**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≤∣T∣≤100000)*.

Вторая - число *n (1≤n≤3000)*, каждая следующая из n строк содержит шаблон из набора *P = {p1, ...., pn} 1≤∣pi∣≤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 *Р* в текст *Т*.

Например, образец *аb??с?* 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≤∣T∣≤100000)*

Шаблон *(P,1≤∣P∣≤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()*,

которыйвычисляет 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⋅P)*, где: *V* — количество позиций, где *C[i]=*число подстрок, *P* — длина исходного шаблона.

Итоговая сложность *O(P+S+T+Z+V⋅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. A.

## 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()