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: Алгоритм Ахо-Корасика

Студент гр. 1304 Шаврин А.П.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить и реализовать метод точного поиска набора образцов в строке при помощи алгоритма Ахо-Корасика.

Задание.

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 $P = \{p_1, \ldots, p_n\}\ 1 \leq |p_i| \leq 75$
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.

Рисунок 1. Задание 1.

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.

Рисунок 2. Задание 2.

Выполнение работы.

1. Сперва был написан класс узла структуры Бор.

Данный класс имеет следующие публичные поля:

- self.children_nodes – словарь дочерних узлов, где ключ – новый суффикс дочернего узла
- self.patterns_indexes – список индексов искомых подстрок, которые равны данному узлу.
- self.suffix_link – суффиксная ссылка на узел, являющийся самым большим суффиксом данного узла

Данный класс имеет следующие методы:

- __init__ - инициализирует все значения класса

2. Затем был реализован класс Solver, который решает обе задачи.

Данный класс имеет следующие приватные поля:

- self.__task_type = task_type – строковая переменная определяющая тип решаемой задачи (“Classic” или “Joker”)
- self.__bohr_root = BohrNode() – корневой узел структуры Бор
- self.__text = str() – текст, в котором ищутся подстроки
- self.__patterns = list() – список искомых подстрок
- self.__answer = list() – ответ на задачу (список пар или список индексов вхождения, в зависимости от задачи)
- self.__count_patterns = int() – количество подстрок
- self.__pattern_parts_indexes = list() – индексы вхождения не маскируемых подстрок в подстроку с джокером
- self.__joker_pattern = str() – подстрока с джокерами
- self.__joker_symbol = str() – символ джокера

Данный класс имеет следующие методы:

- __init__ - инициализирует начальные параметры (входной параметр task_type отвечает за тип задачи)
- __classic_input – считывание входных данных для классического алгоритма
- __joker_input - считывание входных данных для алгоритма с джокерами
- __format_joker_input – приводит входные данный к классическому виду (в self.__patterns записываются все не маскированные подстроки в подстроке с джокером и заполняется список индексов их вхождения в подстроку с джокерами)
- read_data – считывает данные нужным методом (в зависимости от параметра self.__task_type)
- __build_bohr – строит структуру Бор
- __build_suffix_links – строит суффиксные ссылки в структуре Бор
- __aho_korasik_algorithm – решает классическую задачу алгоритмом Ахо-Корасик
- __joker_aho_korasik_algorithm – решает задачу с джокерами, используя классический алгоритм Ахо-Корасика. Сначала находятся все вхождения не маскированных подстрок паттерна в текст. Затем создается массив - счетчик, в котором в индексе, соответствующем индексу в тексте, хранится количество попавших не маскированных подстрок паттерна в нужном порядке и в нужных местах. Если это количество равно количеству не маскированных подстрок паттерна, то этот индекс является ответом.
- solve – вызывает необходимый метод решения задачи (в зависимости от параметра self.__task_type)
- print_answer – выводит ответ в формате нужном для той или иной задачи (в зависимости от параметра self.__task_type)

3. В конце реализован метод main, в котором создается обьект класса Solver и вызываются все необходимые его методы, для решения задачи.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	NTAG 3 TAGT TAG T	2 2 2 3
2.	ACTANCA A$$A$ $	1

Выводы.

Изучен и реализован метод точного поиска набора образцов в строке при помощи алгоритма Ахо-Корасика.

Для решения первой задачи используется классический алгоритм решения задачи точного поиска набора образцов в строке. Алгоритм Ахо-Корасика решает классическую задачу за линейное время.

Для решения второй задачи используется тот же классический алгоритм, однако входные данные были приведены к классическому виду, а затем, результат работы алгоритма, обработан для нахождения ответа на поставленную задачу. Поиск подстрок заданного шаблона с помощью алгоритма Ахо-Корасик выполняется за время O(m+n+a), где n — суммарная длина подстрок, то есть длина шаблона, m — длина текста, a — количество появлений подстрок шаблона.

Данный алгоритм является одним из наиболее эффективных для поиска всех заданных паттернов в тексте.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class BohrNode:
    """
    This is the node class of the Bohr structure
    """

    def __init__(self) -> None:
        """
        This method initializes the initial node parameters needed to solve the problem
        """

        self.children_nodes = {}
        self.patterns_indexes = []
        self.suffix_link = None


class Solver:
    """
    This class solves the problem of finding occurrences of a substring in a string
    """

    def __init__(self, task_type: str = "Classic") -> None:
        """
        This method initializes the initial parameters needed to solve the problem
        """

        self.__task_type = task_type
        self.__bohr_root = BohrNode()
        self.__text = str()
        self.__patterns = list()
        self.__answer = list()

        self.__count_patterns = int()
        self.__pattern_parts_indexes = list()
        self.__joker_pattern = str()
        self.__joker_symbol = str()

    def __classic_input(self) -> None:
        """
        This method reads the input data for the classical algorithm
        :return: None
        """

