

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

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

Тема: Подстроки

Вариант 4

Выполнила:

Буй Тхук Хуен - К3139

Проверила:

Петросян А.М.

Санкт-Петербург

2025 г.

## Содержание отчета

I. Задачи по варианту.....	3
Задача №1. Наивный поиск подстроки в строке.....	3
Задача №6. Z-функция.....	5
Задача №7. Наибольшая общая подстрока.....	8
II. Вывод.....	12

## I. Задачи по варианту

### Задача №1. Наивный поиск подстроки в строке

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.**  $1 \leq |p|, |t| \leq 10^4$ .
- **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.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

#### • Листинг кода

```
from Lab4.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def find_occurrences(text, pattern):
    occurrences = []
    len_p = len(pattern)
    for i in range(len(text) - len_p + 1):
        if text[i:i+len_p] == pattern:
            occurrences.append(i + 1)
    return occurrences

def main():
    p, t = read(PATH_INPUT, 1)
    results = find_occurrences(t, p)
    write(PATH_OUTPUT, results, 1)

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

#### • Текстовое объяснение решения

- + Функция `find_occurrences(text, pattern)`: ищет все вхождения подстроки `pattern` в строке `text` и возвращает их позиции (начиная с 1, а не с 0)
  - Создаётся пустой список `occurrences`, в который будут записываться начальные позиции найденных вхождений.
  - Запоминаем длину искомой подстроки `pattern`

- Перебираем все возможные начальные позиции  $i$  в строке `text`, где может начинаться `pattern`.
- Для каждой позиции  $i$  проверяем, совпадает ли подстрока `text[i..i+len_p-1]` с `pattern`.
- Если совпадение найдено, добавляем позицию  $i + 1$  в список (так как нумерация символов начинается с 1).
- Возвращаем список всех найденных позиций.

- Результат работы код на примерах из текста задачи:

Task1\...\in			Task1\...\out		
1	aba	✓	1	2	✓
2	abaCaba		2	1 5	
			3		

- Test:

```

Test Case: test1
Execution Time = 0.00001030 s, Memory Usage = 0.1689 KB

Test Case: test2
Execution Time = 0.00000950 s, Memory Usage = 0.1689 KB

Test Case: test3
Execution Time = 0.00001030 s, Memory Usage = 0.2314 KB

```

Тест	Время выполнения (s)	Затраты памяти (KB)
a banana	0.00001030	0.1689
aba abacabaababa	0.00000950	0.1689
aaa aaaaaaaaaa	0.00001030	0.2314

- Вывод

- + Алгоритм КМР: Стабильная производительность, не зависящая от структуры t- или p-цепи.
- +  $O(|t| + |p|)$  во всех случаях, включая наихудший случай.
- + Подходит для больших данных
- + Потребляет дополнительную память  $O(|p|)$ : незначительно для коротких p, но может иметь эффект, если  $|p| \approx 10^6$ .

## Задача №6. Z-функция

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.**  $2 \leq |s| \leq 10^6$ .
- **Формат вывода / выходного файла (output.txt).** Выведите значения Z-функции для всех индексов  $1, 2, \dots, |s|$  строки  $s$ , в указанном порядке.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

input.txt	output.txt	input.txt	output.txt
aaaAAA	2 1 0 0 0	abacaba	0 1 0 3 0 1

### • Листинг кода

```
from Lab4.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def z_function(s):
    n = len(s)
    z = [0] * n
    l, r, k = 0, 0, 0
    for i in range(1, n):
        if i <= r:
            z[i] = min(r - i + 1, z[i - l])
            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
                z[i] += 1
            if i + z[i] - 1 > r:
                l, r = i, i + z[i] - 1
    return z

def main():
    s = read(PATH_INPUT, 6)
    results = z_function(s)
    write(PATH_OUTPUT, results, 6)

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

- Текстовое объяснение решения:

Эта функция вычисляет Z-массив для строки  $s$ . Каждый элемент  $z[i]$  — это длина наибольшего подстрока, начинающегося с позиции  $i$ , который совпадает с префиксом строки  $s$ .

- +  $l, r$  — индексы, задающие текущий отрезок  $[l, r]$ , где  $s[l..r]$  совпадает с префиксом  $s[0..r-l]$
- + Начинаем цикл с  $i=l, i=r$  (по заданию Z-функция считается с 1 индекса).
- + Внутри цикла:
  - Если  $i$  находится в пределах текущего окна  $[l, r]$ , мы пытаемся использовать уже вычисленные значения Z-функции.  $z[i]$  инициализируется значением  $\min(r-i+1, z[i-l])$ , чтобы не выйти за пределы окна.
  - Затем мы продолжаем увеличивать значение  $z[i]$ , если символы  $s[z[i]]$  и  $s[i+z[i]]$  совпадают. Это позволяет нам вычислить максимальную длину префикса, который совпадает с частью строки.
  - Если текущее окно выходит за пределы предыдущего, мы обновляем границы  $l$  и  $r$ .
- + В конце мы возвращаем массив  $z$ , который содержит значения Z-функции для всех индексов строки.

