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: Ахо-Корасик

Студент гр. 3388 _____ Еникеев А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы

Разработать программу, решающую задачу точного поиска набора образцов в тексте, используя алгоритм Ахо-Корасика и конечных (сжатых) ссылок для оптимизации. На основе полученного алгоритма дополнительно решить задачу точного поиска для одного образца с джокером. Проанализировать сложность алгоритмов.

Задание

Вар. 3

Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

Пункт 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Входные данные

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

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

Выходные данные

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.

Пункт 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}*.

Входные данные

Текст $(T, 1 \leq |T| \leq 100000)$

Шаблон $(P, 1 \leq |P| \leq 40)$

Символ джокера

Выходные данные

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

Номера должны выводиться в порядке возрастания.

Выполнение работы

Построение бора

Класс *AhoCorasick* хранит бор в виде списка словарей *trie*, где каждый элемент списка — узел, а ключи словаря — символы для переходов. Например, если первый узел содержит ребра (переходы) *a, b, c*, то первый элемент списка будет содержать словарь *{'a': 1, 'b': 2, 'c': 3}*, где ключ каждого символа соответствует узлу, который продолжает префикс, если вершина является конечной (терминальной), то соответствующий переход ссылается на пустой словарь *{}*.

За добавление шаблона в структуру бора отвечает метод *add_pattern()*, он перебирает символы в шаблоне, если символ есть в боре, то выполняет переход к следующему узлу (начинает с узла 0), иначе добавляет символ в бор. После окончания перебора символов добавляется данные о терминальном узле в список кортежей *output*.

Каждый узел бора имеет определенный индекс, который определяет его местоположение в списке переходов узлов *trie*, в списке *output* (содержит данные о терминальности узла), в списке суффексных ссылок *fail* и в списке сжатых ссылок *dict_suffix*.

Построение автомата

Автомат Ахо-Корасик расширяет структуру бора, добавляя два типа ссылок: суффиксные (*fail*) и сжатые (*dict_suffix*). Эти ссылки позволяют эффективно переходить между состояниями автомата при несовпадении символов и ускоряют поиск терминальных узлов.

Реализация ссылок выполняется методом *build()*, который использует *BFS*-обход (поиск в ширину). Это гарантирует, что ссылки для родительских узлов вычисляются раньше, чем для дочерних.

Разберем этапы работы метода:

- Инициализируем очередь для потомков корня: их суффиксные ссылки и сжатые ссылки устанавливаются в корень (0), так как для первого уровня нет более коротких суффиксов.
- Обработка узлов в порядке *BFS*
- Выбирается узел. Для каждого его дочернего узла (перехода по символу): добавляем в очередь (для последующей обработки), поднимаемся по цепочке *fail*-ссылок (начинаем с *fail*-ссылки родительского узла), пока не дойдем до корня или узла, из которого есть переход по текущему символу. Суффиксная ссылка дочернего узла устанавливается в найденный переход или корень.
- Если суффиксная ссылка (*fail*) дочернего узла ведет в терминальный узел (с непустым *output*), то *dict_suffix* копирует *fail*. Иначе *dict_suffix* наследуется от *dict_suffix* узла, на который указывает *fail*.

Поиск вхождений

Метод *search_all()* выполняет поиск всех вхождений точных шаблонов в тексте, используя построенный автомат Ахо-Корасик.

