САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4
по курсу «Алгоритмы и структуры данных»
Тема: Подстроки

Выполнил:
Малыхин Н.С.
К3141

Санкт-Петербург 2025

Содержание

Содержание **2**
Задачи **2**
Задание №1. Наивный поиск подстроки в строке (1 балл) 3
Задание №5. Префикс-функция (1.5 балла) 5
Задание №6. Z-функция (1.5 балла) 7
Вывод **11**

Задачи

Задание №1. Наивный поиск подстроки в строке (1 балл)

Даны строки p и t. Требуется найти все вхождения строки p в строку t в качестве подстроки.

- Формат ввода / входного файла (input.txt). Первая строка входного файла содержит p, вторая – t. Строки состоят из букв латинского алфавита.
- Ограничения на входные данные. $1 \leq |p|, |t| \leq 104$.
- Формат вывода / выходного файла (output.txt). В первой строке выведите число вхождений строки p в строку t. Во второй строке выведите в возрастающем порядке номера символов строки t, с которых начинаются вхождения p. Символы нумеруются с единицы.

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


@measure_time_and_memory
def naive_pattern_search(p, t):
    len_p = len(p)
    occurrences = [] # Позиции вхождения

    for i in range(len(t) - len_p + 1):
        # Сравниваем подстроку текста с искомой подстрокой
        if t[i:i + len_p] == p:
            occurrences.append(i + 1) # Добавляем

    return [len(occurrences), occurrences]


def main():
    # Чтение входных данных из файла
```

```
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    p = input_data[0]
    t = input_data[1]

    # Вычисление результата
    result = naive_pattern_search(p, t)

    # Запись результата в выходной файл
    output_data = str(result[0]) + "\n"
    output_data += " ".join(str(x) for x in result[1])
    write_to_file(output_path, output_data)


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

Объяснение:

1. Инициализация

Определяем длину шаблона p — len_p = len(p). Создаём пустой список occurrences для хранения позиций всех найденных вхождений.

2. Перебор всех возможных позиций

Проходим по всем индексам i строки t, где может начаться вхождение шаблона, то есть от i=0 до i = |t| - |p|.

3. Сравнение подстрок

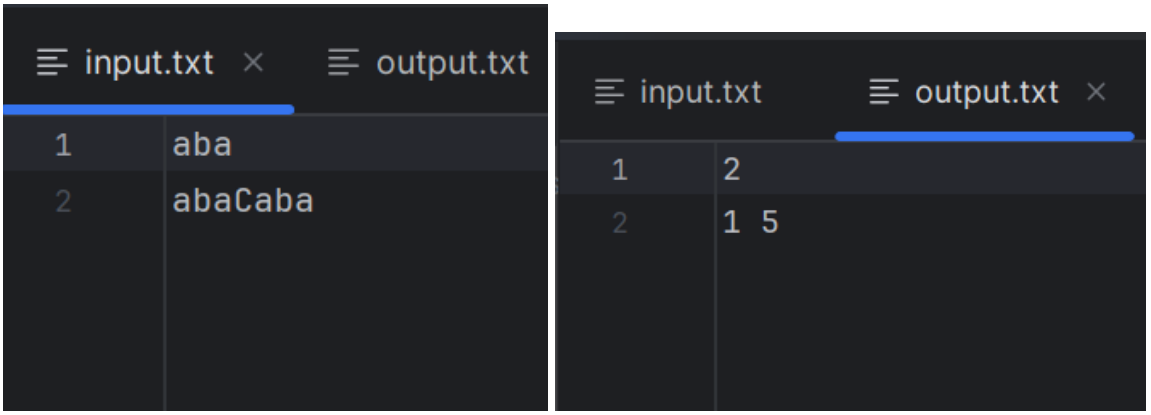
Для каждой позиции i сравниваем подстроку t[i:i+len_p] с шаблоном p. Если они совпадают, значит найдено вхождение.

