - SUFFIXLINK TO [T]
The max number of edges (1) comes out of ['root', 'T', 'TA', 'TAG']


Aho-Corasick algorithm STARTED!
CURRENT POSITION IN TEXT 1 | CHAR IN TEXT: N
CURRENT STATE: [root]
From state[root] noway to state[root->N]
Go to suffixLink state[]
Didn't find the path to sym returned to the ROOT
CURRENT POSITION IN TEXT 2 | CHAR IN TEXT: T
```

Рисунок 1- дополнительный вывод алгоритма 1 для теста 1

```
ACTANCA
A$$A$
$
Split the pattern into substrings by the joker symbol [$]
After splitting ['A', 'A']
Array of tuples of subs and their start index in the pattern
[('A', 0), ('A', 3)]


BUILDING A TRIE
Go from the ROOT
Now add pattern[A] to trie
Add new node A to node
STATE[A] has become terminal
Go from the ROOT
Now add pattern[A] to trie
Go to node A
STATE[A] has become terminal


Install suffix links in the trie
Add suffix link to ROOT from A


THE AUTOMATON
NODE[root] - TERMINAL?[False] - SUFFIXLINK TO []
    NODE[A] - TERMINAL?[True] - SUFFIXLINK TO [root]
The max number of edges (1) comes out of ['root']


Aho-Corasick algorithm STARTED!
CURRENT POSITION IN TEXT 1 | CHAR IN TEXT: A
CURRENT STATE: [root]
State[A] find A with id 0 at position 0
State[A] find A with id 1 at position 0
CURRENT POSITION IN TEXT 2 | CHAR IN TEXT: C
CURRENT STATE: [A]
```

Рисунок 2 - дополнительный вывод алгоритма 2 для теста 1

**Выводы.**

Был изучен принцип работы алгоритма Ахо-Корасик для нахождения набора подстрок в строке.

# ПРИЛОЖЕНИЕ А

# ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: task1.py

```
import math

DEBUG = False


class Node:
    def __init__(self, preffix='root'):
        self.preffix = preffix
        self.moves = {}
        self.suffixLink = None
        self.terminalFlag = False
        self.patternsID = []

    def setTerminal(self, patternID):
        global DEBUG
        if patternID not in self.patternsID:
            self.patternsID += [patternID]
        if DEBUG:
            print(f"STATE[{self.preffix}] has become terminal")
        self.terminalFlag = True


def buildTrie(patterns):
    """
    Func build trie by patterns
    :param patterns: array of patterns
    :return: root node of trie
    """
    global DEBUG
    if DEBUG:
        print("\n\nBUILDING A TRIE")
    root = Node()
    for i, pattern in enumerate(patterns):
        if DEBUG:
            print("Go from the ROOT")
            print(f"Now add pattern[{pattern}] to trie")
        node = root
        # for each pattern go sym
        for indx, sym in enumerate(pattern):
            # add node to trie if not already exist in trie
            if DEBUG:
                if sym in node.moves:
                    print(f"Go to node {node.moves[sym].preffix}")
                else:
                    print(f"Add new node {pattern[:indx + 1]} to node
{pattern[:indx]}")
            node = node.moves.setdefault(sym, Node(pattern[:indx + 1]))
        node.setTerminal(i)
    return root


def setSuffixLinks(root):
    """
    Make from trie to automaton
    :param root: root of trie
    :return: return root of automaton
    """
```

```
    global DEBUG
    if DEBUG:
        print("\n\nInstall suffix links in the trie")
    queue = []
    # add suffixLink to root from root childrens
    for vertex in root.moves.values():
        if DEBUG:
            print(f"Add suffix link to ROOT from {vertex.preffix}")
        vertex.suffixLink = root
        queue.append(vertex)

    while queue:
        vertex = queue.pop(0)

        for char, childVertex in vertex.moves.items():

            queue.append(childVertex)
            linkedNode = vertex.suffixLink
            # Climbing the suffix link
            while linkedNode is not None and char not in linkedNode.moves:
                linkedNode = linkedNode.suffixLink

            # If find add suffixLink, else add suffixlink to root
            if linkedNode:
                childVertex.suffixLink = linkedNode.moves[char]
            else:
                childVertex.suffixLink = root
            if DEBUG:
                print(f"Add suffix link FROM:{childVertex.preffix} TO:
{childVertex.suffixLink.preffix}")
            # Transferring all patterns from the suffix link
            childVertex.patternsID += childVertex.suffixLink.patternsID

    return root


def algorithmAho(text, patterns):
    """
    The function is performed by the Aho-Korasik algorithm
    :param text:
    :param patterns: array of pattern,
    :return:
    """
    global DEBUG
    root = setSuffixLinks(buildTrie(patterns))
    if DEBUG:
        treeDFS(root)
        findMaxEdges(root)
        print("\n\nAho-Corasick algorithm STARTED!")
    node = root
    res = []
    for i in range(len(text)):
        if DEBUG:
            print(f"CURRENT POSITION IN TEXT {i + 1} | CHAR IN TEXT:
{text[i]}")
            print(f"CURRENT STATE: [{node.preffix}]")
        while node and text[i] not in node.moves:
            if DEBUG:
                print(f"From state[{node.preffix}] noway to
state[{node.preffix}->{text[i]}]")
                print(f"Go to suffixLink state[{node.suffixLink.preffix if
node.suffixLink else ''}]")
            node = node.suffixLink
        if not node:
```

