

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

Отчет по лабораторной работе №4
по курсу «Алгоритмы и структуры данных»
Тема: Подстроки
Вариант 24

Выполнил:
Чан Тхи Лиен
К3140

Проверил:
Петросян.А.М

Санкт-Петербург
2025 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	
Задание 3. Паттерн в тексте	3
Задание 6. Z-функция	6
Задание 7. Наибольшая общая подстрока	8
Дополнительные задачи	
Задание 1. Наивный поиск подстроки в строке	11
Вывод	12

Цель работы

В данной лабораторной работе изучаются строки, поиск подстрок, алгоритмы Рабина-Карпа, Кнута-Морриса-Пратта, префикс-функция, Z-функция.

Задачи по варианту

Задание 3. Паттерн в тексте

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

1. Функция `rabin_karp`:

```
def rabin_karp(pattern, text): 8 usages
    p = 31 # Простое число
    mod = 10**9 + 9 # Большое простое число для хешей

    len_p = len(pattern)
    len_t = len(text)

    # Вычисляем p^i % mod заранее
    p_pow = [1] * (max(len_p, len_t))
    for i in range(1, len(p_pow)):
        p_pow[i] = (p_pow[i - 1] * p) % mod

    # Хеш паттерна
    hash_p = 0
    for i in range(len_p):
        hash_p = (hash_p + (ord(pattern[i]) - ord('a') + 1) * p_pow[i]) % mod

    # Префиксные хеши текста
    hash_t = [0] * (len_t + 1)
    for i in range(len_t):
        hash_t[i + 1] = (hash_t[i] + (ord(text[i]) - ord('a') + 1) * p_pow[i]) % mod

    occurrences = []
    for i in range(len_t - len_p + 1):
        # Хеш подстроки текста
        current_hash = (hash_t[i + len_p] - hash_t[i] + mod) % mod
        if current_hash == (hash_p * p_pow[i]) % mod:
            occurrences.append(i + 1) # Переводим в нумерацию с 1

    return occurrences
```

- **p** — основание полинома, простое число (31 — часто используется для латинских букв).
- **mod** — большое простое число, чтобы избежать переполнения и минимизировать коллизии.
- Сохраняем длины шаблона и текста.
- Предвычисляем все степени $p^i \bmod \text{mod}$ для последующего использования в хешировании.
- Это нужно для эффективного пересчёта хешей без повторного возведения в степень.

- Вычисляем хеш шаблона **pattern** по формуле:

$$\text{hash}_p = \sum_{i=0}^{\text{len}_p-1} (\text{code}(\text{pattern}[i]) \cdot p^i) \mod \text{mod}$$

- **ord(ch) - ord('a') + 1** — превращает буквы 'a'..'z' в 1..26.
 - Считаем префиксные хеши текста **text**:
 - **hash_t[i]** содержит хеш первых **i** символов.
 - Позволяет быстро вычислить хеш любой подстроки: **hash(text[l:r])**.
 - Инициализируем список вхождений — сюда будут добавляться позиции, где шаблон найден (**occurrences = []**).
 - **current_hash** — хеш подстроки **text[i:i + len_p]** получаем как разницу префиксных хешей.
 - **+ mod** для защиты от отрицательных значений.
 - Сравниваем с **hash_p * p_pow[i] % mod**
 - Если хеши совпадают — считается, что это совпадение. Добавляем позицию **i + 1** (нумерация с 1, как требует задача).
 - Возвращаем список всех позиций вхождений шаблона.
2. Функция для выполнения задач.

```
def task3(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    pattern = data[0]
    text = data[1]

    result = rabin_karp(pattern, text)

    list = []
    list.append(len(result))
    list.append(' '.join(map(str, result)))

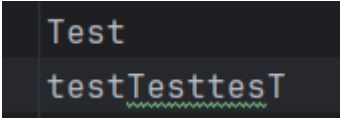
    write_file(PATH_OUTPUT, '\n'.join(map(str, list)) + '\n')

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

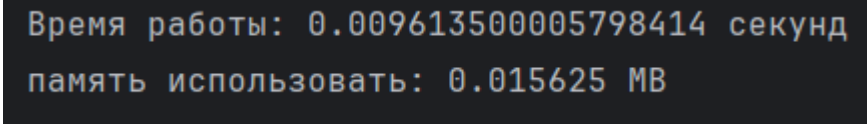
if __name__ == '__main__':
    task3()
```

3. Проведенные тесты.

