Суффиксный массив

Суффиксный массив

Пусть дана строка S длины n. Каждому суффиксу строки S поставим в соответствие его начальную позицию в строке S.

Суффиксный массив — массив, состоящий из начальных позиций всех суффиксов строки S, отсортированных в лексикографическом порядке.

Пример: $S = abacaba$, суффиксы в лексикографическом порядке: a, aba, $abacaba$, $acaba$, ba, $bacaba$, $caba$. Суффиксный массив: 6, 4, 0, 2, 5, 1, 3.

Зачем?

- Поиск наибольшего общего префикса подстрок
- Поиск количества различных подстрок
- Поиск k-й лексикографической подстроки

Массив LCP

LCP (Longest Common Prefix) двух строк S и T — длина наибольшего общего префикса этих строк.

Пример: $LCP(abacaba, abracadabra) == 2$

Массив LCP (lcp) - массив, состоящий из длин наибольших общих префиксов соседних в отсортированном порядке суффиксов строки S.

Алгоритм Касаи-Аримуры-Арикавы-Ли-Парка

Подготовка

- Пусть построен суффиксный массив SA.

```
SA = build_suffix_array(S)
```

- Построим обратный массив R, где $R[i]$ — позиция i-го суффикса в SA.

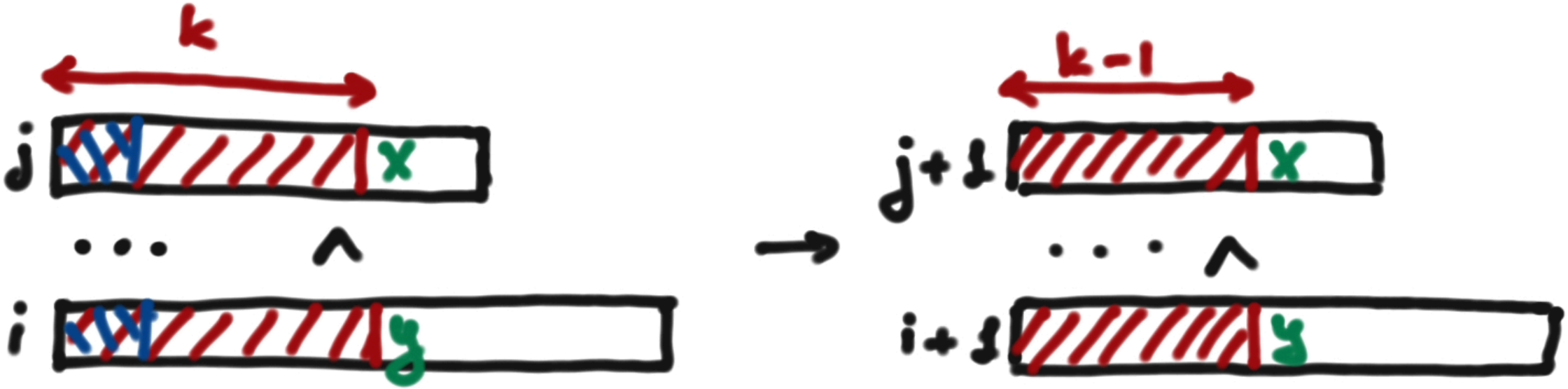
```
R = [0] * len(S)
for i in range(len(S)):
    R[SA[i]] = i
```

Алгоритм Касаи-Аримуры-Арикавы-Ли-Парка

Наблюдение

Пусть $LCP(S[i:], S[j:]) == k > 0$, а $R[i] > R[j]$ (суффикс i лексикографически больше суффикса j).

Тогда $LCP(S[i+1:], S[j+1:]) \geq k-1$, причем $R[i+1] > R[j+1]$.

