Практикум
Задача о рюкзаке
Динамическое программирование. М...
Динамическое программирование. А...
Двумерная динамика
Задача о рюкзаке. Динамическое реш...
Наибольшая общая подпоследовател...
Разбор задачи: переход от теории к п...
Резюме
Оцените пройденные уроки
Резюме
Оцените пройденные уроки
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Резюме
Оцените пройденные уроки
Финальное задание спринта 7
Финальное задание спринта 7
Новое достижение
Оцените задание и сопровождение
СПРИНТ 8
Алгоритмы на строках
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы

Строки. Простейшие операции

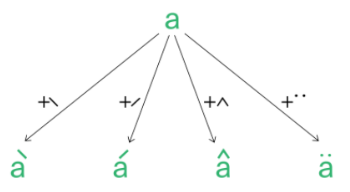
В этом спринте мы будем говорить про алгоритмы, работающие со строками. Строки, на первый взгляд, — довольно простые объекты, но и они могут преподнести немало сюрпризов.

Символы в строках

Для начала обсудим, что из себя представляет строковый тип данных.

Строка — это набор символов, расположенных в определённом порядке. Казалось бы, всё просто. Но за словом «символ» скрывается много разных тонкостей. Например, в составе строки могут находиться не только буквы алфавита, цифры и знаки пунктуации, но и служебные символы: перевод строки, горизонтальная и вертикальная табуляция и так далее.

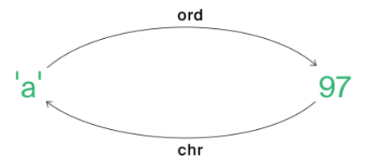
Некоторые символы даже не отображаются на экране. Например, символ BEL предназначен для того, чтобы издать звуковой сигнал, хотя в современных программах зачастую просто игнорируется. А другие символы — если их написать в определённом порядке — превращаются в один символ: какую-нибудь особенную букву или значок.



Символы могут комбинироваться с диакритическими знаками и превращаться в единый знак (в данном случае — в букву)

Большую часть спринта мы будем игнорировать все эти сложности и считать, что каждый символ — это самостоятельный элемент строки и занимает фиксированное число байтов (как правило, один). Пока что вам достаточно знать про кодировки совсем немного.

В большинстве языков программирования существует две функции: `ord` и `chr`, которые предназначены для того, чтобы переводить символ в его код и обратно.



Символ 'a' имеет код 97 в таблице символов ASCII. На базе ASCII сделаны все современные кодировки

Эта пара функций даёт возможность проводить с символами что-то вроде арифметических операций. Например, чтобы получить по символу `'a'` следующий символ в таблице кодировки — букву `'b'`, нужно выполнить следующий код: `chr(ord('a') + 1)`.

Обозначения

Строку можно представить в виде массива, элементами которого являются символы. Это сходство позволяет нам воспользоваться тем же интерфейсом получения элемента по индексу, что был у массивов: i-ый символ строки s будем обозначать как `s[i]`. Такой синтаксис работает не во всех языках, но во всех них есть аналоги для получения символа по индексу. Нумерация символов начинается с нуля.

В описаниях алгоритмов мы нередко будем использовать математическую форму записи: s_i. Когда нам понадобится разделить строку на несколько частей или даже на отдельные символы, в формуле мы будем ставить над ними горизонтальную черту, чтобы подчеркнуть, что это части единой строки. Например, запишем, что строка s состоит из n символов от s_0 до s_{n-1}:

$$s = \overline{s_0 s_1 \ldots s_{n-1}}.$$

Алла: А зачем изобретать какие-то специальные алгоритмы на строках, если мы уже умеем работать с массивами?

В массивах обычно каждый элемент важен сам по себе и обрабатывается отдельно от остальных. А когда мы работаем со строками, для нас важны не одиночные элементы, а целые группы подряд идущих символов.

Поэтому мы введём ещё одну запись: `s[j,k)` или $s_{[jk)}$. Так мы будем обозначать подстроку, состоящую из всех символов строки с j-го по $(k-1)$-ый.

Мы специально используем полуоткрытый интервал $[j, k)$, не включающий k.

Во-первых, благодаря этому можно закодировать подстроки нулевой длины. Например, $s_{[kk)}$ будет пустой строкой.

Во-вторых, это позволяет компактнее записывать разбиение строки на части: $s = s_{[0n)} = \overline{s_{[0k)} s_{[kn)}}$. Символ s_k будет содержаться в строке $s_{[kn)}$, но не войдёт в $s_{[0k)}$.

Длина строки $s_{[jk)}$ равна $k - j$. Если бы мы использовали интервал, включающий оба конца, в формуле возникла бы дополнительная единица, за которой пришлось бы внимательно следить.

Строка $s =$ "$abcdef$"

Что из себя представляет подстрока $s_{[2,4)}$?

- "bcd"
- "bc"
- "cde"
- ✓ "cd"

Позиции нумеруются с нуля, поэтому на позиции 2 стоит символ "c". Позиция 4 в подстроку уже не входит, а значит последняя позиция подстроки — 3, на которой стоит "d".

Вставка в середину строки

Итак, мы выяснили, что строки ведут себя аналогично массивам. О том, как работает вставка в конец массива, мы подробно говорили в прошлых уроках — это довольно простая и быстрая операция как для массивов, так и для строк.

Другое дело — вставка в середину строки. Чтобы её совершить, надо сначала выделить память, а затем сдвинуть все символы после индекса вставки на несколько позиций вперёд. Мы уже разбирали, как производить вставку элемента в середину массива. Но со строками ситуация оказывается сложнее, поскольку чаще всего нам нужно будет добавлять не один символ, а сразу несколько. По этой причине сдвиг следует проводить с конца строки, чтобы не стереть другие символы. И лишь после того как символы сдвинуты, можно на освободившееся место записывать вставляемую подстроку. Получается очень дорогая операция:

```
# Вставить строку substring в строку string перед позицией index.
функция insert(string, index, substring):
    length = длина(string)
    shift = длина(substring)
    если index > length: # index == length - край строки
        ошибка! нет такой позиции
    расширить область памяти строки до размера length + shift
    если length > 0:
        # Если length == 0, делать сдвиг нет смысла.
        # Кроме того, не следует в вычислениях писать (length - 1),
        #   не проверив, что индекс не ноль.
        # В некоторых языках длина представляется беззнаковым целым числом,
        #   в таком случае (length - 1) будет равен не -1, а числу MAX_INT,
        #   и цикл станет некорректным. Мы этого избегаем.
        для i от (length - 1) до index включительно, с шагом -1:
            string[shift + i] = string[i]

    для i от 0 до (shift - 1):
        string[index + i] = substring[i]
```

Если вам после этого курса захочется написать свой текстовый редактор, такой метод вам не подойдёт: всякий раз, когда пользователь будет добавлять символ в середину уже написанного текста, придётся проводить операцию сдвига, которая затрачивает $O(n)$ времени. Если вы столкнётесь с подобной проблемой, почитайте про верёвочную строку (англ. rope) — это специальная структура данных, которая описывает строку в виде дерева, состоящего из подстрок меньшего размера. Благодаря такой организации данных, вставка не требует сдвига всех символов, и работает за логарифмическое время. Правда, поиск символа по индексу в такой структуре, напротив, замедляется: вместо $O(1)$ он тоже займёт $O(\log n)$.

Решите задачи «Разворот строки», «Вставка строк».

Вперёд

Я Практикум
Задача о рюкзаке ✓
Динамическое программирование. М... ✓
Динамическое программирование. А... ✓
Двумерная динамика ✓
Задача о рюкзаке. Динамическое реш... ✓
Наибольшая общая подпоследовател... ✓
Разбор задачи: переход от теории к п... ✓
Резюме ✓
Оцените пройденные уроки ✓
Резюме ✓
Оцените пройденные уроки ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Резюме ✓
Оцените пройденные уроки ✓
Резюме ✓
Оцените пройденные уроки ✓
Финальное задание спринта 7 ✓
Финальное задание спринта 7 ✓
Новое достижение ✓
Оцените задание и сопровождение ✓
СПРИНТ 8
Алгоритмы на строках ✓
Строки. Простейшие операции ✓
Сравнение строк ✓
Подстроки, префиксы и суффиксы ✓

Сравнение строк

Алла

А какая самая простая задача при работе со строками?

Рита

Наверное, сравнение двух строк `first` и `second` на равенство.

