Лекция 8.
Строки

179 Школа, Овчинников Андрей

29 ноября 2022 г.

Аннотация

Перед тем, как вы прочитаете то, что написано ниже обращу внимание на то, что в данном конспекте могут быть допущены ошибки и опечатки, если вы находите подобное, то пишите мне в личку или группу в телеграмм с упоминанием (*то есть через* @). Буду рад, если вы поможете довести конспект до хорошего безошибочного состояния.

1 О теме

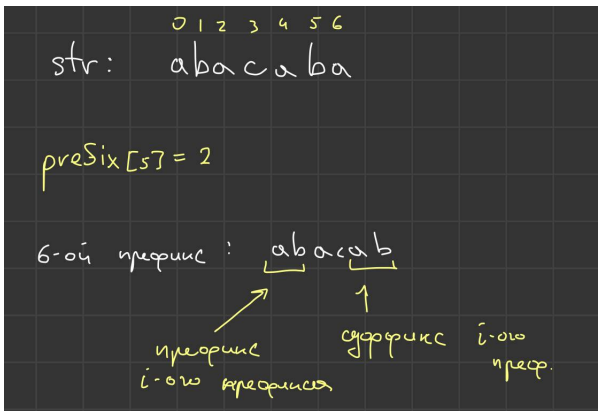
Мы рассмотрим несколько базовых алгоритмов применяемых к строкам. Они используются достаточно часто и поэтому очень полезны.
Также во избежания путаницы, символы в строке будем нумеровать с 0.

2 Префикс-функция

2.1 Определение

Дадим определение префикс-функции от строки и некоторой позиции в строке.
$\pi(str, i)$ = максимальному L, такому что $str[0 : L] = str[i - L + 1 : i + 1]$ (где запись $str[l : r]$ - означает подстроку строки str с позиции l по r, **не включая правую границу**).

То есть говоря словами, это длина максимального префикса строки str **не совпадающего** с префиксом длины i, такой что он совпадает с некоторым суффиксом префикса длины i (для лучшей понятности см. рисунок ниже).



2.2 Тривиальное решение

2.2.1 Описание

Проходим все позиции в строке, и начиная от каждой замеряем длину максимально совпадающего префикса с подстрокой начинающейся в текущей позиции.

2.2.2 Код

```
vector<int>
prefix_function(string &s)
{
    int sz = s.size();
    vector<int> prefix(sz, 0);
    for (int i = 1; i < sz; ++i) {
        int len = 0;
        while (len < sz && s[len] == s[i + len]) {
            prefix[i + len] = max(prefix[i + len], len);
            ++len;
        }
    }
    return prefix;
}
```

2.2.3 Оценка асимптотики

Получаем вполне понятные $O(n^2)$, где n - длина строки.

2.3 Быстрое рещение

2.3.1 Описание

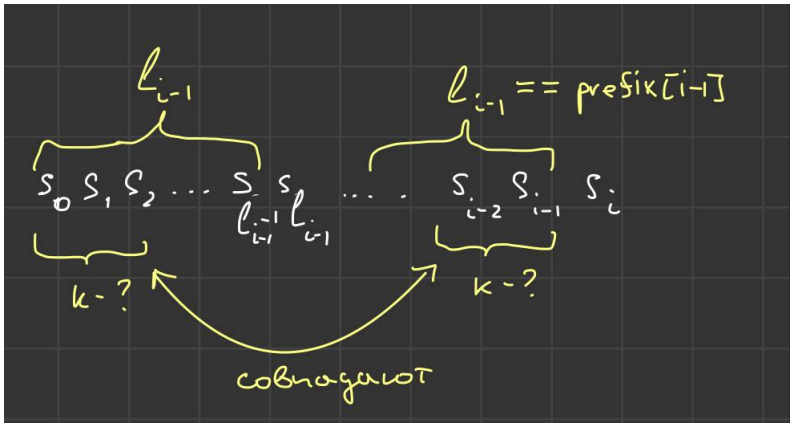
Для того, чтобы получить улучшение асимптотики, нам необходимо заметить три факта:

- $prefix[i+1] \leq prefix[i]+1$, очевидно сместившись на одну позицию, значение префикса возрастёт не более чем на 1, так как в противном случае, для предыдущего символа мы нашли не максимальный совпадающий префикс
- если мы рассмотрим $prefix[i]$, то оно $==$ $prefix[i-1]+1$, в случае $str[i] == str[prefix[i-1]]$, в противном случае нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к

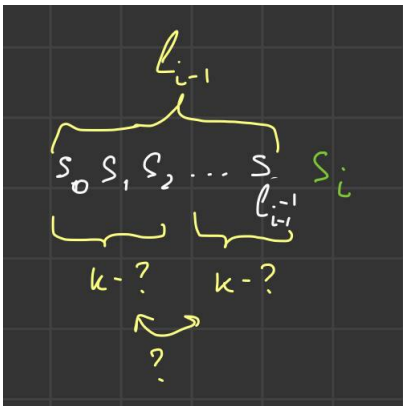
такой (наибольшей) длине $k < prefix[i-1]$, но чтобы по-прежнему выполняется префикс-свойство в позиции i, $str[0:k] == str[i-k:i-1]$, соответственно i-ый символ будет как бы продолжением для k-ого префикса.