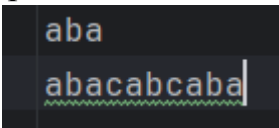
- Первый:

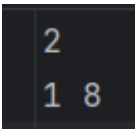
input.txt: 

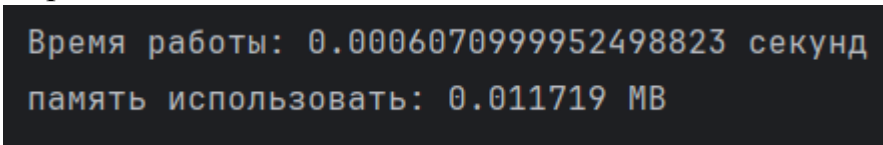
output.txt: 



- Второй:

input.txt: 

output.txt: 



Задание 6. Z-функция

Постройте Z-функцию для заданной строки s .

1. Функция `z_function`:

```
def z_function(s): 8 usages
    n = len(s)
    z = [0] * n
    l, r, k = 0, 0, 0

    for i in range(1, n):
        if i > r:
            l, r = i, i
            while r < n and s[r] == s[r - l]:
                r += 1
            z[i] = r - l
            r -= 1
        else:
            k = i - l
            if z[k] < r - i + 1:
                z[i] = z[k]
            else:
                l = i
                while r < n and s[r] == s[r - l]:
                    r += 1
                z[i] = r - l
                r -= 1

    return z
```

- $z[i]$ — длина совпадения префикса s с подстрокой, начинающейся с позиции i .
- $[l, r]$ — текущее окно, где $s[l:r+1]$ совпадает с префиксом s .
- k — вспомогательная переменная для сравнения уже известных значений.
- Итерируем по всем позициям строки, начиная с 1.
- Если i вне текущего окна ($i > r$), значит, нам нужно заново считать префикс вручную:
 - Начинаем сравнение $s[i:]$ с префиксом $s[0:]$.
 - Пока символы совпадают — расширяем правую границу r .
 - После цикла $z[i] = r - l$, длина совпадения.
 - Сдвигаем r обратно на 1, потому что вышли за границу.
- Если i внутри окна ($i \leq r$), то:
 - $k = i - l$ — позиция в префиксе, соответствующая i .
 - $z[k]$ — уже ранее рассчитанная длина совпадения.
 - Если $z[k]$ не выходит за границу текущего окна — просто копируем значение.
 - Если $z[k]$ выходит за границу — начинаем сравнение вручную, как в случае выше.
 - Возвращаем массив z , где каждый элемент — длина наибольшего префикса, совпадающего с суффиксом.

2. Функция для выполнения задач.

```
def task6(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

    z = z_function(data[0])

    write_file(PATH_OUTPUT, " ".join(map(str, z[1:])) + "\n")

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task6()
```

3. Проведенные тесты.

- input.txt: `abacaba`

- output.txt: `0 1 0 3 0 1`

```
Время работы: 0.007750399992801249 секунд
память использовать: 0.011719 MB
```

Задание 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.

1. Функция `compute_hashes`:

```
def compute_hashes(s, length, base, mod): 4 usages
    n = len(s)
    h = [0] * (n + 1)
    p = [1] * (n + 1)
    for i in range(1, n + 1):
        h[i] = (h[i-1] * base + ord(s[i-1])) % mod
        p[i] = (p[i-1] * base) % mod
    return h, p
```

- Предвычисляет префиксные хеши и степени основания `base`.
- Это позволяет быстро вычислять хеш любой подстроки `s[l:r]` за $O(1)$.

2. Функция `get_sub_hash`:

```
def get_sub_hash(h, p, l, r, mod): 4 usages
    return (h[r] - h[l] * p[r - l]) % mod
```

- Быстро считает хеш подстроки `s[l:r]` по формуле.
- Это классический способ вычитания из префиксного хеша.

3. Функция `has_common_substring`:

```
def has_common_substring(s, t, length, base1, mod1, base2, mod2):
    h1_s, p1_s = compute_hashes(s, length, base1, mod1)
    h2_s, p2_s = compute_hashes(s, length, base2, mod2)
    h1_t, p1_t = compute_hashes(t, length, base1, mod1)
    h2_t, p2_t = compute_hashes(t, length, base2, mod2)

    seen = {}
    for i in range(len(s) - length + 1):
        hs1 = get_sub_hash(h1_s, p1_s, i, i + length, mod1)
        hs2 = get_sub_hash(h2_s, p2_s, i, i + length, mod2)
        seen[(hs1, hs2)] = i

    for j in range(len(t) - length + 1):
        ht1 = get_sub_hash(h1_t, p1_t, j, j + length, mod1)
        ht2 = get_sub_hash(h2_t, p2_t, j, j + length, mod2)
        if (ht1, ht2) in seen:
            return seen[(ht1, ht2)], j
    return None
```