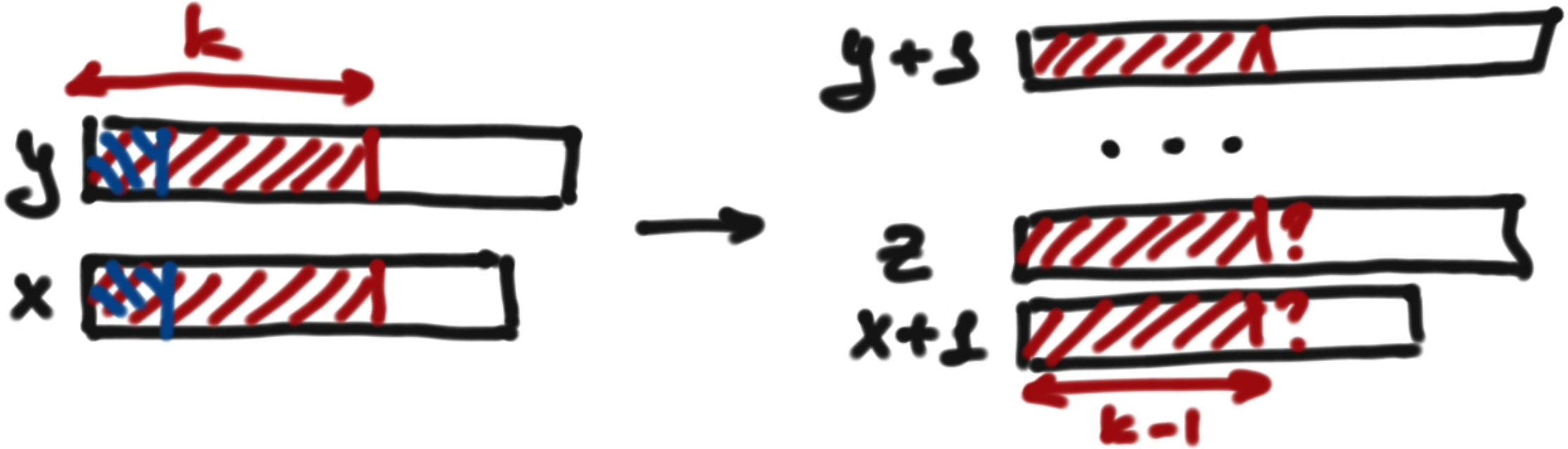


Алгоритм Касаи-Аримуры-Арикавы-Ли-Парка

Идея

Пусть знаем $lcp(R[x])$ (значение массива lcp в точке $R[x]$).

Тогда, согласно наблюдению, $lcp(R[x+1])$ не меньше $lcp(R[x]) - 1$. То есть можем начинать сравнивать суффиксы с позиции $lcp(R[x]) - 1$.



Пример

Алгоритм Касаи-Аримуры-Арикавы-Ли-Парка

Реализация

```
def build_lcp_array(S, SA, R):
    n = len(S)
    lcp = [0] * n
    k = 0  # длина наибольшего общего префикса с предыдущего шага
    for x in range(n):
        i = R[x]
        if i == 0:
            k = 0
            continue
        if k > 0:
            k -= 1
        while S[SA[i] + k] == S[SA[i - 1] + k]:
            k += 1
        lcp[i] = k
    return lcp
```

Алгоритм Касаи-Аримуры-Арикавы-Ли-Парка

```
def build_lcp_array(S, SA, R):
    n = len(S)
    lcp = [0] * n
    k = 0  # длина наибольшего общего префикса с предыдущего шага
    for x in range(n):
        i = R[x]
        if i == 0:
            k = 0
            continue
        if k > 0:
            k -= 1
        while S[SA[i] + k] == S[SA[i - 1] + k]:
            k += 1
        lcp[i] = k
    return lcp
```

Каждая итерация while цикла увеличивает k на 1, а значит, всего итераций будет не более $3n$ (так как for 1 раз уменьшает k до нуля и n раз на -1).

Поиск наибольшего общего префикса подстрок

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти наибольший общий префикс подстрок s_1, s_2, ..., s_m.

Поиск наибольшего общего префикса подстрок

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти наибольший общий префикс подстрок s_1, s_2, ..., s_m.

Решение.

Найдем позиции i_1, i_2, ..., i_m суффиксов s_1, s_2, ..., s_m в массиве SA.

Среди них найдем минимальную i_{min} и максимальную i_{max}.

