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

# Отчет по лабораторной работе №4 (семестр 2)
# по курсу «Алгоритмы и структуры данных»
# Тема: Подстроки
# Вариант 10

Выполнила:
Коновалова Кира Романовна
К3139

Преподаватель:
Петросян Анна Мнацакановна

Санкт-Петербург
2025г.

# Содержание отчета

## Задачи по варианту. Задачи по выбору

### Задача №2. Карта

Текст задачи:

В далеком 1744 году во время долгого плавания в руки капитана Александра Смоллетта попала древняя карта с указанием местонахождения сокровищ. Однако расшифровать ее содержание было не так уж и просто.

Команда Александра Смоллетта догадалась, что сокровища находятся на $x$ шагов восточнее красного креста, однако определить значение числа она не смогла. По возвращению на материк Александр Смоллетт решил обратиться за помощью в расшифровке послания к знакомому мудрецу. Мудрец поведал, что данное послание таит за собой некоторое число. Для вычисления этого числа необходимо было удалить все пробелы между словами, а потом посчитать количество способов вычеркнуть все буквы кроме трех так, чтобы полученное слово из трех букв одинаково читалось слева направо и справа налево.

Александр Смоллетт догадывался, что число, зашифрованное в послании, и есть число $x$. Однако, вычислить это число у него не получилось.

После смерти капитана карта была безнадежно утеряна до тех пор, пока не оказалась в ваших руках. Вы уже знаете все секреты, осталось только вычислить число $x$.

- **Формат ввода / входного файла (input.txt).** В единственной строке входного файла дано послание, написанное на карте.
- **Ограничения на входные данные.** Длина послания не превышает $3 \cdot 10^5$. Гарантируется, что послание может содержать только строчные буквы английского алфавита и пробелы. Также гарантируется, что послание не пусто. Послание не может начинаться с пробела или заканчиваться им.
- **Формат вывода / выходного файла (output.txt).** Выведите одно число $x$ – число способов вычеркнуть из послания все буквы кроме трех так, чтобы оставшееся слово одинаково читалось слева направо и справа налево.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

| input.txt | output.txt | input.txt | output.txt |
|---|---|---|---|
| treasure | 8 | you will never find the treasure | 146 |

Листинг кода:

```python
from itertools import combinations


def count_palindromic_triplets(message: str) -> int:
    # Удаляем пробелы из строки
    filtered_message = message.replace(" ", "")
    n = len(filtered_message)

    # Подсчитываем все возможные палиндромные тройки
    count = 0
    for i, j, k in combinations(range(n), 3):
        if filtered_message[i] == filtered_message[k]:
            count += 1

    return count


def read_input(filename: str) -> str:
    with open(filename, "r", encoding="utf-8") as file:
        return file.readline().strip()
```

```
def write_output(filename: str, result: int):
    with open(filename, "w", encoding="utf-8") as file:
        file.write(str(result) + "\n")


if __name__ == "__main__":
    input_text = read_input("../txtf/input.txt")
    result = count_palindromic_triplets(input_text)
    write_output("../txtf/output.txt", result)
```

Текстовое объяснение задачи:

1. В начале алгоритм удаляет все пробелы из входной строки, чтобы анализировать только буквы. Это делается с помощью метода replace(" ", "")
2. Поиск всех возможных палиндромных троек
    a. Используется функция combinations(range(n), 3), которая перебирает все возможные сочетания трех индексов из строки.
    b. Для каждой такой тройки индексов (i, j, k) проверяется, является ли полученная трехбуквенная подстрока палиндромом (то есть совпадают ли первая и последняя буквы: filtered_message[i] == filtered_message[k])
    c. Если условие выполняется, счетчик увеличивается на 1