- Проверяет, существует ли общая подстрока длины **length** у **s** и **t**.
 - Для строки **s**:
 - Для всех подстрок длины **length** вычисляет **двойные хеши** и кладёт их в **seen**.
 - Для строки **t**:
 - Для каждой подстроки длины **length** вычисляется её двойной хеш.
 - Если такой хеш уже был у подстроки из **s**, значит, нашлась совпадающая подстрока.
 - Используется двойное хеширование:
 - Первая хеш-функция: **base1=911, mod1=10⁹ + 7**;
 - Вторая: **base2=3571, mod2=10⁹ + 9**.
4. Функция **longest_common_substring**:

```
def longest_common_substring(s, t): 1 usage
    base1, mod1 = 911, 10**9 + 7
    base2, mod2 = 3571, 10**9 + 9
    left, right = 0, min(len(s), len(t))
    res = (0, 0, 0)

    while left <= right:
        mid = (left + right) // 2
        found = has_common_substring(s, t, mid, base1, mod1, base2, mod2)
        if found:
            res = (found[0], found[1], mid)
            left = mid + 1
        else:
            right = mid - 1

    return res
```

- Это основная функция, которая ищет максимально возможную длину общей подстроки.
- Инициализируем границы: от **0** до **min(len(s), len(t))**.
- Двоичный поиск по длине подстроки **mid**.
- Для каждой длины проверяем, существует ли общая подстрока:
 - Если да → сохраняем позиции, увеличиваем **left**.
 - Если нет → уменьшаем **right**.
- Возвращаемое значение: **(start_in_s, start_in_t, length)**.

5. Функция для выполнения задач.

```
def task7(): 1 usage
    process = psutil.Process(os.getpid())
    t1_start = perf_counter()
    start_memory = process.memory_info().rss / 1024 / 1024
    data = read_file(PATH_INPUT)

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

    write_file(PATH_OUTPUT, "\n".join(results))

    t1_stop = perf_counter()
    end_memory = process.memory_info().rss / 1024 / 1024
    print('Время работы: %s секунд ' % (t1_stop - t1_start))
    print(f"память использовать: {end_memory - start_memory:.6f} MB")

if __name__ == '__main__':
    task7()
```

6. Проведенные тесты.

- input.txt:

cool toolbox
aaa bb
aabaa babbaab

- output.txt:

1	1	3
3	0	0
2	3	3

```
Время работы: 0.007969700003741309 секунд
память использовать: 0.015625 MB
```

Дополнительные задачи

Задание 1. Наивный поиск подстроки в строке

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

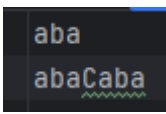
1. Функция `naive_substring_search`:

```
def naive_substring_search(p, t):  
    positions = []  
    for i in range(len(t) - len(p) + 1):  
        if t[i:i + len(p)] == p:  
            positions.append(i + 1)  
    return positions
```

- Создаём пустой список, в который будем добавлять номера позиций, с которых начинается вхождение p в t .
 - Запускаем цикл по всем возможным позициям начала подстроки в t .
 - $\text{len}(t) - \text{len}(p) + 1$ — столько возможных "окон" может быть, чтобы подстрока p полностью поместилась внутри t .
 - Проверяем: подстрока $t[i:i+\text{len}(p)]$ (то есть "окно" длины p , начиная с позиции i) совпадает с p .
 - Если совпадает — добавляем позицию $i + 1$ в список.
 - Возвращаем список всех позиций, с которых p входит в t .
- #### 2. Функция для выполнения задач.

```
def task1():  
    process = psutil.Process(os.getpid())  
    t1_start = perf_counter()  
    start_memory = process.memory_info().rss / 1024 / 1024  
    data = read_file(PATH_INPUT)  
  
    p = data[0]  
    t = data[1]  
  
    positions = naive_substring_search(p, t)  
  
    list = []  
    list.append(len(positions))  
    list.append(' '.join(map(str, positions)))  
  
    write_file(PATH_OUTPUT, '\n'.join(map(str, list)) + '\n')  
  
    t1_stop = perf_counter()  
    end_memory = process.memory_info().rss / 1024 / 1024  
    print('Время работы: %s секунд ' % (t1_stop - t1_start))  
    print(f'память использовать: {end_memory - start_memory:.6f} MB')  
  
if __name__ == '__main__':  
    task1()
```

3. Проведенные тесты.

- input: 

- output: 

Время работы: 0.000525799987372011 секунд
память использовать: 0.011719 МВ

Вывод:

- Выполнение всех заданий.
- Изучение и работа со строками, поиском подстрок, алгоритмами Рабина-Карпа, Кнута-Морриса-Пратта, префикс-функцией, Z-функцией.