

# 后缀数组

MengChunlei

February 28, 2021

- 定义
- 一般做法
- 改进做法
- 计数排序
- 基数排序
- 最后的改进
- 一些优化
- height 数组

## 1 定义

对于一个长度为  $n$  的字符串  $s : s[0], s[1], \dots, s[n-2], s[n-1]$ , 定义后缀  $suf[i] = s[i] \rightarrow s[n-1]$ , 即从下标  $i$  开始到结尾的子串.  $s$  对应的后缀数组也是一个长度为  $n$  的数组  $sa$ . 其中  $sa[i]$  表示在所有  $n$  个后缀从小到大进行排序后, 排第  $i$  名的后缀是  $suf[sa[i]]$ .

比如  $s = \text{aabaaaaab}$ , 那么对应的  $sa$  为:

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| <i>index</i> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| <i>s</i>     | a | a | b | a | a | a | a | b |
| <i>sa</i>    | 3 | 4 | 5 | 0 | 6 | 1 | 7 | 2 |

第 0 名:  $suf[3] = \text{aaaab}$

第 1 名:  $suf[4] = \text{aaab}$

第 2 名:  $suf[5] = \text{aab}$

第 3 名:  $suf[0] = \text{aabaaaaab}$

第 4 名:  $suf[6] = \text{ab}$

第 5 名:  $suf[1] = \text{abaaaaab}$

第 6 名:  $suf[7] = \text{b}$

第 7 名:  $suf[2] = \text{baaaaab}$

另外还有一个数组称之为 rank 数组  $rk$ , 它的长度也是  $n$ ,  $rk[i]$  表示后缀  $suf[i]$  的排名. 那么有以下关系:

$sa[rk[i]] = rk[sa[i]] = i$

## 2 一般做法

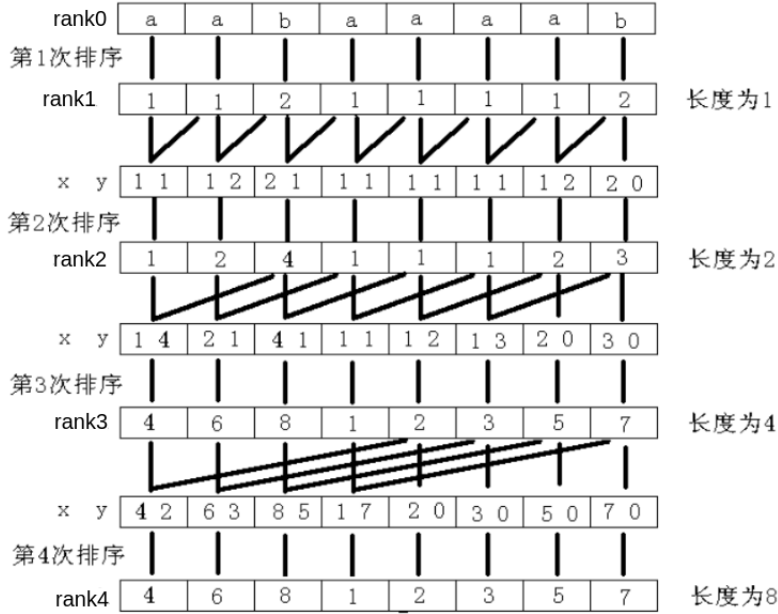
很容易有以下  $O(n^2 \log(n))$  的做法, 即直接对所有的后缀进行快排.

Listing 1: Normal

```
1 std::vector<int> ComputeSA1(const std::string &s) {
2     int n = static_cast<int>(s.size());
3     std::vector<int> sa(n);
4     for (int i = 0; i < n; ++i) {
5         sa[i] = i;
6     }
7     std::sort(sa.begin(), sa.end(), [&](int p1, int p2) {
8         for (; p1 < n && p2 < n && s[p1] == s[p2]; ++p1, ++p2);
9         return p1 < n && p2 < n ? s[p1] < s[p2] : p1 == n;
10    });
11    return sa;
12 }
```

### 3 改进做法

优化的思路是这样的: 设  $rk_w[i]$  表示以  $i$  开始的长度为  $w$  的子串的排名, 即  $s[i] \rightarrow s[i + w - 1]$ . 那么对于  $rk_{2w}$  来说, 它的计算可以借助于  $rk_w$  来进行. 简答来说, 就是以  $rk_w[i]$  为第一关键字, 以  $rk_w[i + w]$  为第二关键字进行排序, 就可以求出  $rk_{2w}$  数组. 示意图如下. 第  $i$  次排序的输入为  $rank[i - 1]$ , 输出为  $rank[i]$ .



