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**](#_3xyvgf4lb01m) **3**

[Задача №2. Карта](#_y1vvkk22n6g5) 3

[Текст задачи:](#_1gubvkz3kuin) 3

[Задача №4. Равенство подстрок](#_a4mxuempfnni) 5

[Текст задачи:](#_lfovev2203p3) 5

[Задача №5. Префикс-функция](#_naesphyp00ss) 8

[Задача №7. Наибольшая общая подстрока](#_nf8orxk47h1h) 9

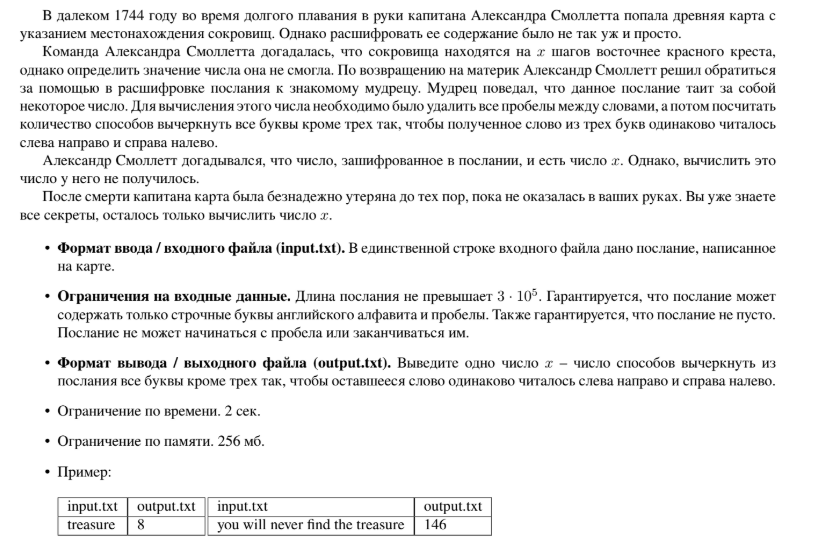
[Текст задачи:](#_kx2d24pkcpyt) 9

[**Вывод**](#_cc6bfpggan4) **13**

## **Задачи по варианту. Задачи по выбору**

### **Задача №2. Карта**

### Текст задачи:



Листинг кода:

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
   1. Используется функция combinations(range(n), 3), которая перебирает все возможные сочетания трех индексов из строки.
   2. 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])
   3. 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:

### 

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

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\_\_":

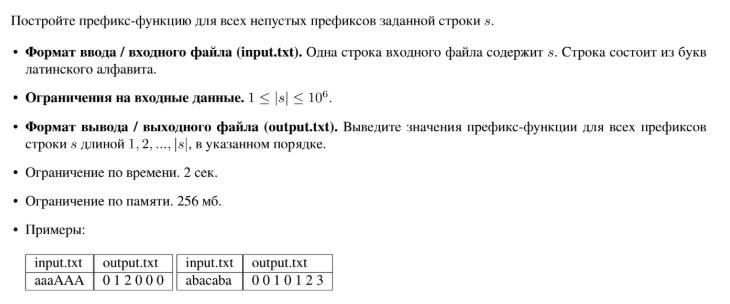
unittest.main()

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:



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\_\_":

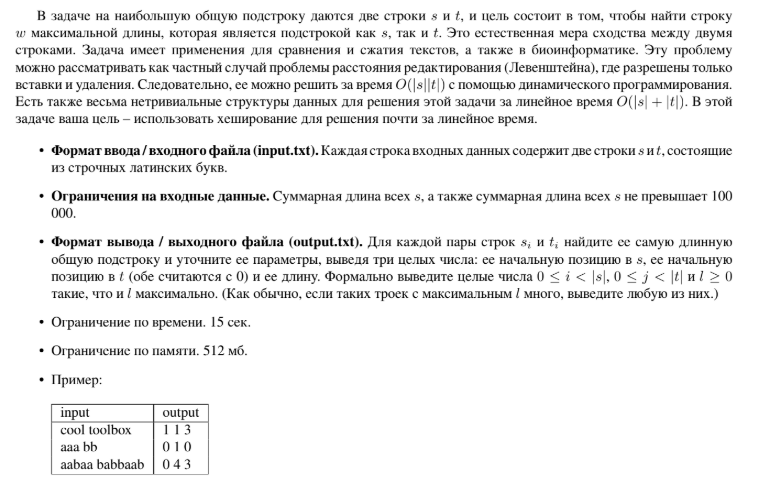
unittest.main()

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:



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