Для начала рассмотрим вспомогательный метод *get_next_node()*, который вычисляет узел, соответствующий текущему префиксу текста:

- Использует прямые переходы в боре
- При отсутствии перехода поднимается по суффиксным ссылкам (fail), пока не найдет допустимый переход или не вернется в корень
- Кэширует найденный переход в текущем узле для ускорения последующих запросов, то есть устанавливает текущий переход из вершины к найденной, чтобы избежать последующего перемещения по цепочке суффиксных ссылок

Теперь рассмотрим работу метода *search_all()*:

- Обход автомата начинает с корневого узла. Циклом обрабатываем каждый символ текста. *occurrences* - список результатов в формате *(start_position, pattern_index)*
- Находим методом *get_next_node()* узел, который соответствует текущему префиксу
- Создаем временную переменную *temp* (индекс узла), идем по цепочке терминальных ссылок, добавляя терминальные вершины в список *occurrences*

Таким образом, для каждого символа алгоритм останавливает на узле, которые завершает текущий префикс, проверяет является ли данный узел терминальным, а также все сжатые ссылки узла. Так гарантируется правильная работа автомата, например, если текущей символ не встречался ни в одном шаблоне, то автомат остановиться на корне.

Поиск одного образца с джокером

Алгоритм реализует функция *find_wildcard_matches()*, которая не является методом класса *AhoCorasick*, но работает его экземпляром.

Шаги алгоритма:

- Разбиение шаблона на подстроки без джокеров: подстроки и их смещения добавляются в список *sub_patterns*
- Для подстрок строиться автомат Ахо-Корасик
- Поиск вхождений подстрок в тексте и их добавление в список *matches*
- Массив *C* длинной *len_text - len_pattern + 1* заполняется 0
- *C[i]* — количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i (на основе смещения), например, если для шаблона *a$b b* встречается в тексте на позиции *3*, то *i = 3 - 1 - 2 = 0*, следовательно шаблон можно будет

поместить на позицию 0 в тексте, если и *a* будет принадлежать позиции 0

- Далее каждый *C[i] = k* проверяется, *k* должно быть равно количеству букв шаблона без символов джокера, если *k* совпало, то каждый символ шаблона сравнивается с текстом (исключая джокеры) для полной проверки

Результат: отсортированный список позиций, где шаблон полностью совпадает с текстом.

Оценка сложности

1. Построение бора *O(M)*, где *M* — суммарная длина всех шаблонов.
2. Построение суффиксных и сжатых ссылок *O(M)*, где *M* — суммарная длина всех шаблонов. Используется *BFS*-обход бора, который посещает все узлы. Для каждого узла вычисление суффиксной ссылки занимает амортизированно *O(1)*, так как каждый переход по *fail* уменьшает глубину узла.
3. Поиск вхождений в тексте *O(L+Z)*, где: *L* — длина текста, *Z* — общее количество найденных вхождений.

Итоговая сложность *O(M+L+Z)* по времени.

Память: *O(M)* для хранения бора, суффиксных и сжатых ссылок.

Сложность алгоритма поиска с джокерами

1. Разбиение шаблона на подстроки без джокеров *O(P)*, где *P* — длина исходного шаблона. Каждый символ шаблона обрабатывается один раз.
2. Построение автомата Ахо-Корасик *O(S)*, где *S* — суммарная длина всех подстрок.

3. Поиск подстрок в тексте $O(T+Z)$, где: T — длина текста, Z — общее количество найденных вхождений подстрок. Стандартная сложность алгоритма Ахо-Корасик.
4. Подсчет совпадений в массиве C $O(Z)$. Для каждого вхождения подстроки выполняется $O(1)$ операций.
5. Проверка полного совпадения шаблона $O(V \cdot P)$, где: V — количество позиций, где $C[i]$=число подстрок, P — длина исходного шаблона.

Итоговая сложность $O(P+S+T+Z+V \cdot P)$ по времени.

Память $O(M + T)$ для хранения структуры автомата, подстрок шаблона и массива C.

Тестирование

Результаты тестирования программы представлены в табл. 1.

Табл. 1

Входные данные	Выходные данные
abc 3 abc bc c	1 1 2 2 3 3
axxbaxbabb a*b *	5 8
abcdefghkl 3 fghkla cd bcd	2 3 3 2

Исходный код программы см. в прил. А.