Тесты:

```
import unittest
from lab4.task2.src.main import count_palindromic_triplets


class TestTreasureMap(unittest.TestCase):
    def test_example_cases(self):
        self.assertEqual(count_palindromic_triplets("treasure"), 8)
        self.assertEqual(count_palindromic_triplets("you will never find the
treasure"), 146)

    def test_minimum_length(self):
        self.assertEqual(count_palindromic_triplets("abc"), 0)  # Только одна
возможная тройка

    def test_mixed_characters(self):
        self.assertEqual(count_palindromic_triplets("racecar"), 9)  # Разные
палиндромные комбинации
        self.assertEqual(count_palindromic_triplets("abba"), 2)  #
Палиндромные сочетания

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

Вывод по задаче:

Данный алгоритм находит все возможные трехбуквенные комбинации в строке, игнорируя пробелы, и подсчитывает те из них, которые являются палиндромами. Он последовательно перебирает все тройки символов, проверяя совпадение первой и последней буквы. Временная сложность алгоритма составляет O(n^3), что делает его неэффективным для больших входных данных. Улучшить производительность можно, снизив сложность до O(n^2)с помощью префиксных массивов частот символов или динамического программирования

**Задача №4. Равенство подстрок**

Текст задачи:

В этой задаче вы будете использовать хеширование для разработки алгоритма, способного предварительно обработать заданную строку $s$, чтобы ответить эффективно на любой запрос типа «равны ли эти две подстроки $s$?» Это, в свою очередь, является основной частью во многих алгоритмах обработки строк.

- **Формат ввода / входного файла (input.txt).** Первая строка содержит строку $s$, состоящую из строчных латинских букв. Вторая строка содержит количество запросов $q$. Каждая из следующих $q$ строк задает запрос тремя целыми числами $a$, $b$ и $l$.
- **Ограничения на входные данные.** $1 \le |s| \le 500000, 1 \le q \le 100000, 0 \le a, b \le |s| - l$ (следовательно, индексы $a$ и $b$ начинаются с 0).
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса выведите «Yes», если подстроки $s_a s_{a+1} \ldots s_{a+l-1} = s_b s_{b+1} \ldots s_{b+l-1}$ равны, и «No» – если не равны.
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Пример:

| input | output |
|---|---|
| trololo<br>4<br>0 0 7<br>2 4 3<br>3 5 1<br>1 3 2 | Yes<br>Yes<br>Yes<br>No |

Листинг кода:

```
import random

def compute_hashes(s, x, m):
    n = len(s)
    h = [0] * (n + 1)
    for i in range(1, n + 1):
        h[i] = (x * h[i - 1] + ord(s[i - 1])) % m
    return h


def get_hash(h, a, l, x_pow, m):
    return (h[a + l] - x_pow[l] * h[a] % m + m) % m
```

```python
def preprocess_powers(x, max_len, m):
    x_pow = [1] * (max_len + 1)
    for i in range(1, max_len + 1):
        x_pow[i] = (x_pow[i - 1] * x) % m
    return x_pow


if __name__ == "__main__":
    # Чтение входных данных
    with open("../txtf/input.txt", "r") as f:
        s = f.readline().strip()
        q = int(f.readline().strip())
        queries = [tuple(map(int, f.readline().split())) for _ in range(q)]

    n = len(s)
    m1, m2 = 10 ** 9 + 7, 10 ** 9 + 9
    x = random.randint(1, 10 ** 9)

    h1 = compute_hashes(s, x, m1)
    h2 = compute_hashes(s, x, m2)

    x_pow1 = preprocess_powers(x, n, m1)
    x_pow2 = preprocess_powers(x, n, m2)

    results = []
    for a, b, l in queries:
        hash_a1 = get_hash(h1, a, l, x_pow1, m1)
        hash_b1 = get_hash(h1, b, l, x_pow1, m1)
        hash_a2 = get_hash(h2, a, l, x_pow2, m2)
        hash_b2 = get_hash(h2, b, l, x_pow2, m2)

        if hash_a1 == hash_b1 and hash_a2 == hash_b2:
            results.append("Yes")
        else:
            results.append("No")

    with open("../txtf/output.txt", "w") as f:
        f.write("\n".join(results) + "\n")
```

Текстовое объяснение задачи:

Задача заключается в проверке равенства подстрок строки с помощью хеширования. Используется полиномиальная хеш-функция, которая позволяет быстро сравнивать подстроки без их прямого сравнения. Хеш каждой подстроки вычисляется заранее с использованием префиксов строки. Для этого выбираются два модуля (m1 и m2), чтобы минимизировать вероятность коллизий, и случайное значение для x, которое используется при вычислениях.

Для каждого запроса, состоящего из позиций начала подстрок и их длины, хеши двух подстрок вычисляются за O(1) времени с помощью предварительных вычислений. Сравниваются хеши подстрок, и если они совпадают по обоим модулям, то подстроки равны.

Тесты:

```
import unittest
from lab4.task4.src.main import compute_hashes, get_hash, preprocess_powers


class TestSubstringEquality(unittest.TestCase):
    def setUp(self):
        self.s = "trololo"
        self.x = 31
        self.m1, self.m2 = 10 ** 9 + 7, 10 ** 9 + 9
        self.h1 = compute_hashes(self.s, self.x, self.m1)
        self.h2 = compute_hashes(self.s, self.x, self.m2)
        self.x_pow1 = preprocess_powers(self.x, len(self.s), self.m1)
        self.x_pow2 = preprocess_powers(self.x, len(self.s), self.m2)

    def test_equal_substrings(self):
        self.assertEqual(get_hash(self.h1, 0, 7, self.x_pow1, self.m1),
get_hash(self.h1, 0, 7, self.x_pow1, self.m1))
        self.assertEqual(get_hash(self.h2, 0, 7, self.x_pow2, self.m2),
get_hash(self.h2, 0, 7, self.x_pow2, self.m2))