```
            if DEBUG:
                print("Didn't find the path to sym returned to the ROOT")
            node = root
            continue
        node = node.moves[text[i]]
        for pattern in node.patternsID:
            if DEBUG:
                print(
                    f"State[{node.preffix}] find {patterns[pattern]} with id
{pattern} at position {i - len(patterns[pattern]) + 1}")
            res.append((i - len(patterns[pattern]) + 1, pattern))
    return res


def findMaxEdges(root):
    """
    Func find all node witch have max edges
    :param root: root of trie
    :return: nothing
    """
    maxEdges = -math.inf
    nodes = []
    queue = [root]
    # using DFS find nodes
    while queue:
        currentNode = queue.pop(0)
        if len(currentNode.moves.values()) > maxEdges:
            # if max > current refresh list of nodes
            nodes.clear()
            nodes.append(currentNode)
            maxEdges = len(currentNode.moves.values())
        elif len(currentNode.moves.values()) == maxEdges:
            # find another node with max edges
            nodes.append(currentNode)
        for child in currentNode.moves.values():
            # add children in queue
            queue.append(child)
    print(f"The max number of edges ({maxEdges}) comes out of {[node.preffix
for node in nodes]}")


def treeDFS(root):
    """
    Print automaton
    :param root: root node of Trie
    :return: nothing
    """
    print("\n\nTHE AUTOMATON")
    visited = []
    queue = [root]
    while queue:
        currentNode = queue.pop(0)
        if currentNode not in visited:
            addSpacing = "\t" * len(currentNode.preffix) if
currentNode.preffix != "root" else ""
            print(
                f"{addSpacing}NODE[{currentNode.preffix}] -
TERMINAL?[{currentNode.terminalFlag}] - SUFFIXLINK TO
[{currentNode.suffixLink.preffix if currentNode.suffixLink else ''}]")
            visited += [currentNode]
            for child in currentNode.moves.values():
                queue = [child] + queue
```

```
if __name__ == "__main__":
    text = input()
    patterns = [input() for _ in range(int(input()))]

    result = sorted(algorithmAho(text, patterns))
    print("\n".join([f"{ind + 1} {pattern + 1}" for ind, pattern in result]))
    if DEBUG:
        for ind, pattern in result:
            patternLen = len(patterns[pattern])
            text = text[:ind] + " " * patternLen + text[ind + patternLen:]
        print("Having cut out all the patterns from the text left:",
text.replace(' ', ''), sep='\n')
```

Название файла: task2.py

```
import math

DEBUG = False


class Node:
    def __init__(self, preffix='root'):
        self.preffix = preffix
        self.moves = {}
        self.suffixLink = None
        self.terminalFlag = False
        self.patternsID = []

    def setTerminal(self, patternID):
        global DEBUG
        if patternID not in self.patternsID:
            self.patternsID += [patternID]
        if DEBUG:
            print(f"STATE[{self.preffix}] has become terminal")
        self.terminalFlag = True


def buildTrie(patterns):
    global DEBUG
    if DEBUG:
        print("\n\nBUILDING A TRIE")
    root = Node()
    for i, pattern in enumerate(patterns):
        if DEBUG:
            print("Go from the ROOT")
            print(f"Now add pattern[{pattern}] to trie")
        node = root
        for indx, sym in enumerate(pattern):
            if DEBUG:
                if sym in node.moves:
                    print(f"Go to node {node.moves[sym].preffix}")
                else:
                    print(f"Add new node {pattern[:indx + 1]} to node
{pattern[:indx]}")
            node = node.moves.setdefault(sym, Node(pattern[:indx + 1]))
        node.setTerminal(i)
    return root


def setSuffixLinks(root):
    global DEBUG
    if DEBUG:
        print("\n\nInstall suffix links in the trie")
    queue = []
    for vertex in root.moves.values():
```

```
        if DEBUG:
            print(f"Add suffix link to ROOT from {vertex.preffix}")
        vertex.suffixLink = root
        queue.append(vertex)

    while queue:
        vertex = queue.pop(0)

        for char, childVertex in vertex.moves.items():

            queue.append(childVertex)
            linkedNode = vertex.suffixLink
            # Climbing the suffix link
            while linkedNode is not None and char not in linkedNode.moves:
                linkedNode = linkedNode.suffixLink