Остался только вопрос, как находить такое k за быстро?

Давайте рассмотрим схему ниже.



По сути, подстроки $str[0 : prefix[i-1]]$ и $str[i-1-prefix[i-1] : i]$ совпадают, поэтому поиск k, будет одинаков для этих подстрок. Откуда становиться очевидным, что лучшее k это $prefix[prefix[i-1]-1] == prefix[l_{i-1}-1]$ (см. ниже)



А далее повторяем рассуждения:

- Если можем продлить $str[i]$-ым символом, то получили ответ
- Иначе повторяем рассуждения с поиском максимального k

2.3.2 Код

```
vector<int>
prefix_function(string &s)
{
    int n = s.size();
    vector<int> prefix(n, 0);
    for (int i = 1; i < n; ++i) {
        int k = prefix[i - 1];
        while (k > 0 && s[i] != s[k]) {
            k = prefix[k - 1];
        }
        if (s[i] == s[k]) {
            ++k;
        }
        prefix[i] = k;
    }
    return prefix;
}
```

2.3.3 Оценка асимптотики

Необходимо заметить, что сложность алгоритма оценивается числом выполненных *while*, тогда заметим, что k на каждой следующей операции возрастает не более чем на 1, следовательно, за всё время не может уменьшиться более n раз. Откуда итоговая асимптотика $O(n)$.

3 Z-функция

3.1 Определение

$\zeta(str, i)$ = максимальное k, такое что $str[0 : k] == str[i : i + k]$. Заметно, что определение очень похоже на префикс-функцию, и по сути они взаимозаминяемы (кроме того, вы можете при желании написать преобразование их массивов в друг друга).
Не будем тратить время на пояснение решений, доказательство будет/было рассказано не лекции.

3.2 Код

```
vector<int>
z_function(string &s)
{
        int n = s.size(), l = 0, r = 0;
        vector<int> z(n, 0);
        for (int i = 1; i < n; ++i) {
                if (i <= r) {
                        z[i] = min(r - i + 1, z[i - l]);
                }
                while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
                        ++z[i];
                }
                if (i + z[i] - 1 > r) {
                    l = i;
                    r = i + z[i] - 1;
                }
        }
        return z;
}
```

3.3 Оценка асимптотики

Алгоритм работает за $O(n)$, где n - длина строки.

4 Хеши

4.1 Определение + идея

Хеш функцией от строки будем называть следующее значение:

$$hash(str) = \sum_{i=0}^{len(str)} str[i] * base^{len(str)-i-1} \tag{1}$$

Понятно, что по сути мы превращаем строку в число, причём в системе счисления по основанию $base$, однако понятно, что строки даже не очень большой длины будут давать значение $hash$-функции куда большее чем мы способны хранить (и да, в питоне и с длинкой всё тоже не будет работать, так как строка длины 10^5, будет порождать число **не менее** $2^{(}100'000)$ не влезает даже в питоне)))).

Что же делать? Заметим, что при небольшом (в глобальном смысле) количестве строк, их хеш функции (и хеш-функции от их префиксов) 'займут' немного значение числовой прямой, так давайте зациклим их, беря хеш-функцию по модулю, однако для избежания очевидных проблем возникающих при делении нацело, давайте модуль и $base$ будут взаимно просты, кроме того будем брать их простыми числами.

По итогу, каждой строке соответствует *почти* уникальное число, что позволяет нам сравнивать их за $O(1)$. Однако следует помнить про коллизии.

4.2 Код

```
const int MAXN = 1e5;
const int base = 179;
const int MOD = 1e9 + 7;

int power[MAXN];

int
get_hash(vector<int> &hashes, int l, int r) { // (l, r]
    return ((hashes[r] - (hashes[l] * power[r - l])) % MOD + MOD) % MOD;
}

int
main()
{
//.....

//.....
    power[0] = 1;
    for (int i = 1; i < MAXN; ++i) {
        power[i] = power[i - 1] * base;
    }
    vector<int> hashes((int)s.size() + 1, 0);
    for (int i = 1; i <= (int)s.size(); ++i) {
        hashes[i] = (hashes[i - 1] * base + (int)s[i - 1]) % MOD;
    }
    return 0;
}
```