        self.assertEqual(get_hash(self.h1, 2, 3, self.x_pow1, self.m1),
get_hash(self.h1, 4, 3, self.x_pow1, self.m1))
        self.assertEqual(get_hash(self.h2, 2, 3, self.x_pow2, self.m2),
get_hash(self.h2, 4, 3, self.x_pow2, self.m2))

    def test_unequal_substrings(self):
        self.assertNotEqual(get_hash(self.h1, 1, 2, self.x_pow1, self.m1),
                            get_hash(self.h1, 3, 2, self.x_pow1, self.m1))
        self.assertNotEqual(get_hash(self.h2, 1, 2, self.x_pow2, self.m2),
                            get_hash(self.h2, 3, 2, self.x_pow2, self.m2))


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

Вывод по задаче:

Алгоритм работает за время O(n+q), где nnn — длина строки, а q — количество запросов. Для каждого запроса вычисление хешей занимает O(1) времени, благодаря предварительным вычислениям. Это позволяет эффективно решать задачу, даже для больших входных данных

## Задача №5. Префикс-функция

Текст задачи:

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

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

Листинг кода:

```
def compute_prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        j = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    return pi


if __name__ == "__main__":
    # Чтение входных данных
    with open("../txtf/input.txt", "r") as f:
        s = f.readline().strip()

    # Вычисление префикс-функции
    prefix_values = compute_prefix_function(s)

    # Запись результата в файл
    with open("../txtf/output.txt", "w") as f:
        f.write(" ".join(map(str, prefix_values)) + "\n")
```

Текстовое объяснение задачи:

Задача заключается в вычислении префикс-функции для всех непустых префиксов строки. Префикс-функция для строки sss на позиции i обозначает длину наибольшего собственного суффикса, который является

префиксом строки s[0..i]. Это важная составляющая многих алгоритмов обработки строк, например, алгоритма Кнута-Морриса-Пратта для поиска подстроки.

Для решения задачи используется алгоритм, который вычисляет префикс-функцию за линейное время O(n), где n — длина строки. В алгоритме используется массив pipipi, где каждый элемент pi[i] хранит значение префикс-функции для префикса строки s[0..i]. Алгоритм обрабатывает строку символ за символом, и если символы совпадают, увеличивает длину текущего префикс-сущности. Если нет — отходит назад по префикс-функции до первого возможного совпадения.

Тесты:

```
import unittest
from lab4.task5.src.main import compute_prefix_function

class TestPrefixFunction(unittest.TestCase):
    def test_case_1(self):
        s = "aaaAAA"
        expected_output = [0, 1, 2, 0, 0, 0]
        self.assertEqual(compute_prefix_function(s), expected_output)

    def test_case_2(self):
        s = "abacaba"
        expected_output = [0, 0, 1, 0, 1, 2, 3]
        self.assertEqual(compute_prefix_function(s), expected_output)

    def test_case_3(self):
        s = "a"
        expected_output = [0]
        self.assertEqual(compute_prefix_function(s), expected_output)


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

Вывод по задаче:

Временная сложность алгоритма для вычисления префикс-функции составляет O(n), где n — это длина строки

## Задача №7. Наибольшая общая подстрока

Текст задачи:

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 $s$ 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 $0 \le i < |s|$, $0 \le j < |t|$ и $l \ge 0$ 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:

```
import random


def compute_hashes(s, x, m):
    n = len(s)
    h = [0] * (n + 1)
    for i in range(1, n + 1):
        h[i] = (x * h[i - 1] + ord(s[i - 1])) % m
    return h


def get_hash(h, a, l, x_pow, m):
    return (h[a + l] - x_pow[l] * h[a] % m + m) % m


def preprocess_powers(x, max_len, m):
    x_pow = [1] * (max_len + 1)
    for i in range(1, max_len + 1):
        x_pow[i] = (x_pow[i - 1] * x) % m
    return x_pow


def has_common_substring(s, t, length, x, m1, m2):
    h1_s, h2_s = compute_hashes(s, x, m1), compute_hashes(s, x, m2)
    h1_t, h2_t = compute_hashes(t, x, m1), compute_hashes(t, x, m2)
```