- Результат работы код на примерах из текста задачи:

input.txt	output.txt
1 aaaAAA	1 0 1 0 0 0
2	2

  

input.txt	output.txt
1 abacaba	1 0 1 0 3 0 1
2	2

- Test

```
Test Case: test1
Execution Time = 0.00003340 s, Memory Usage = 0.1924 KB

Test Case: test2
Execution Time = 0.00001010 s, Memory Usage = 0.2314 KB

Test Case: test3
Execution Time = 0.00000760 s, Memory Usage = 0.2314 KB
```

Тест	Время выполнения (s)	Затраты памяти (KB)
aaaaa	0.00003340	0.1924
abacabab	0.00001010	0.2314
abcdefgh	0.00000760	0.2314

- Вывод

- + Временная сложность  $O(n)$ : Высокая производительность даже при больших объемах данных ( $n \leq 10^6$ )
- + Нет необходимости в сложных структурах данных. Экономит память ( $O(n)$ )
- + Не всегда самый быстрый на практике: Для коротких выборок алгоритм Бойера-Мура может быть более эффективным благодаря правилу «плохого символа». Для множественных выборок метод Рабина-Карпа (с использованием хеширования) иногда оказывается более оптимальным.

## Задача №7. Наибольшая общая подстрока

В задаче на наибольшую общую подстроку даются две строки  $s$  и  $t$ , и цель состоит в том, чтобы найти строку  $w$  максимальной длины, которая является подстрокой как  $s$ , так и  $t$ . Это естественная мера сходства между двумя строками. Задача имеет применения для сравнения и сжатия текстов, а также в биоинформатике. Эту проблему можно рассматривать как частный случай проблемы расстояния редактирования (Левенштейна), где разрешены только вставки и удаления. Следовательно, ее можно решить за время  $O(|s||t|)$  с помощью динамического программирования. Есть также весьма нетривиальные структуры данных для решения этой задачи за линейное время  $O(|s| + |t|)$ . В этой задаче ваша цель – использовать хеширование для решения почти за линейное время.

- **Формат ввода / входного файла (input.txt).** Каждая строка входных данных содержит две строки  $s$  и  $t$ , состоящие из строчных латинских букв.
- **Ограничения на входные данные.** Суммарная длина всех  $s$ , а также суммарная длина всех  $t$  не превышает 100 000.
- **Формат вывода / выходного файла (output.txt).** Для каждой пары строк  $s_i$  и  $t_i$  найдите ее самую длинную общую подстроку и уточните ее параметры, выведя три целых числа: ее начальную позицию в  $s$ , ее начальную позицию в  $t$  (обе считаются с 0) и ее длину. Формально выведите целые числа  $0 \leq i < |s|$ ,  $0 \leq j < |t|$  и  $l \geq 0$  такие, что и  $l$  максимально. (Как обычно, если таких троек с максимальным  $l$  много, выведите любую из них.)
- Ограничение по времени. 15 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input	output
cool toolbox	1 1 3
aaa bb	0 1 0
aabaa babbaab	0 4 3

### • Листинг кода

```
from Lab4.utils import read, write

PATH_INPUT = '../txtf/input.txt'
PATH_OUTPUT = '../txtf/output.txt'

def poly_hash(s, base=257, mod=10 ** 9 + 7):
    """Вычисление полиномиального хеша строки."""
    hash_value = 0
    for i, c in enumerate(s):
        hash_value = (hash_value * base + ord(c)) % mod
    return hash_value

def get_hashes(s, length, base=257, mod=10 ** 9 + 7):
    """Вычисление хешей всех подстрок строки s длины length."""
    n = len(s)
    hashes = {}
    base_power = 1 # base^length % mod
    for i in range(length):
        base_power = (base_power * base) % mod

    # Вычисляем хеш для первой подстроки
    current_hash = 0
    for i in range(length):
        current_hash = (current_hash * base + ord(s[i])) % mod
```



```

hashes[current_hash] = [0] # хеш и его индекс начала

# Прокачиваем хеши для других подстрок
for i in range(1, n - length + 1):
    current_hash = (current_hash * base - ord(s[i - 1]) * base_power +
ord(s[i + length - 1])) % mod
    if current_hash not in hashes:
        hashes[current_hash] = []
    hashes[current_hash].append(i)

return hashes

def common_substring(s, t):
    # Двоичный поиск по длине
    left, right = 0, min(len(s), len(t))
    best_length = 0
    best_indices = []

    while left <= right:
        mid = (left + right) // 2
        # Хеши подстрок длины mid для обеих строк
        s_hashes = get_hashes(s, mid)
        t_hashes = get_hashes(t, mid)

        found = False

        # Ищем общие хеши
        for t_hash, t_indices in t_hashes.items():
            if t_hash in s_hashes:
                # Если хеши совпадают, проверяем сами подстроки
                for i in s_hashes[t_hash]:
                    for j in t_indices:
                        if s[i:i + mid] == t[j:j + mid]:
                            if mid > best_length:
                                best_length = mid
                                best_indices = [(i, j)]
                            elif mid == best_length:
                                best_indices.append((i, j))
                            found = True
                            break
                    if found:
                        break
                if found:
                    break

        if found:
            left = mid + 1 # Пытаемся найти подстроку большей длины
        else:
            right = mid - 1 # Ищем меньшую длину

    return best_indices, best_length

def main():