Listing 2: Better

```

1  std::vector<int> UpdateRank(const std::vector<int> &sa,
2                             const std::vector<int> &old_rank, int n, int w) {
3      std::vector<int> rk(n);
4      rk[sa[0]] = 0;
5      for (int p = 0, i = 1; i < n; ++i) {
6          if (old_rank[sa[i]] == old_rank[sa[i - 1]] &&
7              ((sa[i] + w >= n && sa[i - 1] + w >= n) ||
8               (sa[i] + w < n && sa[i - 1] + w < n &&
9                old_rank[sa[i] + w] == old_rank[sa[i - 1] + w]))) {
10             rk[sa[i]] = p;
11         } else {
12             rk[sa[i]] = ++p;
13         }
14     }
15     return rk;
16 }
17 std::vector<int> ComputeSA2(const std::string &s) {
18     int n = static_cast<int>(s.size());
19     std::vector<int> sa(n);
20     std::vector<int> rk(n);
21     for (int i = 0; i < n; ++i) {
22         sa[i] = i;
23         rk[i] = s[i];
24     }
25     for (int w = 1; w < n; w <= 1) {
26         std::sort(sa.begin(), sa.end(), [&](int x, int y) {
27             return rk[x] == rk[y]
28                 ? (x + w < n && y + w < n ? rk[x + w] < rk[y + w] : x + w >= n)
29                 : rk[x] < rk[y];
30         });
31         rk = UpdateRank(sa, rk, n, w);
32     }
33     return sa;
34 }

```

这个算法的复杂度为  $O(n \log^2(n))$ . 接下来, 进一步的改进需要借助于计数排序和基数排序. 先了解下.

## 4 计数排序

假设一个待排列的序列中所有的元素只有  $C$  种, 那么可以通过计算每种元素的个数进行排列. 假设排列的元素有  $n$  个. 这个的复杂度为  $O(n + C)$ . 这是一个稳定排序算法. 示例代码如下:

Listing 3: Counting Sort

```
1 template <typename Elem>
2 std::vector<Elem> CountingSort(const std::vector<Elem> &eles,
3                               std::function<int(const Elem &)> functor,
4                               int C) {
5     int n = static_cast<int>(eles.size());
6     std::vector<int> bucket(C, 0);
7     for (int i = 0; i < n; ++i) {
8         ++bucket[functor(eles[i])];
9     }
10    for (int i = 1; i < C; ++i) {
11        bucket[i] += bucket[i - 1];
12    }
13    std::vector<Elem> result(n);
14    for (int i = n - 1; i >= 0; --i) {
15        result[--bucket[functor(eles[i])]] = eles[i];
16    }
17    return result;
18 }
```

## 5 基数排序

如果待排序的元素有  $k$  个关键字, 可以先对第  $k$  个关键字进行稳定排序, 然后再对第  $k - 1$  个元素进行稳定排序, 以此类推, 最后对第 1 个关键字进行稳定排序. 如下图所示:

|     |     |     |     |
|-----|-----|-----|-----|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |

下面是代码示例:

Listing 4: Radix Sort

```
1 template <typename Elem>
2 std::vector<Elem>
3 RadixSort(const std::vector<Elem> &eles, int k, const std::vector<int> &cs,
4           std::function<int(const Elem &, int index)> functor) {
5     std::vector<Elem> result = eles;
6     for (int i = k - 1; i >= 0; --i) {
7         result = CountingSort<Elem>(
8             result, std::bind(functor, std::placeholders::_1, i), cs[i]);
9     }
10    return result;
11 }
```

## 6 最后的改进

对第三步中  $O(n \log^2(n))$  的方法进行改进. 可以将循环内的每次排序看作是两个关键字的排序, 先对第二个关键字排序, 再对第一个关键字排序. 这个算法的复杂度为  $O(n \log(n))$

---

```

1  std::vector<int> ComputeSA3(const std::string &s) {
2      int n = static_cast<int>(s.size());
3      int m = std::max(256, n) + 1;
4      std::vector<int> sa(n), rk(n);
5      for (int i = 0; i < n; ++i) {
6          sa[i] = i, rk[i] = s[i];
7      }
8      for (int w = 1; w < n; w <= 1) {
9          sa = RadixSort<int>(sa, 2, {m, m}, [&](const int &x, int index) {
10              return index == 0 ? rk[x] : (x + w >= n ? 0 : rk[x + w] + 1);
11          });
12          rk = UpdateRank(sa, rk, n, w);
13      }
14      return sa;
15 }

```

---

## 7 一些优化

### 7.1 第二关键字排序

第二关键字的排序可以简化为如下实现:

Listing 6: Second Key Sort

---

```

1  std::vector<int> SortSecondKeyWord(const std::vector<int> &sa, int n, int w) {
2      std::vector<int> id(n);
3      int p = 0;
4      for (int i = n - 1; i >= n - w; --i) {
5          id[p++] = i;
6      }
7      for (int i = 0; i < n; ++i) {
8          if (sa[i] >= w) {
9              id[p++] = sa[i] - w;
10         }
11     }
12     return id;
13 }