```
    x_pow1, x_pow2 = preprocess_powers(x, max(len(s), len(t)), m1),
preprocess_powers(x, max(len(s), len(t)), m2)

    hashes_s = {}
    for i in range(len(s) - length + 1):
        hash_pair = (get_hash(h1_s, i, length, x_pow1, m1), get_hash(h2_s, i,
length, x_pow2, m2))
        hashes_s[hash_pair] = i

    for j in range(len(t) - length + 1):
        hash_pair = (get_hash(h1_t, j, length, x_pow1, m1), get_hash(h2_t, j,
length, x_pow2, m2))
        if hash_pair in hashes_s:
            return hashes_s[hash_pair], j, length

    return None


def longest_common_substring(s, t):
    left, right = 0, min(len(s), len(t))
    best_result = (0, 0, 0)

    m1, m2 = 10 ** 9 + 7, 10 ** 9 + 9
    x = random.randint(1, 10 ** 9)

    while left <= right:
        mid = (left + right) // 2
        result = has_common_substring(s, t, mid, x, m1, m2)

        if result:
            best_result = result
            left = mid + 1
        else:
            right = mid - 1

    return best_result


if __name__ == "__main__":
    with open("../txtf/input.txt", "r") as f:
        lines = f.read().strip().split("\n")

    results = []
    for line in lines:
        s, t = line.split()
        i, j, l = longest_common_substring(s, t)
        results.append(f"{i} {j} {l}")

    with open("../txtf/output.txt", "w") as f:
        f.write("\n".join(results) + "\n")
```

Текстовое объяснение задачи:

Задача заключается в нахождении наибольшей общей подстроки между двумя строками sss и ttt. Алгоритм использует хеширование и бинарный поиск для ускорения вычислений.

Алгоритм действует следующим образом:

1. Бинарный поиск по длине подстроки: Мы начинаем с бинарного поиска по длине наибольшей общей подстроки. Диапазон длин подстрок, который мы исследуем, от 0 до минимальной длины строк sss и ttt. Для каждой длины подстроки выполняем проверку на наличие общей подстроки.

2. Хеширование подстрок: Для каждой длины подстроки мы генерируем хеши всех подстрок длины k для строк sss и t. Для этого используются две разные хеш-функции с разными модульными значениями m11 и m2 для уменьшения вероятности коллизий.

3. Поиск совпадений хешей: После вычисления хешей подстрок для строки sss мы сохраняем их в хеш-таблице. Затем для строки ttt проверяем, есть ли в хеш-таблице совпадающий хеш. Если совпадение найдено, значит, существует общая подстрока длины k, и мы обновляем лучший результат.

4. Возврат результата: Бинарный поиск продолжает сужать диапазон возможных длин подстрок, пока не будет найдено максимальное значение. В конечном итоге алгоритм возвращает начальные позиции подстроки в обеих строках и её длину

Тесты:

```
import unittest
from lab4.task7.src.main import longest_common_substring

class TestLongestCommonSubstring(unittest.TestCase):
    def test_exact_match(self):
        """Тест, где обе строки полностью совпадают"""
        s = "abcdef"
        t = "abcdef"
        self.assertEqual(longest_common_substring(s, t), (0, 0, 6))  #
Однозначный ответ

    def test_no_common_substring(self):
```

```
        """Тест, где нет общей подстроки"""
        s = "abc"
        t = "xyz"
        self.assertEqual(longest_common_substring(s, t), (3, 0, 0))  #
Однозначный ответ

    def test_unique_longest_substring(self):
        """Тест с единственной наибольшей общей подстрокой"""
        s = "abcdef"
        t = "zabcxy"
        self.assertEqual(longest_common_substring(s, t), (0, 1, 3))  #
Единственная общая "abc"

    def test_single_character_match(self):
        """Тест, где общая подстрока - один символ"""
        s = "abcd"
        t = "xycz"
        self.assertEqual(longest_common_substring(s, t), (2, 2, 1))  #
Единственное совпадение "c"

    def test_common_substring_at_end(self):
        """Тест, где наибольшая подстрока в конце"""
        s = "xyzabcd"
        t = "mnopabcd"
        self.assertEqual(longest_common_substring(s, t), (3, 4, 4))  #
Единственная общая "abcd"

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

Вывод по задаче:

Алгоритм эффективно находит наибольшую общую подстроку между двумя строками с помощью бинарного поиска по длине подстроки и хеширования. Временная сложность алгоритма составляет $O((|s|+|t|)\log(\min(|s|,|t|)))$, где $|s|$ и $|t|$ — длины строк. Это решение эффективно для больших строк, поскольку бинарный поиск и хеширование значительно сокращают количество операций, избегая наивного сравнения всех подстрок.

## Вывод

В ходе выполнения лабораторной работы я изучила различные алгоритмы, которые помогают эффективно работать с подстроками, включая вычисление префикс-функции и поиск палиндромных троек. Я научилась оптимизировать алгоритмы для повышения их эффективности, особенно когда речь идет о больших входных данных.

В первой задаче я научилась вычислять префикс-функцию, что оказалось полезным для задач, связанных с поиском подстрок. Во второй задаче я работала с поиском палиндромных троек в строке, что продемонстрировало важность внимательного подхода к перебору возможных сочетаний и улучшению алгоритмов, чтобы уменьшить их сложность