```

```

lines = read(PATH_INPUT, 7)
results = []
for line in lines:
    s, t = line.split()
    best_indices, best_length = common_substring(s, t)

    if best_length == 0:
        results.append(f"0 1 0")
    else:
        for i, j in best_indices:
            results.append(f"{i} {j} {best_length}")

write(PATH_OUTPUT, results, 7)

if __name__ == "__main__":
    main()

```

- Текстовое объяснение решения

- + Функция `poly_hash`: вычисляет полиномиальный хеш для строки.
  - Каждый символ строки умножается на некоторую степень основания (в данном случае, 257), и результат сохраняется с использованием остаточного деления по модулю ( $10^9+7$ ). Это помогает избежать переполнения чисел и позволяет работать с большими строками.
- + Функция `get_hashes`: вычисляет хеши всех подстрок длины `length` из строки `s` с использованием техники скользящего окна:
  - Мы начинаем с вычисления хеша первой подстроки (первые `length` символов).
  - Для каждого следующего сдвига подстроки, хеш обновляется по формуле, используя старый хеш, вычитая старый символ и добавляя новый. Это позволяет пересчитывать хеш за время  $O(1)$ , что значительно ускоряет процесс.
- + Функция `common_substring`: реализует бинарный поиск по длине наибольшей общей подстроки.
  - Бинарный поиск нужен для того, чтобы найти максимальную длину общей подстроки между двумя строками. Мы начинаем с поиска средней длины и проверяем, можно ли найти общую подстроку такой длины. Затем, в зависимости от результата, изменяем границы поиска.

- Для каждой длины подстроки, которую мы проверяем, вычисляются хеши для всех подстрок длины `mid` в строках `s` и `t`. Если хеши совпадают, то мы дополнительно сравниваем сами подстроки, чтобы избежать коллизий хешей.
- Если найдена общая подстрока длины `mid`, то мы обновляем список результатов. Если таких подстрок несколько с одинаковой длиной, все они добавляются в список.

- Результат работы код на примерах из текста задачи

input.txt			output.txt		
1	cool toolbox	✓	1	1 1 3	✓
2	aaa bb		2	0 1 0	
3	aabaa babbaab		3	2 3 3	
4			4		

- Test

```

Test Case: test1
Execution Time = 0.00017450 s, Memory Usage = 1.2578 KB

Test Case: test2
Execution Time = 0.00005120 s, Memory Usage = 0.7734 KB

Test Case: test3
Execution Time = 0.00049340 s, Memory Usage = 6.0234 KB

```

Тест	Время выполнения (s)	Затраты памяти (KB)
abcdefg abcxyz	0.00017450	1.2578
abcd xyz	0.00005120	0.7734
aabbaahkafhkhkndn djerwwfgsgvss jerww	0.00049340	6.0234

- Вывод

- + Алгоритм имеет временную сложность  $O(n \log n)$ , где  $n$  — длина строки (как `s`, так и `t`).

- + Двоичный поиск используется для определения максимальной общей длины подстроки от 0 до  $\min(\text{len}(s), \text{len}(t))$ .
- + Функция хеширования используется для вычисления хеша для всех подстрок фиксированной длины в каждом поиске. Это сокращает время по сравнению с прямым сравнением подстрок, благодаря чему алгоритм работает намного быстрее.

## II. Вывод

Сравнение алгоритмов Кнута-Морриса-Практа, Z-функция и Рабина-Карпа

Критерии	Временная сложность	Преимущества	Недостатки
КМР	$O(n + m)$ , где $m$ — длина паттерна, $n$ — длина строки, в которой ищем паттерн.	Быстрое выполнение, оптимален для одиночного поиска.	Может быть менее эффективен для множества поисков.
Z-функция	$O(n + m)$ , где $m$ — длина паттерна, а $n$ — длина строки	Простота реализации, подходит для многократного поиска.	Может не так быстр для паттернов с общими символами.
Рабина-Карпа	$O(n + m)$ где $m$ — длина паттерна, а $n$ — длина строки в среднем но может быть и хуже из-за коллизий	Эффективен при многократных поисках, хороший для длинных текстов.	Могут возникать коллизии, что делает его медленнее.