Lyndon 分解

定义

首先我们介绍 Lyndon 分解的概念。

Lyndon 串：对于字符串 s，如果 s 的字典序严格小于 s 的所有后缀的字典序，我们称 s 是简单串，或者 **Lyndon 串**。举一些例子，`a`, `b`, `ab`, `aab`, `abb`, `ababb`, `abcd` 都是 Lyndon 串。当且仅当 s 的字典序严格小于它的所有非平凡的（非平凡：非空且不同于自身）循环同构串时，s 才是 Lyndon 串。

Lyndon 分解：串 s 的 Lyndon 分解记为 $s = w_1 w_2 \cdots w_k$，其中所有 w_i 为简单串，并且他们的字典序按照非严格单减排序，即 $w_1 \geq w_2 \geq \cdots \geq w_k$。可以发现，这样的分解存在且唯一。

Duval 算法

解释

Duval 可以在 $O(n)$ 的时间内求出一个串的 Lyndon 分解。

首先我们介绍另外一个概念：如果一个字符串 t 能够分解为 $t = ww \cdots \overline{w}$ 的形式，其中 w 是一个 Lyndon 串，而 $\overline{w}$ 是 w 的前缀（$\overline{w}$ 可能是空串），那么称 t 是近似简单串（pre-simple），或者近似 Lyndon 串。一个 Lyndon 串也是近似 Lyndon 串。

Duval 算法运用了贪心的思想。算法过程中我们把串 s 分成三个部分 $s = s_1 s_2 s_3$，其中 s_1 是一个 Lyndon 串，它的 Lyndon 分解已经记录；s_2 是一个近似 Lyndon 串；s_3 是未处理的部分。

过程

整体描述一下，该算法每一次尝试将 s_3 的首字符添加到 s_2 的末尾。如果 s_2 不再是近似 Lyndon 串，那么我们就可以将 s_2 截出一部分前缀（即 Lyndon 分解）接在 s_1 末尾。

我们来更详细地解释一下算法的过程。定义一个指针 i 指向 s_2 的首字符，则 i 从 1 遍历到 n（字符串长度）。在循环的过程中我们定义另一个指针 j 指向 s_3 的首字符，指针 k 指向 s_2 中我们当前考虑的字符（意义是 j 在 s_2 的上一个循环节中对应的字符）。我们的目标是将 $s[j]$ 添加到 s_2 的末尾，这就需要将 $s[j]$ 与 $s[k]$ 做比较：

1. 如果 $s[j] = s[k]$，则将 $s[j]$ 添加到 s_2 末尾不会影响它的近似简单性。于是我们只需要让指针 j, k 自增（移向下一位）即可。
2. 如果 $s[j] > s[k]$，那么 $s_2 s[j]$ 就变成了一个 Lyndon 串，于是我们将指针 j 自增，而让 k 指向 s_2 的首字符，这样 s_2 就变成了一个循环次数为 1 的新 Lyndon 串了。

3. 如果 $s[j] < s[k]$，则 $s_2 s[j]$ 就不是一个近似简单串了，那么我们就要把 s_2 分解出它的一个 Lyndon 子串，这个 Lyndon 子串的长度将是 $j-k$，即它的一个循环节。然后把 s_2 变成分解完以后剩下的部分，继续循环下去（注意，这个情况下我们没有改变指针 j, k），直到循环节被截完。对于剩余部分，我们只需要将进度「回退」到剩余部分的开头即可。

实现

下面的代码返回串 s 的 Lyndon 分解方案。

C++

```cpp
// duval_algorithm
vector<string> duval(string const& s) {
  int n = s.size(), i = 0;
  vector<string> factorization;
  while (i < n) {
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) {
      factorization.push_back(s.substr(i, j - k));
      i += j - k;
    }
  }
  return factorization;
}
```

Python

```python
# duval_algorithm
def duval(s):
    n, i = len(s), 0
    factorization = []
    while i < n:
        j, k = i + 1, i
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
            factorization.append(s[i : i + j - k])
            i += j - k
    return factorization
```

复杂度分析

接下来我们证明一下这个算法的复杂度。

外层的循环次数不超过 n，因为每一次 i 都会增加。第二个内层循环也是 $O(n)$ 的，因为它只记录 Lyndon 分解的方案。接下来我们分析一下内层循环。很容易发现，每一次在外层循环中找到的 Lyndon 串是比我们所比较过的剩余的串要长的，因此剩余的串的长度和要小于 n，于是我们最多在内层循环 $O(n)$ 次。事实上循环的总次数不超过 $4n-3$，时间复杂度为 $O(n)$。

最小表示法（Finding the smallest cyclic shift）

对于长度为 n 的串 s，我们可以通过上述算法寻找该串的最小表示法。

我们构建串 ss 的 Lyndon 分解，然后寻找这个分解中的一个 Lyndon 串 t，使得它的起点小于 n 且终点大于等于 n。可以很容易地使用 Lyndon 分解的性质证明，子串 t 的首字符就是 s 的最小表示法的首字符，即我们沿着 t 的开头往后 n 个字符组成的串就是 s 的最小表示法。

于是我们在分解的过程中记录每一次的近似 Lyndon 串的开头即可。

C++

```cpp
// smallest_cyclic_string
string min_cyclic_string(string s) {
  s += s;
  int n = s.size();
  int i = 0, ans = 0;
  while (i < n / 2) {
    ans = i;
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) i += j - k;
  }
  return s.substr(ans, n / 2);
}
```

Python

```python
# smallest_cyclic_string
def min_cyclic_string(s):
    s += s
    n = len(s)
    i, ans = 0, 0
    while i < n / 2:
        ans = i
        j, k = i + 1, i
```

```python
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
            i += j - k
    return s[ans : ans + n / 2]
```

习题

- UVa #719 - Glass Beads

  本页面主要译自博文 Декомпозиция Линдона. Алгоритм Дюваля. Нахождение наименьшего циклического сдвига 与其英文翻译版 Lyndon factorization。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。