Тогда ответом будет $LCP(SA[i_{min}], SA[i_{min}+1], ..., SA[i_{max}])$.

То есть минимум в lcp на отрезке $[i_{min}, i_{max}]$.

Свели к задаче RMQ.

Поиск количества различных подстрок

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти количество различных подстрок в строке S.

Поиск количества различных подстрок

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти количество различных подстрок в строке S.

Решение.

Пройдем по суффиксам в лексикографическом порядке.

Пусть на i-м шаге рассматриваем суффикс длины k. Он добавляет k подстрок. Но $lcp[i]$ из них уже были добавлены на предыдущих шагах.

Поэтому ответ - $\sum_{i=0}^{n-1}(n - SA[i]) - \sum_{i=1}^{n-1} lcp[i]$.

Поиск k-й лексикографической подстроки

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти k-ю лексикографическую подстроку в строке S.

Поиск k-й лексикографической подстроки

Пусть S — строка, SA — ее суффиксный массив, R — обратный массив, lcp — массив LCP.

Необходимо найти k-ю лексикографическую подстроку в строке S.

Решение.

Построение суффиксного массива

Алгоритм 1 (наивный)

Отсортируем с помощью быстрой сортировки все суффиксы строки S, сравнивая их лексикографически.

Асимптотика $O(n^2 \log n)$.

Алгоритм 2 (хеши)

Как сравнить строки быстрее чем за $O(n)$ с помощью полиномиального хеширования?

Предподсчитаем полиномиальный хеш.

Пусть даны две подстроки строки S. С помощью бинарного поиска найдем длину наибольшего общего префикса этих подстрок. Далее остается сравнить два символа.

Асимптотика сравнения суффиксов $O(\log n)$.

Асимптотика всего алгоритма $O(n \log^2 n)$.

Алгоритм 3

- Дополним строку S символом $\#$ (который заведомо меньше всех остальных) и построим суффиксный массив для нее.
- Сортировать суффиксы будем итеративно.
- На k-й итерации будем сортировать подстроки длины 2^k,
  то есть $\forall i : S[i : i + 2^k]$.
- При этом будем считать, что строка S циклическая, то есть $S[i] = S[i + n]$.

Алгоритм 3

- При $k = 0$ сортируем символы строки S. Это можно сделать с помощью сортировки подсчетом за $O(n)$.
- Построим массив c: $c[i] < c[j] \iff S[i] < S[j]$.

```
SA = sorted(range(n), key=lambda i: S[i])
c = [0] * n
for i in range(1, n):
    x, y = SA[i], SA[i - 1]
    c[x] = c[y] + (S[x] != S[y])
```

Пример:

Алгоритм 3

- Пусть на k-й итерации мы отсортировали подстроки длины 2^k и получили массив c: $c[i] < c[j] \iff S[i : i + 2^k] < S[j : j + 2^k]$.
- Отсортируем подстроки длины 2^{k+1}. Это сделать легко, если заметить, что $S[i : i + 2^{k+1}] = S[i : i + 2^k] + S[i + 2^k : i + 2^{k+1}] \sim (S[i : i + 2^k], S[i + 2^k : i + 2^{k+1}]) \sim (c[i], c[i + 2^k])$
  С помощью поразрядной сортировки это займет $O(n)$.
- После шага $k = \lceil \log n \rceil$ получим отсортированные подстроки длины n, а следовательно и суффиксы.

Асимптотика $O(n \log n)$.

Алгоритм 3

```
S += '#'
SA = sorted(range(n), key = lambda i: S[i])
c, new_c = [0,] * n, [0,] * n
for i in range(1, n):
    x, y = SA[i], SA[i - 1]
    c[x] = c[y] + (S[x] != S[y])

k = 0
while c[-1] != n - 1:
    SA = radix_sort(range(n), key = lambda i: (c[i], c[i + 2**k]))
    new_c[SA[0]] = 0
    for i in range(1, n):
        x, y = SA[i], SA[i - 1]
        new_c[x] = newA[y] + (c[x] != c[y] or c[x + 2**k] != c[y + 2**k])
    c, new_c = new_c, c
```