Выводы

В ходе выполнения лабораторной работы была реализована программа для нахождения точных вхождений множества шаблонов в текст, а также вхождений одного шаблона с джокерами. Была выполнена оценка сложности по памяти и времени. Алгоритмы были протестированы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from collections import deque

class AhoCorasick:
    def __init__(self):
        self.trie = [{}]
         self.output = [set()]  # Хранит кортежи (pattern_index,
length)
        self.fail = [0]
        self.dict_suffix = [0]

    def add_pattern(self, pattern, pattern_index):
        """Добавляет точный шаблон (без джокеров)."""
        node = 0
        for char in pattern:
            if char not in self.trie[node]:
                self.trie.append({})
                self.output.append(set())
                self.fail.append(0)
                self.dict_suffix.append(0)
                self.trie[node][char] = len(self.trie) - 1
            node = self.trie[node][char]
        self.output[node].add((pattern_index, len(pattern)))

    def build(self):
        queue = deque()
        for char, node in self.trie[0].items():
            queue.append(node)
            self.fail[node] = 0
            self.dict_suffix[node] = 0

        while queue:
            current = queue.popleft()
            for char, child in self.trie[current].items():
                queue.append(child)
                f = self.fail[current]
                while f and char not in self.trie[f]:
                    f = self.fail[f]
                self.fail[child] = self.trie[f].get(char, 0)
                if self.output[self.fail[child]]:
                    self.dict_suffix[child] = self.fail[child]
                else:
                                        self.dict_suffix[child]  =
self.dict_suffix[self.fail[child]]

    def get_next_node(self, node, char):
        if char in self.trie[node]:
            return self.trie[node][char]
        original_node = node
        while node != 0 and char not in self.trie[node]:
```

```
                node = self.fail[node]
            next_node = self.trie[node].get(char, 0)
            self.trie[original_node][char] = next_node
            return next_node

        def search_all(self, text):
            """Возвращает список (start_position, pattern_index) для
точных шаблонов."""
            node = 0
            occurrences = []
            for i, char in enumerate(text):
                node = self.get_next_node(node, char)
                temp = node
                while temp:
                    if self.output[temp]:
                            for (pattern_index, pattern_length) in
self.output[temp]:
                            start_position = i - pattern_length + 2
# 1-based
                                occurrences.append((start_position,
pattern_index))
                    temp = self.dict_suffix[temp]
            return occurrences

        def get_fail_chain(self, node):
                """Возвращает цепочку суффиксных ссылок (fail) для
узла."""
            chain = []
            current = node
            while current != 0:
                current = self.fail[current]
                chain.append(current)
            return chain

        def get_dict_suffix_chain(self, node):
              """Возвращает цепочку сжатых ссылок (dict_suffix) для
узла."""
            chain = []
            current = node
            while current != 0:
                next_node = self.dict_suffix[current]
                chain.append(next_node)
                current = next_node
            return chain

        def max_fail_chain(self):
             """Находит самую длинную цепочку суффиксных ссылок и её
длину."""
            max_length = 0
            max_chain = []
            for node in range(1, len(self.trie)):
                chain = self.get_fail_chain(node)
                if len(chain) > max_length:
                    max_length = len(chain)
                    max_chain = [node] + chain
            return max_length, max_chain
```

```
        def max_dict_suffix_chain(self):
               """Находит самую длинную цепочку сжатых ссылок и её
длину."""
            max_length = 0
            max_chain = []
            for node in range(len(self.trie)):
                chain = self.get_dict_suffix_chain(node)
                if len(chain) > max_length:
                    max_length = len(chain)
                    max_chain = [node] + chain
            return max_length, max_chain

    def print_trie(ac, node=0, prefix=""):
        if ac.output[node]:
             output_str = ", ".join(f"(№{p}, длина={l})" for p, l in
ac.output[node])
        else:
            output_str = ""
        print(f"{prefix}Узел {node}: {output_str}")
        for char, child in ac.trie[node].items():
            print(f"{prefix}  '{char}' -> Узел {child}")
            print_trie(ac, child, prefix + "    ")