Сложность:

- Пусть n = |t|, m = |p|.
- В худшем случае алгоритм перебирает все позиции i от 0 до n - m, то есть около n - m + 1 ≈ n позиций.
- Для каждой позиции сравнивает подстроку длины m с шаблоном — это операция O(m).

- Итого, общая временная сложность — O(n · m).

Результат работы кода:



Задание №5. Префикс-функция (1.5 балла)

Постройте префикс-функцию для всех непустых префиксов заданной строки s.

- Формат ввода / входного файла (input.txt). Одна строка входного файла содержит s. Строка состоит из букв латинского алфавита.
- Ограничения на входные данные. $1 \le |s| \le 106$.
- Формат вывода / выходного файла (output.txt). Выведите значения префикс-функции для всех префиксов строки s длиной 1, 2, ..., |s|, в указанном порядке.

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


@measure_time_and_memory
def prefix_func(s):
    n = len(s)
    prefix = [0] * n # Массив префикс-функции
```

```
    for i in range(1, n):
        j = prefix[i - 1] # Начинаем с предыдущего значения префикс-функции

        # Пока не найдем совпадение или не дойдем до начала
        while j > 0 and s[i] != s[j]:
            j = prefix[j - 1] # "Откатываем" j по префикс-функции

        # Если символы совпали, увеличиваем длину совпадающего префикса
        if s[i] == s[j]:
            j += 1

        # Записываем полученное значение
        prefix[i] = j

    return prefix


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    s = input_data

    # Вычисление результата
    result = prefix_func(s)

    # Запись результата в выходной файл
    output_data = " ".join(str(x) for x in result)
    write_to_file(output_path, output_data)


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

Объяснение:

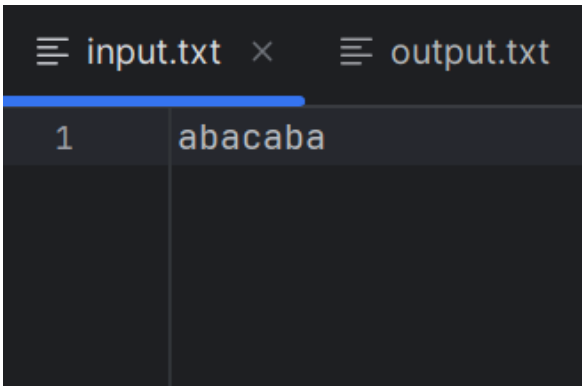
1. Создаётся массив prefix длиной n = |s|, изначально заполненный нулями.
2. Итерация идёт по индексам i от 1 до n-1 (т.к. для i=0 префикс-функция равна 0).

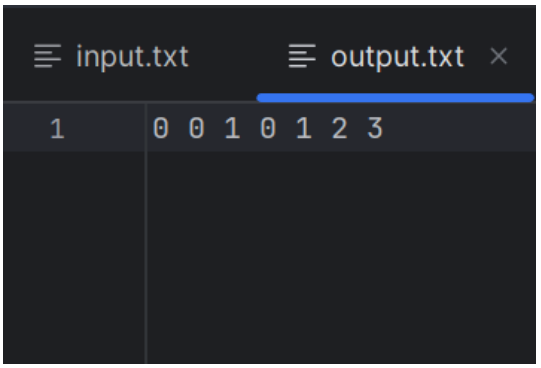
3. Переменная j хранит длину текущего совпадающего префикса-суффикса для предыдущего символа (prefix[i-1]).
4. В цикле while происходит "откат" j по уже вычисленным значениям префикс-функции, если текущий символ s[i] не совпадает с символом s[j].
5. Если символы совпали, j увеличивается на 1 — найден более длинный префикс.
6. Значение prefix[i] = j сохраняется.
7. В итоге возвращается массив префикс-функции для всей строки.

Сложность:

- Алгоритм работает за O(n) времени, где n — длина строки.

