Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4
ПО КУРСУ «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»
ТЕМА: ПОДСТРОКИ
ВАРИАНТ 10

Выполнил:
Соболев А. А
K3240

Проверила:
Ромакина О. М.

Санкт-Петербург
2025

СОДЕРЖАНИЕ

ЗАДАЧИ ПО ВАРИАНТУ

4 задача. Равенство подстрок (1.5 балла)

Постановка 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.

Листинг кода

```
import time, tracemalloc

def build_hashes(s):
    n = len(s)
    m1 = 10**9 + 7
    m2 = 10**9 + 9
    x  = 91138233
    h1 = [0] * (n + 1)
    h2 = [0] * (n + 1)
    p1 = [1] * (n + 1)
    p2 = [1] * (n + 1)
    for i in range(1, n + 1):
        c     = ord(s[i-1]) - ord('a') + 1
        h1[i] = (h1[i-1] * x + c) % m1
        h2[i] = (h2[i-1] * x + c) % m2
        p1[i] = (p1[i-1] * x)      % m1
        p2[i] = (p2[i-1] * x)      % m2
    return h1, h2, p1, p2, m1, m2

def substrings_equal(a, b, length, h1, h2, p1, p2, m1, m2):
    ha1 = (h1[a + length] - h1[a] * p1[length] % m1 + m1) % m1
    hb1 = (h1[b + length] - h1[b] * p1[length] % m1 + m1) % m1
    if ha1 != hb1:
        return False
    ha2 = (h2[a + length] - h2[a] * p2[length] % m2 + m2) % m2
    hb2 = (h2[b + length] - h2[b] * p2[length] % m2 + m2) % m2
    return ha2 == hb2

def check_queries(s, queries):
```

```
    h1, h2, p1, p2, m1, m2 = build_hashes(s)
    results = []
    for a, b, length in queries:
        results.append("Yes" if substrings_equal(a, b, length, h1, h2, p1,
p2, m1, m2) else "No")
    return results

start_time = time.perf_counter()
tracemalloc.start()

with open('input.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
s = lines[0].strip()
q = int(lines[1].strip())
queries = [tuple(map(int, line.split())) for line in lines[2:2 + q]]

answers = check_queries(s, queries)

with open('output.txt', 'w', encoding='utf-8') as f:
    f.write("\n".join(answers))

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")
```

Объяснение

Сначала я за один проход по строке вычислил два массива префиксных хешей и массивы степеней базы по двум модулям. Затем для каждого запроса я через разность префиксных значений с учётом степеней получил хеши нужных подстрок и сравнил их.

Время и память

- Время выполнения: 0.004121 секунд (10 секунд ограничение)
- Использовано памяти: 0.017384 Мб (512 Мб ограничение)

Примеры

Входной файл	Выходной файл
trololo 4 0 0 7 2 4 3 3 5 1 1 3 2	Yes Yes Yes No

Вывод

В этой задаче я научился строить двойные префиксные хеши и отвечать на запросы равенства подстрок за константное время после линейной предобработки.

5 Задача. Префикс-функция (1.5 балла)

Постановка задания

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

Листинг кода

```
import time, tracemalloc

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


with open("input.txt", "r") as file:
    s = file.readline().strip()

start_time = time.perf_counter()
tracemalloc.start()

result = prefix_function(s)

end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")

with open("output.txt", "w") as file:
    file.write(" ".join(map(str, result)) + "\n"
```

Объяснение кода

Я в строке прошёл по её символам от второго до последнего, и для каждого положения поддерживал длину наибольшего собственного префикса, совпадающего с суффиксом, перемещаясь по уже рассчитанным значениям назад при несовпадении. При совпадении текущего символа с символом после этого префикса увеличивал счётчик на единицу и сохранял его в массив. За один линейный проход получил префикс-функцию для всех непустых префиксов строки.

Время и память

- Время выполнения: 0.000523 секунд (2 секунды ограничение)
- Использовано памяти: 0.000122 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
1 abacaba 2	1 0 0 1 0 1 2 3 2
1 aaaAAA	1 0 1 2 0 0 0 2

Вывод

В данной задаче я научился строить префикс-функцию строки за O(n) и оценивать производительность алгоритма по времени и памяти.

9 Задача. Декомпозиция строки (2 балла)

Постановка задачи

Строка ABCABCDEDEDEF содержит подстроку ABC , повторяющуюся два раза подряд, и подстроку DE , повторяющуюся три раза подряд. Таким образом, ее можно записать как ABC*2+DE*3+F , что занимает меньше места, чем исходная запись той же строки.