        self.__text = input()
        self.__count_patterns = int(input())
        self.__patterns = [input() for _ in range(self.__count_patterns)]

    def __joker_input(self) -> None:
```

```
            """
            This method reads the input data for the algorithm with
joker
            :return: None
            """

            self.__text = input()
            self.__joker_pattern = input()
            self.__joker_symbol = input()

        def __format_joker_input(self) -> None:
            """
            This method brings the given to the classical form, for
using the classical algorithm
            :return: None
            """

            self.__patterns             =             list(filter(bool,
self.__joker_pattern.split(self.__joker_symbol)))
            self.__count_patterns = len(self.__patterns)
            is_joker_symbol = True

            for index, symbol in enumerate(self.__joker_pattern):
                if symbol == self.__joker_symbol:
                    is_joker_symbol = True
                    continue
                if is_joker_symbol:
                    self.__pattern_parts_indexes.append(index)
                    is_joker_symbol = False

        def read_data(self) -> None:
            """
            This method reads given by the desired method, depending on
the type of task
            :return: None
            """

            if self.__task_type == "Joker":
                self.__joker_input()
                self.__format_joker_input()
            else:
                self.__classic_input()

        def __build_bohr(self) -> None:
            """
            This method builds a bohr structure
            :return: None
            """

            for pattern_index in range(self.__count_patterns):
                current_node = self.__bohr_root

                for symbol in self.__patterns[pattern_index]:
                    current_node                                        =
current_node.children_nodes.setdefault(symbol, BohrNode())

                current_node.patterns_indexes.append(pattern_index)
```

```
        def __build_suffix_links(self) -> None:
            """
            This method builds suffix links in the Bohr using a
breadth-first crawl
            :return: None
            """

            queue = []
            for child_node in self.__bohr_root.children_nodes.values():
                child_node.suffix_link = self.__bohr_root
                queue.append(child_node)

            while len(queue) > 0:
                current_node = queue.pop(0)

                for           symbol,           child_node           in
current_node.children_nodes.items():
                    queue.append(child_node)
                    current_suffix_node = current_node.suffix_link

                    while (current_suffix_node is not None) and (symbol
not in current_suffix_node.children_nodes.keys()):
                        current_suffix_node                            =
current_suffix_node.suffix_link

                    if current_suffix_node:
                        child_node.suffix_link                         =
current_suffix_node.children_nodes[symbol]
                    else:
                        child_node.suffix_link = self.__bohr_root
                    child_node.patterns_indexes                       +=
child_node.suffix_link.patterns_indexes

        def __aho_korasik_algorithm(self) -> None:
            """
            This method solves the problem using the classical Aho-
Korasik algorithm
            :return: None
            """

            current_node = self.__bohr_root

            for symbol_index, symbol in enumerate(self.__text):
                while (current_node is not None) and (symbol not in
current_node.children_nodes):
                    current_node = current_node.suffix_link

                if current_node is None:
                    current_node = self.__bohr_root
                    continue

                current_node = current_node.children_nodes[symbol]
                for pattern_index in current_node.patterns_indexes:
                    self.__answer.append((symbol_index                 -
len(self.__patterns[pattern_index]) + 1, pattern_index))

            self.__answer = sorted(self.__answer)
```

```
        def __joker_aho_korasik_algorithm(self) -> None:
            """
            This method solves the problem with joker using the
classical Aho-Korasik algorithm
            :return: None
            """

            self.__aho_korasik_algorithm()

            counter = [0] * len(self.__text)
            for aho_index, pattern_index  in self.__answer:
                joker_pattern_occurance_index     =     aho_index     -
self.__pattern_parts_indexes[pattern_index]
                if (joker_pattern_occurance_index >= 0) and \
                        (joker_pattern_occurance_index                 <
len(self.__text)):
                    counter[joker_pattern_occurance_index] += 1

            joker_answer = list()
            for       index       in       range(len(self.__text)       -
len(self.__joker_pattern) + 1):
                if counter[index] == len(self.__patterns):
                    joker_answer.append(index + 1)
            self.__answer = joker_answer

        def solve(self) -> None:
            """
            This method solves the problem with the right method,
depending on the type of task
            :return: None
            """

            self.__build_bohr()
            self.__build_suffix_links()
            if self.__task_type == "Joker":
                self.__joker_aho_korasik_algorithm()
            else:
                self.__aho_korasik_algorithm()

        def print_answer(self) -> None:
            """
            This method outputs the answer in the format that a
particular task requires
            :return: None
            """

            if self.__task_type == "Joker":
                print(*self.__answer, sep='\n')
            else:
                for pair in self.__answer:
                    print(f"{pair[0] + 1} {pair[1] + 1}")


    def main():
        """
        This function solves the problem
        :return: None
```

```
    """
    #solver = Solver("Joker")
    solver = Solver()
    solver.read_data()
    solver.solve()
    solver.print_answer()


if __name__ == "__main__":
    main()
```