Алла

Ну это же совсем легко. Строки равны, если у них одинаковая длина и все символы на соответствующих позициях — совпадают. То есть, если хотя бы один символ не совпадает, строки не равны.

```
функция compare(first, second):
    если длина(first) ≠ длина(second):
        вернуть False

    length = длина(first)
    для i от 0 до (length - 1):
        если first[i] ≠ second[i]:
            вернуть False

    вернуть True
```

Тимофей

Всё верно. И это сравнение работает за $O(n)$, где n — длина строки.

Алла

Но длина какой из строк имеется в виду, `first` или `second`?

Тимофей

Если эти строки имеют разную длину, то мы сразу, за $O(1)$ можем сказать, что строки не равны. А если строки имеют одинаковую длину, то не важно, какую из этих длин брать. Поэтому n — длина любой из строк.

Регистронезависимое сравнение

Гоша

Я вижу одну проблему. Люди часто пишут одни и те же слова то большими буквами, то маленькими. Но ведь получается, если у меня в телефонной книге сохранён контакт «Алла», а я напишу в поиске «АЛЛА», то ничего не найдётся.

Рита

Немного поправлю тебя. «Маленькие» буквы и «большие», а точнее строчные и прописные, называются в программировании буквами в нижнем и верхнем регистрах. Про слова, которые сочетают в себе оба этих типа букв, (как, например, «Алла»), говорят, что они записаны в смешанном регистре.

Чтобы сравнить две строки без учёта регистра, следует сначала привести их к единой форме. Например, заменить все буквы в верхнем регистре на аналогичные буквы в нижнем регистре: `compare(lowercase(first), lowercase(second))`.

Алла —> алла

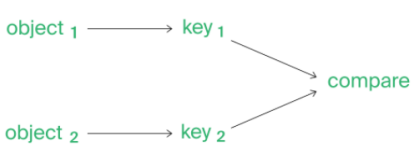
АЛЛА —> алла

Приведение к «нормальной форме» — часто применяемый метод. Помните, как вы искали группы анаграмм в спринте про хеши? Для решения этой задачи нужно было составить хеш-таблицу, в которой значениями были оригинальные слова, а ключами — их нормализованные версии. Например, можно было отсортировать буквы слова в лексикографическом порядке, и тогда `"LISTEN"` и `"SILENT"` попадали в одну и ту же корзину `"EILNST"`. Проверить, что два слова — анаграммы, можно следующим образом: `compare(sorted(first), sorted(second))`.

Обратите внимание, насколько разные сравнения можно проводить по одной и той же схеме:

- сначала мы по строке вычисляем некоторый ключ (например, нормальную форму, длину строки или любую другую её характеристику),
- а затем сравниваем значения ключей.

Мы подробно говорили об этом в спринте про сортировки.



Работа с ключами вместо объектов — распространённый приём. Он встречается и при работе с хеш-таблицами, и в сортировках, и при сравнении строк

Аналогичным образом можно сопоставлять слова, написанные кириллицей и транслитом. Или сравнивать слова по звучанию, преобразовывая их к специальному фонетическому алфавиту.

Гоша

Здорово. Про равенство двух строк теперь понятно. А как строки упорядочить? Какая строка больше, какая меньше?

Рита

Помните, мы говорили про лексикографическую сортировку: сначала сравниваются первые символы строки, если они равны, сравниваются вторые символы, и так далее, пока не найдётся различающийся символ. Сами символы сравниваются по их номеру в таблице кодировки.

Будьте аккуратны с кодировками, а также с регистром букв. Попробуем для примера сравнить две строки: `"A_a"` и `"Aa_"`, а затем привести их к нижнему и верхнему регистрам и снова сравнить уже нормализованные строки:

1) `"A_a" < "Aa_"` даёт `True` — сравнение исходных строк;

2) `"a_a" < "aa_"` даёт `True` — сравнение строк, приведённых к нижнему регистру;

3) `"A_A" < "AA_"` даёт `False` — сравнение строк, приведённых к верхнему регистру.

Такой неожиданный результат сравнения получается из-за того, что символ подчёркивания идёт в таблице кодировки ASCII позднее буквы верхнего регистра, но раньше букв нижнего регистра: `'A' < '_' < 'a'`.

Решите задачи «Пограничный контроль» и «Сравнить две строки».

Вперёд

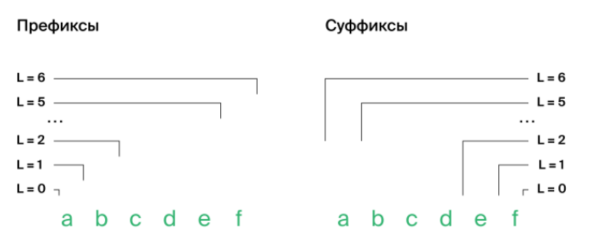
Подстроки, префиксы и суффиксы

Чтобы двигаться дальше, нам нужно формализовать некоторые понятия.

Глоссарий

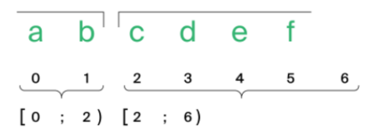
- Алфавит Σ — конечное множество символов. Например, все буквы русского и английского алфавитов, цифры, пробелы и прочие служебные знаки. Сложность некоторых алгоритмов будет зависеть от размера алфавита — $|\Sigma|$, поэтому разумно ограничивать его только теми символами, которые могут встретиться в используемых строках.
- Строка — некоторая последовательность символов алфавита. Длину строки будем обозначать $\text{length}(s)$ или просто $|s|$.
- Подстрока — непрерывная часть строки.

  Для строки $s = “abcdef”$ строка $“bcd”$ является подстрокой; строки $“acf”$ и $“cba”$ — не являются подстроками s.
- Конкатенация — склеивание строк в единую строку. Например, `concat("alpha", "bet")` даст `"alphabet"`.
- Префикс строки — подстрока, с которой строка начинается. Более формально префикс строки — это такая строка, которую можно дополнить, дописав справа ноль или больше символов, и получить исходную строку. Например, для строки s префиксами будут строки $""$, $"a"$, $"ab"$, $"abc"$, $"abcd"$, $"abcde"$, $"abcdef"$. Чтобы получить префикс длины l нужно взять первые l букв строки. Самый короткий префикс — это пустая строка, самый длинный — сама строка s целиком.
- Суффикс строки — подстрока, которой строка заканчивается. Например, у строки s суффиксы: $""$, $"f"$, $"ef"$, $"def"$, $"cdef"$, $"bcdef"$, $"abcdef"$.



У строки "abcdef" (длины 6) есть по 7 суффиксов и префиксов. Обратите внимание, что суффикс и префикс длины L=0 представляют собой пустую строку. А суффикс и префикс длины L=6 совпадают между собой и равны исходной строке

Любая строка $s_{[0,n)}$ может быть несколькими способами разделена на префикс $s_{[0,t)}$ и суффикс $s_{[t,n)}$.



Один из способов разделить строку на две половины: префикс $s_{[0,2)}$ и суффикс $s_{[2,6)}$

Обратите внимание, что для любой строки s пустая строка является префиксом $s_{[n,n)}$. Вся строка s целиком также является префиксом $s_{[0,n)}$. Одновременно эти строки являются и суффиксами s. Однако эти частные случаи удобнее всего рассматривать отдельно. Для этого существует понятие «собственного префикса» (англ. proper prefix).

Собственный префикс строки s — это любой префикс, кроме префикса длины $|s|$, совпадающего с полной строкой.

Иногда определение исключает из собственных префиксов ещё и пустую строку, но нам будет легче работать с префиксами, которым разрешено быть пустыми.

Аналогичным образом определяется собственный суффикс.

Отметьте все собственные префиксы строки $"abc"$

- $""$

  Из всех четырёх префиксов строки только $"abc"$ должен быть исключён из числа собственных префиксов.
- $"a"$

  Из всех четырёх префиксов строки только $"abc"$ должен быть исключён из числа собственных префиксов.
- $"ab"$

  Из всех четырёх префиксов строки только $"abc"$ должен быть исключён из числа собственных префиксов.
- $"abc"$

  Из всех четырёх префиксов строки только $"abc"$ должен быть исключён из числа собственных префиксов.

Зачем нужны префиксы и суффиксы

Алла

Зачем нужны подстроки, я понимаю. Например, слово в предложении — это подстрока. Если мы ищем в тексте какой-то отрывок, это тоже подстрока. Но зачем нам нужны ещё какие-то префиксы и суффиксы?