Задача – построить наиболее экономное представление данной строки s в виде, продемонстрированном выше, а именно, подобрать такие s_1, a_1, ..., s_k, a_k, где s_i - строки, а a_i - числа, чтобы s = $s_1 \cdot a_1 + \cdots + s_k \cdot a_k$. Под операцией умножения строки на целое положительное число подразумевается конкатенация одной или нескольких копий строки, число которых равно числовому множителю, то есть, ABC*2=ABCABC . При этом требуется минимизировать общую длину итогового описания, в котором компоненты разделяются знаком + , а умножение строки на число записывается как умножаемая строка и множитель, разделенные знаком * . Если же множитель равен единице, его, вместе со знаком * , допускается не указывать.

Листинг кода

```
import time, tracemalloc


def build_hashes(s):
    n = len(s)
    m1, m2 = 10**9+7, 10**9+9
    x = 91138233
    h1, h2 = [0]*(n+1), [0]*(n+1)
    p1, p2 = [1]*(n+1), [1]*(n+1)
    for i in range(1, n+1):
        c = ord(s[i-1]) - ord('a') + 1
        h1[i] = (h1[i-1]*x + c) % m1
        h2[i] = (h2[i-1]*x + c) % m2
        p1[i] = (p1[i-1]*x) % m1
        p2[i] = (p2[i-1]*x) % m2
    return h1, h2, p1, p2, m1, m2

def eq_sub(i, j, L, h1, h2, p1, p2, m1, m2):
```

```
    a1 = (h1[i+L] - h1[i]*p1[L] % m1 + m1) % m1
    b1 = (h1[j+L] - h1[j]*p1[L] % m1 + m1) % m1
    if a1 != b1: return False
    a2 = (h2[i+L] - h2[i]*p2[L] % m2 + m2) % m2
    b2 = (h2[j+L] - h2[j]*p2[L] % m2 + m2) % m2
    return a2 == b2

start_time = time.perf_counter()
tracemalloc.start()
with open('input.txt','r',encoding='utf-8') as f:
    s = f.readline().strip()
n = len(s)
h1,h2,p1,p2,m1,m2 = build_hashes(s)
INF = 10**18
dp = [INF]*(n+1)
rep = ['']*(n+1)
dp[0]=0
for i in range(n):
    for L in range(1, n-i+1):
        k = 1
        while i + L*(k+1) <= n and eq_sub(i, i+L*k, L, h1,h2,p1,p2,m1,m2):
            k += 1
        for t in range(1, k+1):
            j = i + L*t
            cost = L
            if t > 1:
                cost += 1 + len(str(t))
            new_cost = dp[i] + (0 if dp[i]==0 else 1) + cost
            if new_cost < dp[j]:
                dp[j] = new_cost
                block = s[i:i+L]
                suffix = f"*{t}" if t>1 else ''
                rep[j] = (rep[i] + ('+' if dp[i]>0 else '') + block +
suffix)
with open('output.txt','w',encoding='utf-8') as f:
    f.write(rep[n])
end_time = time.perf_counter()
cur_mem, peak_mem = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Время выполнения: {(end_time-start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_mem/(1024*1024)):.6f} Мб")
```

Объяснение

Сначала я за один проход по строке вычислил два массива префиксных хешей и массивы степеней базы для быстрого сравнения подстрок. Затем создал массив dp для хранения минимальной длины описания до каждой позиции и массив rep для самого описания. Для каждой позиции i и каждого длины L я с помощью хешей определил максимальное число подряд идущих повторов блока длины L и для каждого количества повторов t пересчитал стоимость описания до позиции $j = i + L \cdot t$, обновляя dp[j] и rep[j]. После обработки всех позиций в rep[n] оказалось оптимальное компактное представление строки.

Время и память

- Время выполнения: 0.004032 секунд (2 секунды ограничение)
- Использовано памяти: 0.017026 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной фалй
1 ABCABCDEDEDEF 2	1 ABC*2+DE*3+F 2
1 Hello 2	1 Hello 2

Вывод

В этой задаче я научился комбинировать хеширование строк и динамическое программирование для построения минимального текстового описания строки с учётом повторов.