            # If find add suffixLink, else add suffixlink to root
            if linkedNode:
                childVertex.suffixLink = linkedNode.moves[char]
            else:
                childVertex.suffixLink = root
            if DEBUG:
                print(f"Add suffix link FROM:{childVertex.preffix} TO:
{childVertex.suffixLink.preffix}")
            # Transferring all patterns from the suffix link
            childVertex.patternsID += childVertex.suffixLink.patternsID

    return root


def algorithmAho(text, patterns):
    """
    The function is performed by the Aho-Korasik algorithm
    :param text:
    :param patterns: array of pattern,
    :return:
    """
    global DEBUG
    root = setSuffixLinks(buildTrie(patterns))
    if DEBUG:
        treeDFS(root)
        findMaxEdges(root)
        print("\n\nAho-Corasick algorithm STARTED!")
    node = root
    res = []
    for i in range(len(text)):
        if DEBUG:
            print(f"CURRENT POSITION IN TEXT {i + 1} | CHAR IN TEXT:
{text[i]}")
            print(f"CURRENT STATE: [{node.preffix}]")
        while node and text[i] not in node.moves:
            if DEBUG:
                print(f"From state[{node.preffix}] noway to
state[{node.preffix}->{text[i]}]")
                print(f"Go to suffixLink state[{node.suffixLink.preffix if
node.suffixLink else ''}]")
            node = node.suffixLink
        if not node:
            if DEBUG:
                print("Didn't find the path to sym returned to the ROOT")
            node = root
            continue
        node = node.moves[text[i]]
        for pattern in node.patternsID:
```

```
            if DEBUG:
                print(
                    f"State[{node.preffix}] find {patterns[pattern]} with id
{pattern} at position {i - len(patterns[pattern]) + 1}")
            res.append((i - len(patterns[pattern]) + 1, pattern))
    return res


def findMaxEdges(root):
    """
    Func find all node witch have max edges
    :param root: root of trie
    :return: nothing
    """
    maxEdges = -math.inf
    nodes = []
    queue = [root]
    # using DFS find nodes
    while queue:
        currentNode = queue.pop(0)
        if len(currentNode.moves.values()) > maxEdges:
            # if max > current refresh list of nodes
            nodes.clear()
            nodes.append(currentNode)
            maxEdges = len(currentNode.moves.values())
        elif len(currentNode.moves.values()) == maxEdges:
            # find another node with max edges
            nodes.append(currentNode)
        for child in currentNode.moves.values():
            # add children in queue
            queue.append(child)
    print(f"The max number of edges ({maxEdges}) comes out of {[node.preffix
for node in nodes]}")


def treeDFS(root):
    """
    Print automaton
    :param root: root node of Trie
    :return: nothing
    """
    print("\n\nTHE AUTOMATON")
    visited = []
    queue = [root]
    while queue:
        currentNode = queue.pop(0)
        if currentNode not in visited:
            addSpacing = "\t" * len(currentNode.preffix) if
currentNode.preffix != "root" else ""
            print(
                f"{addSpacing}NODE[{currentNode.preffix}] -
TERMINAL?[{currentNode.terminalFlag}] - SUFFIXLINK TO
[{currentNode.suffixLink.preffix if currentNode.suffixLink else ''}]")
            visited += [currentNode]
            for child in currentNode.moves.values():
                queue = [child] + queue


def createSubsAndStarts(pattern, joker):
    """
    func spilt pattern by joker, and find for each sub start index in pattern
    :param pattern: pattern with jockers
    :param joker: wildcard
    :return: subs, array of tuples with
```

```
    """
    global DEBUG
    patterns = list(filter(None, pattern.split(joker)))
    if DEBUG:
        print(f"Split the pattern into substrings by the joker symbol
[{joker}]")
        print(f"After splitting {patterns}")
    starts = []
    if pattern[0] != joker:
        starts.append(0)
    for i in range(1, len(pattern)):
        if pattern[i - 1] == joker and pattern[i] != joker:
            starts.append(i)
    if DEBUG:
        print(f"Array of tuples of subs and their start index in the
pattern")
        print(list(zip(patterns, starts)))
    return patterns, list(zip(patterns, starts))


def solve(text, pattern, joker):
    """
    func make prework computation, after start aho-korasik alg
    :param text: source text
    :param pattern: pattern with masks
    :param joker: wildcard
    :return: return array of index occurrence pattern in text
    """
    global DEBUG
    patterns, tupleInfo = createSubsAndStarts(pattern, joker)
    occurrences = algorithmAho(text, patterns)
    arrayC = len(text) * [0]
    for occurrence in occurrences:
        j = occurrence[0] - tupleInfo[occurrence[1]][1]
        if j >= 0:
            if DEBUG:
                print(f"For pattern occurence {tupleInfo[occurrence[1]][0]}
with startIndex {tupleInfo[occurrence[1]][1]} update arrayC[{j}]+=1")
            arrayC[j] += 1
    answer = []
    if DEBUG:
        print("Array C:")
        print(arrayC)
    for i in range(len(text) - len(pattern) + 1):
        if arrayC[i] == len(patterns):
            answer.append(i + 1)
    return answer


if __name__ == "__main__":
    text = input()
    pattern = input()
    joker = input()
    result = solve(text, pattern, joker)
    print(*result, sep='\n')
    if DEBUG:
        patternLen = len(pattern)
        for i in result:
            text = text[:i - 1] + " " * patternLen + text[i - 1 +
patternLen:]
        print("Having cut out all the patterns from the text left:",
text.replace(' ', ''), sep='\n')
```