Тимофей

Подстрока — сложный объект, у неё есть две «плавающих» границы: позиция начала и позиция конца. А у префикса и суффикса одна из двух границ чётко закреплена: мы точно знаем, где находится начало префикса, где — конец суффикса. Поэтому работать с ними гораздо проще.

Алла

Да, но ведь большинство подстрок не являются ни префиксами, ни суффиксами. Чем же они нам помогут?

Тимофей

Давай в строке $s_{[0,n)}$ рассмотрим подстроку $s_{[l,r)}$. Чтобы задать такую подстроку, мы должны указать позиции её начала и конца: l и r. Но можно сделать эти два действия по-отдельности:

- Разделим строку по левой границе подстроки на префикс $s_{[0,l)}$ и суффикс $s_{[l,n)}$. Вся подстрока $s_{[l,r)}$ лежит внутри суффикса $s_{[l,n)}$. Получается, что взяв от исходной строки суффикс, мы зафиксировали одну из двух границ подстроки — левую.
- Рассмотрим теперь этот суффикс $s_{[l,n)}$ и возьмём от него префикс, заканчивающийся на позиции r. Это зафиксирует вторую границу подстроки — правую.

Алла

То есть подстрока — это префикс суффикса?

Тимофей

Да, ты правильно поняла. Любую подстроку можно представить в виде префикса некоторого суффикса.

Для примера возьмём строку `"show must go on"` и разметим в ней подстроку `"must"`.

```
show must go on  (текст)
      _________
     |must go on  (суффикс)
      ____
      must|       (префикс суффикса)
```

Можно сказать и наоборот: подстрока — это суффикс префикса.

```
show must go on  (текст)
_________
show must|       (префикс)
      ____
     |must       (суффикс префикса)
```

Алла

Всё равно не понимаю, зачем нам это?

Тимофей

Во многих задачах нам придётся перебирать разные подстроки — например, при поиске шаблона в тексте. Этот процесс мы разобьём на два цикла. Во внешнем цикле будем сдвигать левую границу подстроки, то есть перебирать все возможные суффиксы строки. Во внутреннем цикле сдвигаться будет правая граница — то есть мы будем перебирать префиксы этого суффикса.

Идея двух вложенных циклов, перебирающих левую и правую границы может показаться тривиальной. Но благодаря тому, что мы рассматриваем получающиеся подстроки в качестве таких простых объектов как префиксы и суффиксы, нам будет легко описать, как изменяются такие признаки как число совпадений подстроки с шаблоном при сдвиге одной из границ.

Кроме того, некоторые алгоритмы требуют сделать предподсчёт некоторых характеристик для различных участков строки. Предподсчёт для всех возможных подстрок занял бы очень много времени и памяти — ведь число строк квадратично зависит от длины строки. А все возможные префиксы (или суффиксы) можно перебрать за линейное время. Поэтому важно уметь формулировать вычисления в терминах префиксов и суффиксов строки.

В следующих нескольких уроках мы рассмотрим, как с помощью анализа префиксов эффективно искать шаблон в строке.

Вперёд

Я Практикум
Финальное задание спринта 7
Финальное задание спринта 7
Новое достижение
Оцените задание и сопровождение
СПРИНТ 8
Алгоритмы на строках
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Префикс-функция
Вычисление префикс-функции
Эффективный поиск шаблона в тексте
Префиксное дерево
Разбор задачи «Быстрое сравнение д...
Резюме
Оцените пройденные уроки
Финальное задание спринта 8

Поиск шаблона в строке. Наивный алгоритм

Наверное, самая востребованная операция при работе со строками — поиск подстроки в строке. Она применяется и в текстовых редакторах, и когда вы фильтруете данные в лог-файлах, и когда пытаетесь найти на компьютере документ с определённым словом в названии.

Задача формулируется следующим образом: найти индекс первого вхождения подстроки `pattern` в строке `text`. Чтобы избежать путаницы, будем называть подстроку `pattern` шаблоном. Давайте договоримся, что пока это будет фиксированная строка. А потом мы перейдём к более сложным шаблонам, под которые одновременно смогут подходить разные строки.

В этом уроке мы рассмотрим наивный алгоритм такого поиска. А следующий урок будет посвящён его более эффективной реализации.

Поиск первого вхождения шаблона в строку

Наивный метод предполагает, что мы будем перебирать все возможные позиции начала шаблона, а затем символ за символом сравнивать строку с шаблоном. На первом же несовпадении проверку можно закончить и перейти к следующей позиции строки.

Подстроки на зелёным фоне — это участки шаблона, которые совпали со строкой. Красным отмечены позиции, на которых обнаружено первое несоответствие текста с шаблоном. На большинстве стартовых позиций не совпадает первый же символ. Однако на позиции 2 первые несколько символов совпали, и только третья буква шаблона разошлась с текстом в строке

Реализовать эту функцию можно следующим образом:

```
# Найти первое вхождение подстроки pattern в строке text,
#   находящееся на позиции не раньше start.
функция find(text, pattern, start = 0):
    если длина(text) < длина(pattern):
        вернуть -1  # Длинный шаблон не может содержаться в короткой строке.
    для pos от start до (длина(text) - длина(pattern)):
        # Проверяем, не совпадёт ли шаблон, сдвинутый на позицию pos,
        #   с соответствующим участком строки.
        match = True
        для offset от 0 до длина(pattern) - 1:
            если text[pos + offset] ≠ pattern[offset]:
                # Одного несовпадения достаточно, чтобы не проверять
                #   дальше текущее расположение шаблона.
                match = False
                break
        # Как только нашлось совпадение шаблона, возвращаем его.
        # Это первое вхождение шаблона в строку.
        если match == True:
            вернуть pos
        # Если совпадение не нашлось, цикл перейдёт к проверке следующей позиции.
    # Числом -1 часто маркируют, что подстрока не была найдена,
    #   поскольку в строке нет позиции -1.
    # В качестве альтернативы можно возвращать None.
    вернуть -1
```

Поиск всех вхождений шаблона в строку

Гоша

Странно! Разве в этой функции должно быть не два параметра? Зачем в ней появился третий аргумент — `start`?

Рита

Этот параметр очень поможет нам, если мы захотим найти не только первое вхождение слова, но и все последующие.

Попробуем в строке `"мне ни холодно ни жарко"` найти слово `"ни"`. Первое вхождение будет на позиции 4, второе — на позиции 15.

В исходной строке размечены вхождения шаблона "ни". Позицией вхождения считается индекс строки, соответствующий нулевому символу шаблона. В нашем случае это позиции 4 и 15.

Если мы передадим в функцию `find` параметр `start = 0`, будет найдено вхождение на позиции 4. Тот же результат мы получим, если будем передавать в качестве `start` числа 1, 2, 3, 4. Но если мы передадим `start = 5`, результатом работы функции будет уже следующее вхождение: число 15 — это первое вхождение шаблона `"ни"` в наш текст, расположенное не раньше позиции 5.

Если мы будем запускать поиск шаблона, стартуя с позиций `start` от 5 до 15, мы получим один и тот же результат — 15. Начиная с позиции 16, результат снова изменится: он будет равен -1, так как в оставшейся части строки шаблон отсутствует.

Напишем алгоритм, который найдёт все вхождения подстроки в строку.

```
функция find_all(text, pattern):
    occurrences = [ ]
    start = 0 # Начнём поиск с начала строки.
    # Найдём первое вхождение, если оно есть.
    пока (pos = find(text, pattern, start)) ≠ -1:
        occurrences.push(pos) # Сохраним вхождение в список.
        start = pos + 1       # И продолжим поиск, начиная с позиции,
                              #   следующей за только что найденной.
    вернуть occurrences
```

Сложность наивного алгоритма

Алла

Я посчитала сложность этого метода. В нём используются вложенные циклы, поэтому алгоритм выполняется в худшем случае за $O(n \cdot m)$, где n — длина строки, m — длина шаблона.

Тимофей

Всё правильно. Кстати, такие алгоритмы тоже часто называют квадратичными, хотя тут и нет знака возведения в квадрат.

Алла

А почему так?

Тимофей

Потому что в худшем случае длина шаблона может быть пропорциональна длине строки. Например, шаблон имеет длину, равную половине исходной строки: $m = n/2$. Тогда нам нужно проверить приблизительно $n/2$ позиций и в каждой позиции провести $n/2$ сравнений. Получится, что $O(m \cdot n) = O(n^2)$. То есть алгоритм работает за время, квадратичное по длине исходной строки.

Алла

Теперь понятно, спасибо.