ДОПОЛНИТЕЛЬНЫЕ ЗАДАЧИ

1 задача. Наивный поиск подстроки в строке (1 балл)

Постановка задачи

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

Листинг кода

```
import time, tracemalloc

def compute_prefix_function(pattern):
    m = len(pattern)
    pi = [0] * m
    j = 0
    for i in range(1, m):
        while j > 0 and pattern[i] != pattern[j]:
            j = pi[j - 1]
        if pattern[i] == pattern[j]:
            j += 1
        pi[i] = j
    return pi
def kmp_search(pattern, text):
    n, m = len(text), len(pattern)
    pi = compute_prefix_function(pattern)
    occurrences = []
    j = 0
    for i in range(n):
        while j > 0 and text[i] != pattern[j]:
            j = pi[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            occurrences.append(i - m + 2)
            j = pi[j - 1]
    return occurrences
start_time = time.perf_counter()
tracemalloc.start()
with open("input.txt", "r") as file:
    pattern = file.readline().strip()
    text = file.readline().strip()
```

```
occurrences = kmp_search(pattern, text)
with open("output.txt", "w") as file:
    file.write(f"{len(occurrences)}\n")
    file.write(" ".join(map(str, occurrences)) + "\n")
end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использование памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")
```

Объяснение

Данную задачу я решил с помощью алгоритма Кнута-Морриса-Пратта. Сначала я создаю массив π длины m, затем в цикле для i сравниваю pattern[i] и pattern[j]: при несовпадении откатываю j, при совпадении увеличиваю. После вычисления префикс-функции начинаю обход текста: в цикле сравниваю text[i] и pattern[j], при несовпадении снова откатываю j, а при совпадении увеличиваю на 1. Когда j достигает m, добавляю в список вхождений позицию и сбрасываю j для поиска следующих совпадений.

Время и память

- Время выполнения: 0.011962 секунд (2 секунды ограничение)
- Использование памяти: 0.017143 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
1 aba 2 abaCaba	1 2 2 1 5

Вывод

В этой задаче я научился находить все вхождения подстроки в строке с помощью алгоритма Кнута-Морриса-Пратта.

3 задача. Паттерн в тексте (1 балл)

Постановка задачи

В этой задаче ваша цель - реализовать алгоритм Рабина-Карпа для поиска заданного шаблона (паттерна) в заданном тексте.

Листинг кода

```
import time, tracemalloc

def rabin_karp(pattern, text, prime=10**9 + 7, base=31):
    m, n = len(pattern), len(text)
    pattern_hash = 0
    current_hash = 0
    base_pow = 1
    positions = []

    for i in range(m):
        pattern_hash = (pattern_hash * base + ord(pattern[i])) % prime
        current_hash = (current_hash * base + ord(text[i])) % prime
        if i > 0:
            base_pow = (base_pow * base) % prime
    if pattern_hash == current_hash and text[:m] == pattern:
        positions.append(1)
    for i in range(m, n):
        current_hash = (current_hash - ord(text[i - m]) * base_pow) % prime
        current_hash = (current_hash * base + ord(text[i])) % prime
        current_hash = (current_hash + prime) % prime
        if current_hash == pattern_hash and text[i - m + 1:i + 1] ==
pattern:
            positions.append(i - m + 2)
    return positions
start_time = time.perf_counter()
tracemalloc.start()
with open("input.txt", "r") as fin:
    pattern, text = fin.read().strip().split("\n")

positions = rabin_karp(pattern, text)

with open("output.txt", "w") as fout:
    fout.write(f"{len(positions)}\n")
    fout.write(" ".join(map(str, positions)) + "\n")
```

```
end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Время выполнения: {(end_time - start_time):.6f} секунд")
print(f"Использовано памяти: {(peak_memory / (1024 * 1024)):.6f} Мб")
```

Объяснение кода

Сначала я считаю хеш шаблона и первого фрагмента текста и при совпадении сразу проверяю их на равенство. Затем сдвигаю окно: обновляю хеш и при каждом совпадении хешей проверяю подстроку и записываю её позицию.

Время и память

- Время выполнения: 0.000635 секунд (2 секунды ограничение)
- Использовано памяти: 0.010287 Мб (256 Мб ограничение)

Примеры

Входной файл	Выходной файл
1 aba 2 abacaba	1 2 2 1 5
1 Test 2 testTesttesT	1 1 2 5
1 aaaaa 2 baaaaaaa	1 3 2 2 3 4

Вывод

В этой задаче я научился применять алгоритм Рабина-Карпа для поиска шаблона в тексте.

ВЫВОД

В данной лабораторной работе я научился работать алгоритмами для подстрок: построение префикс-функции и префиксных хешей, поиск по KMP и Рабина-Карпа.