Результат работы кода:





Задание №6. Z-функция (1.5 балла)

Постройте Z-функцию для заданной строки s.

- Формат ввода / входного файла (input.txt). Одна строка входного файла содержит s. Строка состоит из букв латинского алфавита.
- Ограничения на входные данные. $2 \leq |s| \leq 106$.
- Формат вывода / выходного файла (output.txt). Выведите значения Z-функции для всех индексов 1, 2, ..., |s| строки s, в указанном порядке.

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


@measure_time_and_memory
def z_func(s):
    n = len(s)
    z = [0] * n # Массив Z-функции
    l, r = 0, 0 # Границы самого правого отрезка совпадения

    for i in range(1, n):
            # Если текущий индекс внутри отрезка [l, r], берем минимум из
оставшейся длины отрезка и z[i-l]
        if i <= r:
            z[i] = min(r - i + 1, z[i - l])

        # Пытаемся расширить текущее совпадение вручную
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1

        # Если нашли более правый отрезок совпадения, то обновляем границы
        if i + z[i] - 1 > r:
            l, r = i, i + z[i] - 1

    return z[1:]


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    s = input_data

    # Вычисление результата
    result = z_func(s)

    # Запись результата в выходной файл
```

```
    output_data = " ".join(str(x) for x in result)
    write_to_file(output_path, output_data)


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

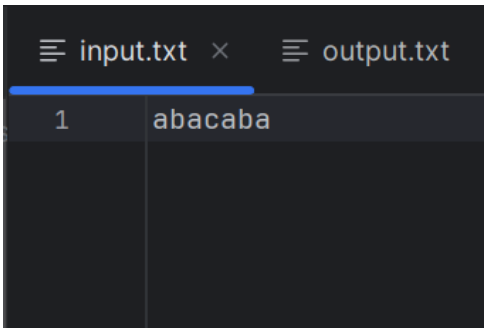
Объяснение:

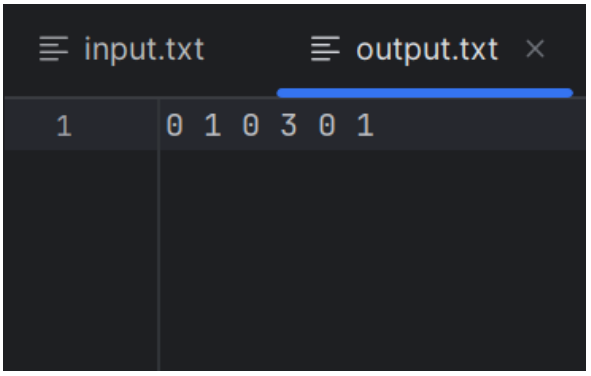
1. Инициализируем массив z длиной n = |s| нулями.
2. Переменные l и r хранят границы текущего наиболее правого отрезка совпадения — интервала [l, r], где s[l..r] совпадает с префиксом строки.
3. Проходим по индексам i от 1 до n-1:
    - Если i ≤ r, то z[i] минимум из:
        - оставшейся длины отрезка справа: r - i + 1
        - значения z[i - l] (аналогичного сдвинутого индекса)
    - Затем пытаемся расширить совпадение вручную, сравнивая символы s[z[i]] и s[i + z[i]].
    - Если расширенное совпадение выходит за текущий правый отрезок r, обновляем l и r.
4. Возвращаем массив z, начиная со второго элемента z[1:], так как z[0] по определению равен нулю (или длине строки, но обычно не используется).

Сложность:

- Алгоритм работает за O(n) времени, где n — длина строки.

Результат работы кода:


input.txt
output.txt
1
abacaba


input.txt
output.txt
1
0 1 0 3 0 1

Вывод

Эти алгоритмы демонстрируют важность предварительной обработки данных для оптимизации поиска и анализа строк, что критично при работе с большими объемами текста.