Гоша

А у меня ещё один вопрос есть. Разве проверка не будет завершаться в самом начале шаблона? Обычно ведь на первых же буквах становится понятно, что шаблон не подойдёт.

Тимофей

Ты прав, чаще всего так и бывает. Но алгоритм должен работать не только в типичных, но и в очень неудачных случаях, иначе можно будет подобрать киллер-последовательность, которая заставит программу зависнуть. Поскольку алгоритмы на строках часто работают с пользовательским вводом, злонамеренный пользователь легко сможет подсунуть алгоритму неудачные данные.

Гоша

Ого! А можешь показать киллер-последовательность, которая заставит алгоритм работать за квадратичное время?

Тимофей

О, это легко! Напишем длинную строку (например, из 1000 символов), состоящую из одинаковых букв: `"AAA...AAA"`. В качестве шаблона возьмём строку длиной 500, где первые 499 символов — та же самая буква, а последний символ отличается: `"AAA...AAZ"`. Получается, что на каждой позиции нам придётся проверить все 500 символов шаблона, чтобы убедиться, что строка нам не подходит.

Решите задачу «Поиск со сдвигом».

Вперёд

Я Практикум
Финальное задание спринта 7
Финальное задание спринта 7
Новое достижение
Оцените задание и сопровождение
СПРИНТ 8
Алгоритмы на строках
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Префикс-функция
Вычисление префикс-функции
Эффективный поиск шаблона в тексте
Префиксное дерево
Разбор задачи «Быстрое сравнение д...
Резюме
Оцените пройденные уроки
Финальное задание спринта 8

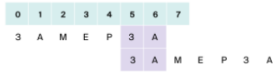
Префикс-функция

В прошлом уроке мы разобрали наивный алгоритм поиска шаблона в тексте. В худшем случае он будет затрачивать на работу $O(n^2)$ времени, где n — длина исходной строки. В следующих уроках мы расскажем о методах поиска, работающих за линейное время. Однако, чтобы их описать, нам понадобится ввести понятие префикс-функции.

Что такое префикс-функция

Неформально говоря, префикс-функция строки — это длина наибольшего перекрытия строки с собой, то есть когда символы в начале строки совпадают с символами в конце строки. Например, строка «облако» перекрывается сама с собой по букве «о», то есть имеет перекрытие длины 1.

Но знать перекрытие полной строки с собой нам будет недостаточно. Помимо этого, мы должны выяснить, как перекрывается с собой каждый из префиксов строки. Договоримся далее называть эти префиксы подстроками. Например, подстрока «замерза» перекрывается с собой по буквам «за» (длина перекрытия 2), а подстрока «замер» — не перекрывается с собой вообще (длина перекрытия 0).



Для подстроки $s_{[0,7)}$ = «замерза» существуют собственный префикс $s_{[0,2)}$ = «за» и такой же собственный суффикс $s_{[5,7)}$ = «за».

Теперь дадим формальное определение.

> Префикс-функцией от строки s и позиции i $(1 \leqslant i \leqslant |s|)$ называется длина k наибольшего собственного префикса подстроки $s_{[0,\ i)}$, который одновременно является её суффиксом (тоже собственным).

Другими словами, это наибольшее из таких чисел $k < i$, что в подстроке $s_{[0,\ i)}$ последние k символов совпадают с первыми k символами. То есть $s_{[0,\ k)} = s_{[i-k,\ i)}$.

Префикс-функция обозначается символом $\pi(s, i)$. Иногда мы будем опускать индекс и писать просто $\pi(s)$. В этом случае подразумевается, что в качестве подстроки берётся полная строка s.

Таким образом, $\pi(s) = \pi(s, |s|)$ — это длина наибольшего перекрытия строки s с собой. Используя это обозначение, можно записать $\pi(s, i) = \pi(s_{[0,i)})$.

Как вычисляется префикс-функция

Давайте сначала попробуем вычислить префикс-функцию вручную. Возьмём строку s = «ГОГОЛЬ» и её префикс-подстроку длины 4, $s_{[0,4)}$ = «ГОГО».

У этой подстроки есть совпадающие префикс и суффикс $s_{[0,\ 2)} = s_{[2,\ 4)}$ = «ГО». Более длинных перекрытий у неё нет. Например, если мы возьмём префикс и суффикс длины 3, то они не совпадут: «ГОГ» $\neq$ «ОГО».

Поэтому мы можем написать, что $\pi(«ГОГОЛЬ», 4) = 2$.

Теперь посчитаем префикс-функцию для всех подстрок-префиксов строки «ГОГОЛЬ». Для этого нарисуем максимальные перекрытия всех этих префиксов с собой:

Подстрока	Длина перекрытия подстроки со своим префиксом (префикс-функция)	Наилучший сдвиг подстроки относительно своего префикса
1 г	0	г г
2 го	0	г о г о
3 гог	1	г о г г о г
4 гого	2	г о г о г о г о
5 гогол	0	г о г о л г о г о л
6 гоголь	0	г о г о л ь г о г о л ь

На иллюстрации показаны самоперекрытия префиксов длины от 1 до 6 (префикс длины 0 — это пустая строка, которая не может перекрываться с собой)

Чаще всего префикс-функция сразу вычисляется для всех возможных префиксов некоторой фиксированной строки s, как в приведённом примере. Поэтому, несмотря на название, префикс-функция обыкновенно реализуется не одним числом, а массивом чисел длины $|s|$, в котором $\pi[i-1] = \pi(s, i)$:

```
π("ГОГОЛЬ") = [0, 0, 1, 2, 0, 0]
```

Обратите внимание, что массив всегда начинается с нуля, ведь единственный собственный префикс любой строки длины 1 — это пустая строка. А последний элемент массива — это длина перекрытия с собой полной строки.

Рассмотрим ещё один пример. Посчитаем $\pi(s)$ — префикс-функцию для строки s = 'ahahaha'.

1. Для $i = 1$ значение будет 0, так как собственный суффикс строки из одного символа 'a' — пустая строка.
2. Для $i = 2$ значение опять будет 0, так как для подстроки 'ah' её единственный собственный префикс не совпадает с её суффиксом, 'a' ≠ 'h'.
3. Для $i = 3$ всё интереснее. У строки 'aha' есть суффикс 'a', который совпадает с префиксом строки. Длина этого суффикса — 1, так что $\pi(s, 3) = 1$.
4. При $i = 4$ мы рассматриваем подстроку 'ahah'. У этой подстроки есть суффикс 'ah', который совпадает с префиксом 'ah' исходной строки. Длина этого префикса 2, следовательно, $\pi(s, 4) = 2$.
5. При $i = 5$ смотрим на строку 'ahaha', её суффикс 'aha' совпадает с префиксом 'aha'. Длина этого суффикса — 3, так что $\pi(s, 5) = 3$.
6. Аналогично, $\pi(s, 6) = 4,\ \pi(s, 7) = 5$.

В итоге для строки s = 'ahahaha' получили $\pi(s) = [0, 0, 1, 2, 3, 4, 5]$.

Какое значение префикс-функции для строки s='bubble'?

☒ $\pi(s)$ = [0,0,0,0,0,0]

У префикса 'bub' явно есть префикс, который совпадает с суффиксом.

⊖ $\pi(s)$ = [0,0,1,1,0,0]

○ $\pi(s)$= [1,0,1,1,0,0]

○ $\pi(s)$ = [0,0,1,2,0,0]

Выберите верное значение префикс-функции для строки s='to be or not to be'.

☒ $\pi(s)$ = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

○ $\pi(s)$ = [0,0,1,0,0,0,1,2,3,0,0,1,0,0,0,0,0,0]

⊖ $\pi(s)$ = [0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,2,3,4,5]

○ $\pi(s)$ = [1,2,3,0,0,0,0,0,0,0,0,0,1,0,1,2,3,4,5]

Вперёд

Вычисление префикс-функции

Тимофей

Вы уже умеете вычислять префикс-функцию на глаз. Теперь давайте посмотрим, как это мог бы делать компьютер.

