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

Отчет по лабораторной работе №1
По курсу «Алгоритмы и структуры данных»
Тема: Жадные алгоритмы. Динамическое программирование
Вариант 13

Выполнила:
Мкртчян А.А.
К3242

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Задача N3

Описание задания

Задача: Реализовать алгоритм поиска подстроки в строке с использованием алгоритма Рабина-Карпа. Требуется найти все вхождения строки-шаблона `P` в строку текста `T` и вывести их индексы (индексация с 1).

Описание решения и исходный код

Алгоритм Рабина-Карпа использует метод хеширования для быстрого поиска подстрок в тексте. Он вычисляет хэш строки-шаблона и хэш первых `m` символов текста, где `m` — длина шаблона. Затем хэш текста обновляется для каждого следующего окна в тексте, что позволяет быстро находить вхождения шаблона.

Исходный код:

```
def rabin_karp(P, T):
    p_len = len(P)
    t_len = len(T)
    base = 256
    mod = 10**9 + 7

    def hash_func(s, length):
        h = 0
        for i in range(length):
            h = (h * base + ord(s[i])) % mod
        return h

    # Вычисляем хэш паттерна P и начального окна T
    p_hash = hash_func(P, p_len)
    t_hash = hash_func(T, p_len)

    # Препроцессинг для быстрого пересчета хэша при сдвиге окна
    h = pow(base, p_len - 1, mod)

    result = []
    for i in range(t_len - p_len + 1):
        if p_hash == t_hash:  # Возможное совпадение
            if T[i:i + p_len] == P:
                result.append(i + 1)  # Добавляем индекс (начиная с 1)
        if i < t_len - p_len:
            t_hash = (t_hash - ord(T[i]) * h) % mod
            t_hash = (t_hash * base + ord(T[i + p_len])) % mod
```

```
            t_hash = (t_hash + mod) % mod  # Избегаем отрицательных хэшей

    return result

# Чтение данных из файлов
with open('input.txt', 'r') as file:
    P = file.readline().strip()
    T = file.readline().strip()

# Вызов функции
indices = rabin_karp(P, T)

# Запись результатов в файл
with open('output.txt', 'w') as file:
    file.write(f"{len(indices)}\n")
    file.write(" ".join(map(str, indices)) + "\n")
```

Входные данные:

aba

abacaba

Выходные данные:

2

1 5

Объяснение: Подстрока `aba` встречается два раза в строке `abacaba` — начиная с индексов 1 и 5.

Выводы

В ходе выполнения работы было реализовано решение задачи поиска подстроки с использованием алгоритма Рабина-Карпа. Проведенные тесты показали корректную работу программы. Алгоритм успешно находит все вхождения строки-шаблона в текст, используя хеширование для ускорения процесса поиска.

- Время работы алгоритма линейно зависит от длины текста и шаблона.

- Алгоритм Рабина-Карпа полезен для задач поиска подстрок с улучшенной производительностью по сравнению с наивным поиском.

Задача N7

Описание задания

Задача: Найти наибольшую общую подстроку между двумя строками. Входные данные содержат несколько строк, каждая из которых состоит из двух строк, разделенных пробелом. Для каждой пары строк необходимо вывести индексы начала наибольшей общей подстроки в обеих строках и длину этой подстроки.

Описание решения и исходный код

Решение задачи состоит из нескольких этапов:

1. Для каждой строки вычисляются хеши всех подстрок длины `k` (используется метод сдвига окна).
2. Применяется бинарный поиск для поиска наибольшей общей подстроки, начиная с наибольшей возможной длины и постепенно уменьшая длину, если общая подстрока не найдена.
3. Хеши подстрок обеих строк сравниваются, и если они совпадают, проверяется совпадение самих подстрок.

Исходный код:

```
def get_hashes(s, k, base=31, mod=10 ** 9 + 7):
    """Вычисление хешей всех подстрок длиной k для строки s."""
    n = len(s)
    hashes = {}
    hash_value = 0
    base_k = pow(base, k, mod)  # base^k % mod

    # Вычисляем хеш для первой подстроки длиной k
    for i in range(k):
        hash_value = (hash_value * base + ord(s[i])) % mod

    hashes[hash_value] = [0]  # добавляем индекс начала подстроки

    # Прокручиваем окно по строке и вычисляем хеши для подстрок
    for i in range(1, n - k + 1):
        hash_value = (hash_value * base + ord(s[i + k - 1]) - ord(s[i -
1]) * base_k) % mod
        if hash_value not in hashes:
            hashes[hash_value] = []
        hashes[hash_value].append(i)

    return hashes


def find_common_substring(s, t, k):
    """Проверка, есть ли общая подстрока длины k между строками s и t."""
    hashes_s = get_hashes(s, k)
    hashes_t = get_hashes(t, k)

    for hash_value in hashes_s:
        if hash_value in hashes_t:
            # Проверяем, совпадают ли подстроки
            for start s in hashes s[hash value]:
                for start_t in hashes_t[hash_value]:
                    if s[start_s:start_s + k] == t[start_t:start_t + k]:
                        return start_s, start_t
    return None


def longest_common_substring(s, t):
    """Бинарный поиск длины наибольшей общей подстроки."""
    low, high = 0, min(len(s), len(t))
    best_length = 0
    best_pos_s = 0
    best_pos_t = 0

    while low <= high:
        mid = (low + high) // 2
        common_substr = find_common_substring(s, t, mid)

        if common_substr:
            best_length = mid
            best_pos_s, best_pos_t = common_substr
            low = mid + 1
        else:
            high = mid - 1
```

```
    return best_pos_s, best_pos_t, best_length


def process_input_output(input_file, output_file):
    # Чтение данных из input.txt
    with open(input_file, 'r') as f:
        lines = f.read().splitlines()

    results = []

    # Проход по каждой строке
    for line in lines:
        s, t = line.split()  # разделение по пробелу
        pos_s, pos_t, length = longest_common_substring(s, t)
        results.append(f"{pos_s} {pos_t} {length}")

    # Запись результатов в output.txt
    with open(output_file, 'w') as f:
        f.write("\n".join(results))

# Пример вызова функции
process_input_output('input.txt', 'output.txt')
```