    def print_automaton(ac):
        print("\nАвтомат Ахо-Корасик:")
        total = len(ac.trie)
        for i in range(total):
            print(f"Узел {i}:")
            if ac.trie[i]:
                print("  Потомки:")
                for char, child in ac.trie[i].items():
                    print(f"    '{char}' -> Узел {child}")
            else:
                print("  Потомки: {}")

            print(f"  Суффиксная ссылка: Узел {ac.fail[i]}")
                      print(f"    Хорошая (сжатая) ссылка: Узел
{ac.dict_suffix[i]}")

            if ac.output[i]:
                  outs = ", ".join(f"(№{p}, длина={l})" for p, l in
ac.output[i])
                print(f"  Вывод: {{{outs}}}")
            else:
                print("  Вывод: {}")

            print("-" * 40)

        print()

    def find_wildcard_matches(text, pattern, wild_char):
            """Находит вхождения шаблона с джокерами, используя
AhoCorasick."""
        sub_patterns = []
        n = len(pattern)
```

```
        i = 0
        while i < n:
            if pattern[i] != wild_char:
                start = i
                while i < n and pattern[i] != wild_char:
                    i += 1
                sub = pattern[start:i]
                sub_patterns.append((sub, start))
            else:
                i += 1

        if not sub_patterns:
            pattern_len = len(pattern)
             return [i + 1 for i in range(len(text) - pattern_len +
1)] if pattern_len <= len(text) else []

        ac = AhoCorasick()
        for idx, (sub, offset) in enumerate(sub_patterns):
            ac.add_pattern(sub, idx)
        ac.build()

        len_text = len(text)
        len_pattern = len(pattern)
          C = [0] * (len_text - len_pattern + 1) if len_text >=
len_pattern else []
        matches = ac.search_all(text)

        for (start_pos, pattern_idx) in matches:
            sub, offset = sub_patterns[pattern_idx]
            pattern_start = start_pos - 1 - offset
            if 0 <= pattern_start < len(C):
                C[pattern_start] += 1

        valid_starts = []
        required = len(sub_patterns)
        for i in range(len(C)):
            if C[i] == required:
                valid = True
                for j in range(len_pattern):
                          if pattern[j] != wild_char and (i + j >=
len_text or text[i + j] != pattern[j]):
                         valid = False
                         break
                if valid and (i + len_pattern <= len_text):
                    valid_starts.append(i + 1)

        return sorted(valid_starts)


    def classic_aho():
        text = input()
        n = int(input())
        patterns = []
        for i in range(n):
            patterns.append(input())
```

```
        ac = AhoCorasick()
        for i, pattern in enumerate(patterns, start=1):
            ac.add_pattern(pattern, i)

        print("\nБор:")
        print_trie(ac)
        ac.build()
        print(f"Бор в виде словаря: {ac.trie}")

        print_automaton(ac)
        fail_length, fail_chain = ac.max_fail_chain()
                    dict_suffix_length,    dict_suffix_chain    =
ac.max_dict_suffix_chain()

                print(f"Самая  длинная  цепочка  суффексных  ссылок:
{fail_chain} (длина = {fail_length})")
                  print(f"Самая  длинная  цепочка  сжатых  ссылок:
{dict_suffix_chain} (длина = {dict_suffix_length})")

        occurrences = ac.search_all(text)
        occurrences.sort(key=lambda x: (x[0], x[1]))

        out_lines = [f"{pos} {p}" for pos, p in occurrences]
        print("\n".join(out_lines))

    def searc_with_joker():
        text = input().strip()
        pattern = input().strip()
        wild_char = input().strip()

        matches = find_wildcard_matches(text, pattern, wild_char)


        for pos in matches:
            print(pos)


    if __name__ == "__main__":
        n = int(input())

        if n == 1:
            classic_aho()
        else:
            searc_with_joker()
```