Наивный алгоритм вычисления префикс-функции

Сначала опишем наивный алгоритм. Для каждого префикса строки мы должны вычислить его перекрытие с собой. Будем перебирать все возможные длины перекрытий в порядке убывания и проверять, действительно ли префикс и суффикс проверяемой длины совпадают.

```
N = length(s)
# Создадим массив π, состоящий из N нулей.
π = [0, 0, ...]

для i от 1 до N:
    # i — длина подстроки-префикса.
    substring = s[0, i)
    # Проверяем, перекрывается ли подстрока substring с собой по k символам.
    для k от (i - 1) до 0:
        # Для этого сравним префикс и суффикс соответствующих длин.
        prefix = substring[0, k)
        suffix = substring[i - k, i)
        если prefix == suffix, то:
            π[i-1] = k  # Запишем значение π-функции.
            break       # Дальше не проверяем — k идёт в порядке уменьшения.
```

Посчитаем сложность этого алгоритма. Фактически, в нём содержится три цикла:

- Внешний цикл перебирает длину i подстроки-префикса $s_{[0,i)}$ от 1 до N.
- Внутренний цикл итерируется по длине k префикса $s_{[0,k)}$ и суффикса $s_{[i-k,i)}$ получившейся подстроки.
- Во внутреннем цикле неявно спрятан ещё один — посимвольное сравнение строк, которое выполняется за $O(k)$.

Всё вместе это даёт кубическую сложность $O(N^3)$, что неприемлемо для длинных строк.

Однако можно посчитать префикс-функцию гораздо эффективнее, воспользовавшись методами динамического программирования.

Эффективный алгоритм вычисления префикс-функции

Напомним, что $\pi_i = \pi[i]$ — это длина наибольшего самоперекрытия подстроки $s_{[0,i+1)}$, где $0 \leqslant i < |s|$.

Базовый случай будет, когда подстрока-префикс имеет длину 1: $\pi_0 = 0$.

Предположим, что мы посчитали значение префикс-функции для всех длин префиксов от 0 до некоторого $i - 1$. Наша задача теперь — вычислить π_i.

Алгоритм будет искать такое перекрытие строки $s_{[0,i)}$, что его можно продолжить символом s_i.

Обозначим длину перекрытия за k. Тогда по определению перекрытия получим, что $s_{[0,k)} = s_{[i-k,i)}$. Эти символы входят и в расширение перекрытия, перепроверять их не требуется. А дополняется перекрытие с одного конца символом s_i, с другого — символом s_k. Это единственные два символа, которые нужно сравнить.

Чтобы выяснить, можно ли продолжить перекрытие, алгоритм проверяет, выполняется ли равенство $s_k = s_i$. Если да, то префикс $s_{[0,k)}$ и суффикс $s_{[i-k,i)}$ продолжаются одной и той же буквой, а значит выполняется равенство $s_{[0,k+1)} = s_{[i-k,i+1)}$. В этом случае мы можем записать, что $\pi_i = k + 1$.

Будем перебирать длины перекрытий, начиная от наибольшего. Самое длинное перекрытие даётся нам префикс-функцией от полной строки $s_{[0,i)}$: $k_1 = \pi_{i-1}$. Каждое следующее перекрытие мы получаем как префикс-функцию от предыдущего перекрытия $k_{j+1} = \pi(s_{[0,k_j)}) = \pi[k_j - 1]$. Если некоторое перекрытие удаётся продолжить, то поиски следует прервать.

В какой-то момент может оказаться, что очередная проверяемая длина $k_\bullet$ равна нулю. Это значит, что не нашлось такого суффикса строки $s_{[0,i)}$, который совпадает с префиксом этой строки, и какой можно было бы продолжить символом s_i. В таком случае, новое перекрытие может получиться длиной 0 или 1:

$$\pi_i = \begin{cases} 1, \text{ если } s[0] = s[i] \\ 0, \text{ если } s[0] \neq s[i] \end{cases}$$

Алла

Спасибо! Вроде бы понятно, но давай я попробую повторить. Получается так:

- Мы начинаем считать префикс-функцию с первого элемента по порядку, то есть перебираем $i = 1 ... (N-1)$. В $\pi[0]$ всегда кладём ноль.
- Чтобы узнать $\pi[i]$, мы сравниваем элементы $s[i]$ и $s[k]$. Для каждого i на первом шаге положим $k = \pi[i-1]$.
- Если $s[i] = s[k]$, то $\pi[i] = k + 1$. Иначе обновляем $k = \pi[k-1]$ и возвращаемся к предыдущему шагу алгоритма.
- Если в какой-то момент $k = 0$, а ответ мы так и не нашли, то сравниваем первый символ с i-ым, и переходим к следующему i.

Тимофей

Да, ты всё верно поняла, молодец. Алгоритм не из простых.

Гоша

Я, кажется, тоже понял алгоритм, хотя он довольно сложный. Кажется, его реализация займёт много строк кода.

Тимофей

Совсем нет. Код алгоритма получается достаточно компактным:

```
функция prefix_function(s):
    # Функция возвращает массив длины |s|
    π = [0, None, None, ...]
    для i из [1 .. |s|):
        k = π[i - 1]
        пока (k > 0) и (s[k] ≠ s[i]):
            k = π[k - 1]
        если s[k] == s[i], то:
            k += 1
        π[i] = k
    вернуть π
```

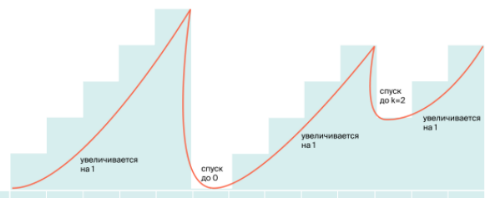
Сложность алгоритма вычисления префикс-функции

Алла

А почему сложность этого алгоритма линейная? Ведь, глядя на код, кажется, что сложность квадратичная: во внешнем цикле $O(N)$ итераций, и на каждой выполняется внутренний цикл `пока`.

Тимофей

Действительно, несмотря на вложенность, сложность алгоритма всё равно линейная. Сейчас расскажу, почему.



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
π[i]	0	1	2	3	4	5	0	1	2	3	4	2	3	4

Изменение значения префикс-функции с ростом длины префикса

Можно отметить важное свойство префикс-функции: $\pi[i] \leqslant \pi[i-1] + 1$. То есть префикс-функция от следующего элемента может превосходить префикс-функцию от текущего не более, чем на 1.

Значение префикс-функции может увеличиваться на 1 или падать до нуля (или до какого-то числа k больше нуля).

При увеличении на 1 мы просто сравниваем два символа. Сложность $O(1)$. А вот уменьшаться значение префикс-функции может в несколько этапов — именно за это отвечает цикл `пока`. На все спуски будет потрачено какое-то неизвестное нам количество операций.

Но значение префикс-функции не может упасть ниже нуля, поэтому суммарное количество падений не превзойдёт общее число подъёмов. А оно в свою очередь не может быть больше, чем N. Получаем, что и число увеличений, и число падений префикс-функции не превышают N, а значит суммарная сложность равна $O(N)$.

В следующем уроке мы рассмотрим, как применять префикс-функцию для поиска шаблона в тексте.

А пока решите задачу «Подсчёт префикс-функции».

Вперёд

Я Практикум
Финальное задание спринта 7
Финальное задание спринта 7
Новое достижение
Оцените задание и сопровождение
СПРИНТ 8
Алгоритмы на строках
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Префикс-функция
Вычисление префикс-функции
Эффективный поиск шаблона в тексте
Префиксное дерево
Разбор задачи «Быстрое сравнение д...
Резюме
Оцените пройденные уроки
Финальное задание спринта 8

Эффективный поиск шаблона в тексте

Теперь рассмотрим алгоритм поиска подстроки в строке с использованием префикс-функции.

Пусть у нас есть шаблон $p = "\mathrm{pattern}"$ и текст $t = "\mathrm{text}"$. Обе строки состоят из символов некоторого алфавита Σ. Составим из них одну общую строку, разделённую символом-сентинелом.

`Сентинел` — специальный символ, который не содержится в алфавите Σ. Его используют, чтобы при конкатенации двух строк обозначить место склейки. Мы в качестве сентинела возьмём символ $\#$.

Итак, у нас получилась строка $``\mathrm{pattern}\#\mathrm{text}"$. Именно от неё мы и будем вычислять префикс-функцию: $\pi(``\mathrm{pattern}\#\mathrm{text}")$.

В качестве примера возьмём шаблон *"sip"* и текст *"mississippi"*. Составим комбинированную строку *"sip#mississippi"* и вычислим от неё префикс-функцию.