Описание тестов

Тест 1:

Ввод:

```
Копировать код
cool toolbox
```

Ожидаемый вывод:

```
1 1 3
```

Объяснение: Наибольшая общая подстрока — "ool", начинается с позиции 1 в обеих строках.

Тест 2:

Ввод:

```
aaa bb
```

Ожидаемый вывод:

```
0 0 0
```

Объяснение: Нет общей подстроки между строками.

Тест 3:

Ввод:

```
aabaa babbaab
```

Ожидаемый вывод:

```
0 4 3
```

Объяснение: Наибольшая общая подстрока — "aab", начинается с позиции 0 в первой строке и с позиции 4 во второй строке.

Выводы

В ходе выполнения задачи был реализован алгоритм нахождения наибольшей общей подстроки с использованием хеширования и бинарного поиска. Программа корректно находит общие подстроки и вычисляет их начальные индексы и длину для каждой пары строк.

- Алгоритм эффективно работает для длинных строк за счет использования хеширования.
- Результаты тестов подтвердили правильность работы алгоритма.

Задание N8

Описание задания

Задача: Дано две строки `t` (текст) и `p` (шаблон), а также число `k`. Необходимо найти все позиции, где шаблон `p` совпадает с подстроками строки `t` с не более чем `k` несовпадениями.

Описание решения и исходный код

Решение задачи включает следующие этапы:

1. **Вычисление хеш-значений**: Хеш-значения префиксов строки не используются в этом варианте решения, но могут быть добавлены для более эффективного поиска, если задача расширится.
2. **Сравнение подстрок**: Для каждой возможной подстроки `t` длины `len(p)` подсчитывается количество несовпадений с шаблоном `p`. Если количество несовпадений не превышает `k`, то индекс этой подстроки добавляется в результаты.

Исходный код:

```python
def hash_prefixes(s):
    """Вычисляет хеш-значения префиксов строки s."""
    p = 31  # простое число для хеширования
    m = 10 ** 9 + 9  # модуль
    n = len(s)
    hash_values = [0] * (n + 1)
    p_pow = [1] * (n + 1)

    for i in range(n):
        hash_values[i + 1] = (hash_values[i] * p + ord(s[i])) % m
        p_pow[i + 1] = (p_pow[i] * p) % m

    return hash_values, p_pow


def count_mismatches(t, p, start, k):
    """Считает количество несовпадений между p и подстрокой t[start:start
```

```
+ len(p)]."""
    mismatches = 0
    for i in range(len(p)):
        if t[start + i] != p[i]:
            mismatches += 1
            if mismatches > k:
                return mismatches
    return mismatches


def process_input_output(input_file, output_file):
    with open(input_file, 'r') as f:
        lines = f.read().splitlines()

    results = []

    for line in lines:
        k, t, p = line.split()
        k = int(k)
        t_length = len(t)
        p_length = len(p)

        positions = []

        # Перебираем все возможные позиции в t
        for i in range(t_length - p_length + 1):
            mismatches = count_mismatches(t, p, i, k)  # Передаем k как
аргумент
            if mismatches <= k:
                positions.append(i)

        results.append(f"{len(positions)} {' '.join(map(str,
positions))}")

    # Записываем результаты в output.txt
    with open(output_file, 'w') as f:
        f.write("\n".join(results))


# Вызов функции для обработки
process_input_output('input.txt', 'output.txt')
```

Описание тестов

Тест 1:

Ввод:

```
0 ababab baaa
```

Ожидаемый вывод:

```
0
```

Объяснение: Совпадений без несовпадений не найдено.

Тест 2:

Ввод:

```
1 ababab baaa
```

Ожидаемый вывод:

```
1 1
```

Объяснение: Одна подстрока (начиная с позиции 1) имеет одно несовпадение.

Тест 3:

Ввод:

```
1 xabcabc ccc
```

Ожидаемый вывод:

```
0
```

Объяснение: Не найдено подстрок, соответствующих шаблону с одним несовпадением.

Тест 4:

Ввод:

```
2 xabcabc ccc
```

Ожидаемый вывод:

```
4 1 2 3 4
```

Объяснение: Найдены 4 подстроки, каждая из которых имеет не более 2 несовпадений.

Тест 5:

Ввод:

```
3 aaa xxx
```

Ожидаемый вывод:

```
1 0
```

Объяснение: Подстрока (начиная с позиции 0) имеет 3 несовпадения.

Выводы

В ходе выполнения задачи был разработан алгоритм для поиска подстрок с несовпадениями, который эффективно анализирует строки на наличие совпадений. Результаты тестов подтвердили корректность работы алгоритма.

- Программа позволяет находить позиции, удовлетворяющие условию несовпадений.
- Подход с перебором обеспечивает простоту реализации,
- однако в случае больших строк может быть оптимизирован за счет использования более эффективных методов хеширования.