Как мы говорили, каждая подстрока является суффиксом некоторого префикса. Значит, если шаблон встречается в строке, то на него оканчивается какой-то префикс. Тогда соответствующая подстрока должна иметь вид $"\mathrm{pattern}\#\ldots\mathrm{pattern}"$. В нашем случае такой подстрокой окажется $s_{[0,13)}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
s[i]	S	I	P	#	M	I	S	S	I	S	S	I	P	P	I
π[i]	0	0	0	0	0	0	1	1	2	1	1	2	3	0	0

Вычислим значения префикс-функции для строки, полученной конкатенацией шаблона, сентинела и текста

Префикс-функция π_{12} от такого префикса будет равна длине шаблона $|p|$. Это свойство любой позиции, на которой оканчивается вхождение шаблона. Значит, чтобы найти в тексте все вхождения шаблона, нам достаточно посчитать префикс-функцию комбинированной строки $"\mathrm{pattern}\#\mathrm{text}"$ и найти позиции, где она равна $|p|$. Эти позиции будут соответствовать последней букве шаблона.

Стоит сказать, что $|p|$ — это максимально возможное значение префикс-функции, ведь префикс большей длины обязательно захватывал бы сентинел, который встречается в строке лишь единожды. Чтобы суффикс начинался на $``\mathrm{pattern}\#"$, он должен совпадать со всей строкой, но несобственные суффиксы в вычислении префикс-функции не участвуют.

В таком методе для поиска шаблона в тексте не требуется ничего, кроме вычисления префикс-функции.

Эффективность алгоритма

Этот алгоритм состоит из двух шагов:

- построение префикс-функции,
- сканирование получившегося массива в поисках нужного значения.

Оба этапа занимают линейное время $O(|p| + |t|)$. Это намного быстрее, чем $O(|p| \cdot |t|)$ — скорость работы наивного алгоритма.

Описанный метод требует $O(|p| + |t|)$ дополнительной памяти для хранения значений префикс-функции. Впрочем, его возможно модифицировать так, чтобы он тратил только $O(|p|)$ памяти. Для этого следует хранить префикс-функцию только от первых $|p|$ символов комбинированной строки (то есть от той части строки, которая соответствует шаблону), а также префикс-функцию для последнего обрабатываемого элемента.

В таком случае построение префикс-функции и поиск её максимального значения будут совмещены. Промежуточные значения префикс-функции можно не запоминать: динамическое программирование всегда обращается только к последнему вычисленному значению префикс-функции и первым $|p|$ значениям.

```
функция поиск(p, text):
    # Функция возвращает все позиции вхождения шаблона в тексте.
    result = []
    s = p + '#' + text
    π = [0, None, None, ...]  # Массив длины |p|.
    π_prev = 0
    для i из [1 .. |s|):
        k = π_prev
        пока (k > 0) и (s[k] ≠ s[i]):
            k = π[k - 1]
        если s[k] == s[i], то:
            k += 1
        # Запоминаем только первые |p| значений π-функции.
        если i < |p|, то:
            π[i] = k
        # Запоминаем последнее значение π-функции.
        π_prev = k
        # Если значение π-функции равно длине шаблона, то вхождение найдено.
        если k == |p|, то:
            # i - это позиция конца вхождения шаблона.
            # Дважды отнимаем от него длину шаблона, чтобы получить позицию начала:
            #  - чтобы «переместиться» на начало найденного шаблона,
            #  - чтобы не учитывать добавленное "pattern#".
            result.добавить(i - 2 * |p|)
    вернуть result
```

Решите задачу «Глобальная замена».

Вперёд

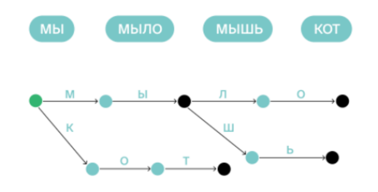
Префиксное дерево

В предыдущих уроках мы работали с одним текстом и одним шаблоном. А теперь давайте разберём методы, позволяющие работать сразу с большим набором шаблонов. Для этого нам понадобится структура данных, которая называется «бор», или «префиксное дерево». По-английски она называется вымышленным словом "trie". Это соединение слов "tree" — дерево, и "information retrieval" — поиск информации.

Алла: А что из себя представляет это дерево и для каких задач используется?

Тимофей: Префиксное дерево хранит в себе набор слов. Каждое слово «прокладывает путь» по дереву от корня до некоторого узла. Рёбра в этом пути соответствуют отдельным символам слова.

Путь от корня до узла, соответствующей добавленному слову, проходит по цепочке узлов, каждый из которых соответствует префиксу исходного слова. Поэтому дерево и называют префиксным.



Здесь изображён бор для множества слов: «мы», «мыло», «мышь», «кот». Чёрным цветом отмечены терминальные узлы. Узлы, покрашенные бирюзовым цветом, — промежуточные. Зелёным цветом отмечен корневой узел. Промежуточные узлы соответствуют префиксам «м», «мыш», «мыл», «к», «ко». Корень дерева соответствует пустой строке

В дереве есть два типа узлов. Терминальные узлы — это узлы, соответствующие словам, добавленным в бор. Терминальными, то есть конечными, их называют, поскольку в такие узлы ведут последние буквы слов. Но это не значит, что на терминальном узле ветка заканчивается, ведь некоторые слова можно дополнить до другого слова. Например, слово «мы» можно дописать до слова «мыло».

В терминальные узлы нередко записывают вспомогательную информацию про слово.

Помимо терминальных узлов в дереве есть промежуточные узлы. Они не соответствуют никакому слову, но являются префиксами для одного или нескольких слов, находящихся в боре.

Как найти слово в префиксном дереве

При работе с префиксным деревом необходимо уметь быстро находить узел, соответствующий строке запроса. Для этого алгоритм стартует из корня и совершает переходы между узлами — символ за символом. Если узел в дереве отсутствует, алгоритм возвращает `None`:

```
функция find_node(root, string):
    current_node = root
    для i от 0 до (length(string) - 1):
        symbol = string[i]  # На каждом шаге работаем с одним символом.

        если из current_node нет перехода по symbol:
            вернуть None

        # Сдвинуться на следующий символ.
        current_node = перейти из current_node по ребру symbol

    вернуть current_node
```

Эффективность этого алгоритма поиска ключа (слова) зависит от способа хранения рёбер. Предположим, длина слова равна n. Мощность алфавита возможных символов — $|\Sigma|$. Тогда каждый узел бора имеет не более, чем $|\Sigma|$ рёбер и дочерних узлов.

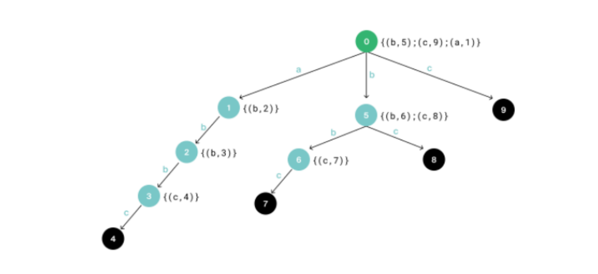
Хранить переходы по бору можно одним из следующих способов:

1. В списке рёбер $\{(\text{symbol}, \text{next node}); \dots\}$. Чтобы перейти к следующему узлу, нужно найти ребро, помеченное подходящим символом, а это требует полного перебора списка. При таком способе хранения сложность поиска узла, отвечающего строке, будет $O(n \cdot |\Sigma|)$.
2. В сбалансированном бинарном дереве поиска. Сложность каждой из трёх операций — $O(n \cdot \log|\Sigma|)$. Алфавит Σ обычно небольшой, значит, алгоритм будет работать быстро.
3. В хеш-таблице. Ключом будет символ, а значением — узел. Сложность поиска узла в этом случае будет равна $O(n)$ в среднем.
4. В массиве переходов длины $|\Sigma|$: на позиции с индексом i будет храниться ссылка на ребро, отвечающее i-му символу алфавита. Тогда переход по каждому символу будет выполняться за константное время, а строка в целом будет обработана за $O(n)$. По скорости этот метод оптимален, но при большом размере алфавита он будет расходовать много памяти.

Гоша: То есть в каждом узле хранится какая-то из перечисленных структур данных, да?

Тимофей: Верно.

Например, изобразим префиксное дерево, в котором переходы хранятся в виде списка рёбер:



В каждом узле записан список рёбер в формате {(символ, ссылка на следующий узел); ...}. Узлы пронумерованы, что позволяет хранить их в обыкновенном массиве, а в качестве ссылок использовать просто номера узлов. В это префиксное дерево записаны строки "abbc", "bbc", "bc", "c".

Как построить префиксное дерево

Построить бор — означает записать в него набор слов. При добавлении слова алгоритм действует почти так же, как при поиске: пытается пройти по рёбрам от корня до узла, соответствующего добавляемой строке. Отличие в том, что если в процессе работы алгоритма какое-то ребро не будет найдено, алгоритм создаст необходимое ребро и перейдёт в только что созданный узел. Когда алгоритм дойдёт до итогового узла, тот станет терминальным.

Например, пусть в боре уже записано одно слово «мыло». Тогда при добавлении слова «мышь» алгоритм сделает два шага по существующим рёбрам и дойдёт до узла, соответствующего подстроке «мы». А вот для буквы «ш» ребра нет, поэтому оно будет создано — вместе с новым узлом. Затем будет создано ребро для буквы «ь» и ещё один — на этот раз терминальный — узел.

Следующим мы запишем слово «мы». Для каждой буквы найдётся ребро в дереве, поэтому добавлять новые не нужно. Алгоритм дойдёт до соответствующего узла и отметит, что он терминальный. Теперь бор знает, что «мы» — не просто префикс слов «мыло» и «мышь», а самостоятельное слово.

Наконец добавим слово «кот». Уже на первом шаге потребуется добавить новое ребро, так как слов на букву «к» нам пока не встречалось. Так будут созданы обыкновенные узлы «к», «ко» и терминальный узел «кот».

Запишем алгоритм в виде псевдокода:

```
функция add_string(root, string):
    current_node = root
    для i от 0 до (length(string) - 1):
        symbol = string[i]  # На каждом шаге работаем с одним символом.

        если из current_node нет перехода по symbol:
            создать узел new_node
            создать ребро symbol из current_node в new_node

        # Сдвинуться на следующий символ.
        current_node = перейти из current_node по ребру symbol

    пометить current_node как терминальный узел
    вернуть current_node
```


Какая сложность алгоритма построения бора по заданному набору слов?

- $O(n)$, где n — количество слов
- $O(\log n)$, где n — количество слов
- $O(n \log n)$, где n — количество слов
- $O(|\Sigma|^2)$, где $|\Sigma|$ — размер алфавита
- $O(|\Sigma|)$, где $|\Sigma|$ — размер алфавита
- $O(\log|\Sigma|)$, где $|\Sigma|$ — размер алфавита
- $O(|\Sigma| \log|\Sigma|)$, где $|\Sigma|$ — размер алфавита
- $O(L)$, где L — суммарная длина слов во множестве
- $O(\log L)$, где L — суммарная длина слов во множестве
- $O(L \log L)$, где L — суммарная длина слов во множестве
- $O(L^2)$, где L — суммарная длина слов во множестве

При добавлении каждого слова нужно выполнить количество операций, пропорциональное его длине. Сложность алгоритма $O(L)$, где L — суммарная длина слов во множестве.

Применение префиксного дерева

Алла: Как строить бор — понятно. А для каких задач его используют?

Тимофей: Префиксное дерево, построенное из множества слов-шаблонов $\{p_i\}$, поможет быстро проверить, входит ли некоторая строка s в это множество или нет.

Например, мы можем добавить в префиксное дерево все слова русского языка. Это позволит нам создать функцию проверки орфографии, которая будет находить слово s в словаре (если оно там есть). В качестве бонуса функция будет предлагать варианты автодополнения.

Чтобы выполнить такую проверку, нужно по слову найти узел в префиксном дереве, как мы уже делали это ранее. После чего следует отыскать все терминальные узлы, являющиеся потомками найденного узла.

Итак, мы будем перебирать по очереди все символы строки s, начиная с корня и спускаясь по бору вниз. Далее возможны три случая:

1. Если какого-то из переходов не нашлось, значит, строки s в боре нет. Для нашей задачи это означает, что слово s не встречается в словаре и не является префиксом никакого из слов. Поэтому дополнить это слово невозможно.
2. Если переходы по каждому из символов нашлись, то в конце нужно проверить, является ли узел терминальным.

   (а) Если да, значит, искомая строка принадлежит множеству: $s \in \{p_i\}$ и является словарным словом.

   (б) Если нет, то искомая подстрока — префикс одной или более строк данного множества. Например, для строки «к» в боре существует узел (он расположен на пути к узлу «кот»), но он не является терминальным.

   В обоих случаях возможно предложить варианты автодополнения. Для этого следует найти все терминальные узлы, являющиеся потомками узла, на котором остановился алгоритм.

Напомним, что у терминального узла тоже могут быть потомки, в том числе терминальные. Например, слово «мы» есть в словаре, и оно имеет в качестве потомков слова «мышь» и «мыло», которые будут предложены в качестве вариантов автодополнения.

Сложность проверки линейно зависит от длины проверяемого слова.

Поиск сложного шаблона в строке

Гоша: Хорошо, мы разобрались, как искать строку среди множества шаблонов. Но может ли префиксное дерево помочь нам в противоположной задаче: найти шаблон в строке?

Рита: Нет, для этой задачи лучше воспользоваться алгоритмом Кнута-Морриса-Пратта.

Тимофей: Но есть похожая задача, в которой бор может быть полезен. Давайте будем искать в тексте не одну строку, а сразу несколько слов или подстрок. Например, любое из слов: «кот», «котик», «кошак», «кошка», «кошечка».

Можно сказать, что теперь шаблон, который мы ищем в тексте — это не строка, а набор строк: `(кот|котик|кошак|кошка|кошечка)`.

Гоша: А почему бы просто не воспользоваться алгоритмом поиска для каждой из подстрок?

Тимофей: Так сделать можно, но это решение окажется не самым эффективным. Ведь тогда нам придётся сравнивать текст с каждым шаблоном. Это займёт $O(n \cdot L)$ операций, где n — длина текста, $L = \sum_i m_i$ — сумма длин всех шаблонов.

Но ведь шаблоны не являются независимыми. Если мы видим, что в шаблоне «кошка» совпали 3 символа, то в шаблонах «кошак» и «кошечка» первые три символа можно уже не проверять, они тоже совпадут со строкой. Шаблоны «кот» и «котик» тоже можно не проверять: они точно уже не подойдут в данной позиции. Получается, шаблоны взаимосвязаны. На этом соображении основан метод, который позволяет искать в тексте несколько различных подстрок не поочерёдно, а одновременно.

Вначале мы составим бор, в который входят все строки, которые мы собираемся искать. Этот бор и будет играть роль шаблона. Затем мы модифицируем наивный алгоритм поиска строк. Как и прежде, мы будем по одной перебирать стартовые позиции, к которым можно «приложить» шаблон. Но проверять на каждом шаге мы теперь будем не одну подстроку, а весь шаблон в целом. Символы строки будут задавать путь по префиксному дереву. Если путь по существующим рёбрам дерева проложить не удаётся, шаблон в текущей стартовой позиции не подходит. Если же алгоритм в процессе движения по рёбрам дошёл до одного из терминальных узлов, шаблон найден.

Запишем это в виде псевдокода:

```
функция find_any(text, patterns):
    построить префиксное дерево trie по набору шаблонов patterns
    для pos от 0 до (length(text) - 1):  # Перебираем стартовые позиции.
        # Начинаем с корня бора.
        current_node = корень дерева trie
        # Перебираем символы шаблона, начиная со стартовой позиции.
        offset = 0
        mismatch_not_found = True  # Расхождений с шаблоном пока не найдено.
        пока mismatch_not_found и (pos + offset) < length(text):
            symbol = text[pos + offset]
            если из current_node есть переход по symbol:
                # Сдвинуться на следующий символ.
                current_node = перейти из current_node по ребру symbol
                если current_node - терминальный узел:
                    вернуть pos
                offset += 1
            иначе:  # Подходящее ребро отсутствует.
                # Найдено разночтение, мы должны завершить сканирование шаблона
                #   на текущей стартовой позиции. Внутренний цикл прервётся.
                # Работа на этом не заканчивается:
                #   будет взята следующая стартовая позиция.
                mismatch_not_found = False
    # Ни на одной стартовой позиции мы не дошли до терминального узла,
    #   значит, шаблон не найден.
    вернуть -1
```

В худшем случае от каждой позиции текста алгоритму придётся пройти по самой длинной ветви бора, поэтому такой способ займёт $O(n \cdot M)$ операций, где n — длина текста, а $M = \max m_i$ — длина самого длинного из искомых шаблонов. Поэтому алгоритм серьёзно экономит время, когда в вашем распоряжении много коротких шаблонов.

Решите задачу: «Случай верблюда».

Вперёд

Практикум
Финальное задание спринта 7
Финальное задание спринта 7
Новое достижение
Оцените задание и сопровождение
СПРИНТ 8
Алгоритмы на строках
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Строки. Простейшие операции
Сравнение строк
Подстроки, префиксы и суффиксы
Поиск шаблона в строке. Наивный алг...
Префикс-функция
Вычисление префикс-функции
Эффективный поиск шаблона в тексте
Префиксное дерево
Разбор задачи «Быстрое сравнение д...
Резюме
Оцените пройденные уроки
Финальное задание спринта 8

Разбор задачи «Быстрое сравнение двух строк»

Задача сравнения двух строк решается достаточно просто за линейное время: если надо сравнить строки s и t, то достаточно найти их наибольший общий префикс и посмотреть на первый различающийся символ, если такой есть, или заключить, что одна строка является префиксом другой. Результат операции сравнения можно представить в виде числа. Если $s < t$, то результат равен -1, если $s = t$, то 0, если $s > t$, то 1.

Теперь расширим эту задачу. Пусть нам дан набор, состоящий из n строк. Наша задача — научиться быстро отвечать на запросы вида «сравнить строки s и t из этого набора». Заодно разберёмся, что в данном случае значит «быстро».

Наивное решение

Наиболее простой подход состоит в том, чтобы применять обычный алгоритм сравнения двух строк. Тогда отвечать на запрос мы будем за $O(L)$, где L — длина самой длинной строки.

Если запросов будет поступать много, то имеет смысл воспользоваться кешированием ответов.

Кеширование — это техника запоминания уже запрошенных данных и полученных ответов на запросы. Например, в ленивом динамическом программировании мы сохраняли ответы для однажды посчитанных состояний, чтобы не пересчитывать их повторно.

В рассматриваемой задаче, сохраняя ответы для каждой запрашиваемой пары строк, мы сэкономим время на повторяющихся запросах, но увеличим пространственную сложность. На хранение всех результатов уйдет $O(n^2)$ дополнительной памяти, так как всего пар $\frac{n(n-1)}{2}$, и для каждой надо сохранить одно число — результат сравнения (-1, 0 или 1). Время работы на подсчёт ответов для всех пар составит $O(n^2L)$.

Улучшаем решение

Теперь научимся быстрее обрабатывать каждый запрос по отдельности. Рассмотрим утверждение для некоторого числа i и двух строк s и t: «Префиксы длины i строк s и t совпадают».

Заметим, что это утверждение до некоторого значения i истинно, но, начиная с некоторого i, ложно. Например, для строк $s = "abacaba"$ и $t = "abadaba"$ при $i = 1, 2, 3$ утверждение верно, а при $i \geq 4$ ложно.

i	1	2	3	4	5	6	7
s	A	B	A	C	A	B	A
t	A	B	A	D	A	B	A
s[0:i] = t[0:i]	1	1	1	0	0	0	0

Такое свойство некоторого условия называется «монотонностью». На самом деле мы уже имели дело с этим свойством, когда говорили о бинарном поиске.

Бинарный поиск может применяться только на монотонных данных. Помните, мы разбирали поиск по словарю? Записи в нём упорядочены лексикографически — именно это и есть монотонность, благодаря которой мы успешно применили бинарный поиск и нашли нужное нам слово.

Обозначим $T(k)$ — время на сравнение двух префиксов длины k на точное равенство. Тогда первую позицию, в которой строки s и t различаются, можно искать бинарным поиском за $O(T(L) \cdot \log L)$. Теперь подумаем, как можно сравнивать префиксы эффективно. Если сравнивать их наивно, за линейное время, $T(k) = O(k)$, то толку от использования бинарного поиска не будет.

Допустим, для строк s и t заранее вычислены массивы префиксных хешей. Тогда сравнить два префикса длины k можно за константное время $O(1)$, обратившись к нужному префиксному хешу для каждой строки. Значит, сравнить строки s и t мы можем за $O(\log L)$ при условии, что префиксные хеши уже посчитаны.

Мы готовимся к тому, что запросов будет приходить много, поэтому выгодно посчитать префиксные хеши заранее, сделав предподсчёт. Префиксный хеш одной строки s вычисляется за $O(|s|)$, поэтому суммарно это займёт $O(nL)$ времени, что сопоставимо с затратами на считывание строк. На хранение хешей уйдёт $O(nL)$ дополнительной памяти. Зато время ответа на каждый запрос улучшилось с $O(L)$ до $O(\log L)$!

PYTHON

```python
all_strings = ...  # Считываем n строк.
n = len(all_strings)

# Вычисляем префиксные хеши.
# Как это делать -- изучали в спринте 4.  :)
hashes = [get_hashes(s) for s in all_strings]
# В hashes[i] лежит массив префиксных хешей для i-ой строки.

queries = ...  # Считываем все запросы.
# Каждый запрос состоит из пары индексов строк, которые нужно сравнить.

for i, j in queries:
    # Итерация цикла сравнивает i-ую и j-ую строки.
    # В цикле перебираются пары индексов.
    s = all_strings[i]
    t = all_strings[j]

    # -- Бинарный поиск позиции первого различия --
    # Будем думать про left, mid и right как про длины общих префиксов.
    # Изначально мы можем гарантировать сопадение только нулевой длины.
    left = 0
    # Заведомо недостижимый ответ. Это, например, длина строки + 1.
    right = min(len(s), len(t)) + 1
    while right - left > 1:
        # Получаем середину.
        mid = (left + right) // 2
        # Так как mid - это длина, то [mid-1] - индекс последнего символа.
        if hashes[i][mid - 1] == hashes[j][mid - 1]:
            # Так как при длине префиксов mid они совпадают,
            # то передвигаем left.
```

Итого, используя алгоритм с предподсчётом префиксных хешей, мы получаем асимптотику $O(nL + Q \log L)$, если нам поступает Q запросов на сравнение пары строк. Иногда для алгоритма отдельно записывают время на предподсчёт и время ответа на запрос. В данном случае предподсчёт занял бы $O(nL)$, а ответ на каждый запрос $O(\log L)$.

Если бы мы захотели ещё улучшить рассмотренный алгоритм, то можно было бы добавить кеширование ответов. Проанализируем время работы алгоритма с добавленным кешированием. Предподсчёт хешей займёт $O(nL)$, затем на вычисление ответов для всех пар потребуется $O(n^2 \log L)$ операций, далее ответ на любой запрос выполняется за $O(1)$. Затраты на память составили бы $O(nL)$ на префиксные хеши плюс $O(n^2)$ на кеш ответов.

Вопрос. Возможно ли подобрать две неравных строки s и t таких, чтобы их полиномиальные хеши совпадали? Стоит ли учитывать это при решении задачи?

Вперёд

Резюме

При вставке текста в середину строки следует выделить больший блок памяти и сдвинуть участок строки, расположенный правее места вставки. Причём символы нужно смещать в конец выделенного блока памяти, начиная с последнего.

Мы разобрали, как сравнивать строки на равенство несколькими способами: от простого сравнения и сравнения без учёта регистра до более сложного — по произвольному признаку.

Мы рассказали про наивный алгоритм поиска подстроки в строке, работающий за квадратичное время и объяснили, на каких входных данных он будет работать максимально неэффективно.

Алгоритм Кнута-Морриса-Пратта основан на понятии префикс-функции $\pi(s)$ и позволяет искать подстроку в строке за линейное время, эффективно сдвигая шаблон вдоль строки.

Напомним, что префикс-функция — это массив чисел, сформированный для строки так, что i-ый элемент в нём равен длине наибольшего собственного суффикса $s_{[0,i)}$, который также является префиксом $s_{[0,i)}$. Префикс-функция шаблона может быть вычислена за линейное время.

Также мы рассказали про структуру данных бор, или префиксное дерево. Оно используется для эффективного хранения наборов строк, когда одиночной строки в качестве шаблона оказывается недостаточно.

Вперёд

Решите оставшиеся задачи по этой теме в Яндекс.Контесте: https://contest.yandex.ru/contest/26131/problems/.

Когда закончите, переходите к финальному заданию ниже.

Финальное задание спринта 8

Вам предстоит решить две задачи и пройти код-ревью по ним.

Сдавайте решения в Яндекс.Контесте, а когда получите OK по обеим задачам, — запакуйте исходные файлы в один архив и загрузите на ревью. В начале каждого решения в комментарии:

- напишите объяснение так же, как в предыдущем спринте;
- укажите ID успешной посылки, чтобы ревьюер мог удостовериться, что решение рабочее.

Важно: выберите один язык программирования и сдавайте финальные задачи на ревью только на нём. Задачи, которые не требуют ревью, вы можете сдавать на любом из доступных языков.

Успехов!

Вперед