```

---

### 7.2 记录 CountingSort 的结果

在 *CountingSort* 中记录 *functor* 的结果, 避免第二次调用

Listing 7: CountingSortFiner

---

```

1  template <typename Elem>
2  std::vector<Elem> CountingSortFiner(const std::vector<Elem> &eles, int C,
3                                     std::function<int(const Elem &)> functor) {
4      int n = static_cast<int>(eles.size());
5      std::vector<int> bucket(C, 0);
6      std::vector<int> bucket_result(n);
7      for (int i = 0; i < n; ++i) {
8          ++bucket[bucket_result[i] = functor(eles[i])];
9      }
10     for (int i = 1; i < C; ++i) {
11         bucket[i] += bucket[i - 1];
12     }
13     std::vector<Elem> result(n);
14     for (int i = n - 1; i >= 0; --i) {
15         result[--bucket[bucket_result[i]]] = eles[i];
16     }
17     return result;
18 }

```

---

### 7.3 优化计数排序 C 的大小

每次 C 的大小就是可以设置为 *UpdateRank* 中 *p* 的大小, 而不是每次都是 *m*.  
加上所有优化后的代码如下:

Listing 8: Best

---

```
1 std::vector<int> UpdateRankFiner(const std::vector<int> &sa,
2                                const std::vector<int> &old_rank, int n, int w,
3                                int *last_p) {
4     std::vector<int> rk(n);
5     rk[sa[0]] = 0;
6     int p = 0;
7     for (int i = 1; i < n; ++i) {
8         if (old_rank[sa[i]] == old_rank[sa[i - 1]] &&
9             ((sa[i] + w >= n && sa[i - 1] + w >= n) ||
10              (sa[i] + w < n && sa[i - 1] + w < n &&
11               old_rank[sa[i] + w] == old_rank[sa[i - 1] + w]))) {
12             rk[sa[i]] = p;
13         } else {
14             rk[sa[i]] = ++p;
15         }
16     }
17     *last_p = p + 1;
18     return rk;
19 }
20
21 std::vector<int> ComputeSA4(const std::string &s) {
22     int n = static_cast<int>(s.size());
23     int m = std::max(256, n) + 1;
24     std::vector<int> sa(n);
25     std::vector<int> rk(n);
26     std::vector<int> cnt(m, 0);
27     for (int i = 0; i < n; ++i) {
28         ++cnt[rk[i] = s[i]];
29     }
30     for (int i = 1; i < m; ++i) {
31         cnt[i] += cnt[i - 1];
32     }
33     for (int i = n - 1; i >= 0; --i) {
34         sa[--cnt[rk[i]]] = i;
35     }
36     for (int w = 1; w < n; w <= 1) {
37         sa = CountingSortFiner<int>(SortSecondKeyWord(sa, n, w), m,
38                                     [&](const int &x) { return rk[x]; });
39         rk = UpdateRankFiner(sa, rk, n, w, &m);
40     }
41     return sa;
42 }
```

---

## 8 height 数组

*height* 数组是一个长度为 *n* 的数组,  $height[i] = lcp(suf[sa[i]], suf[sa[i - 1]])$ , 即它表示排名第 *i* 的后缀和排名第 *i - 1* 的后缀的最长公共前缀.  $height[0] = 0$

*height* 数组有一个性质:  $height[rk[i]] \geq height[rk[i - 1]] - 1$ . 证明如下:

设  $suf[i - 1] = aAD$ , (*A*, *D* 分别表示一个串, 可能为空, *a* 表示一个字符). 那么  $suf[i] = AD$ .

另外假设  $suf[sa[rk[i - 1] - 1]] = aAB$ , (注意  $B < D$ ), 所以有  $lcp(suf[i - 1], suf[sa[rk[i - 1] - 1]]) = aA$ . 也就是  $height[rk[i - 1]]$  的长度是  $|aA|$ . 所以  $height[rk[i - 1]] - 1$  就是 *A* 的长度.

由于  $suf[i] = AD$ , 且  $suf[sa[rk[i - 1] - 1] + 1] = AB$ ,  $AB < AD$ , 所以  $suf[i]$  跟  $suf[rk[i] - 1]$  的最长公共前缀至少是 *A*. 即  $height[rk[i]] \geq |A| = height[rk[i - 1]] - 1$

借此结论, 有  $O(n)$  的方法计算 *height* 数组:

```
1  std::vector<int> ComputeHeight(const std::string &s,
2                                const std::vector<int> &sa) {
3      int n = static_cast<int>(s.size());
4      std::vector<int> rk(n);
5      for (int i = 0; i < n; ++i) {
6          rk[sa[i]] = i;
7      }
8      std::vector<int> height(n, 0);
9      for (int i = 0, k = 0; i < n; ++i) {
10         if (rk[i] == 0) {
11             continue;
12         }
13         if (k > 0) {
14             --k;
15         }
16         while (i + k < n && sa[rk[i] - 1] + k < n &&
17                s[i + k] == s[sa[rk[i] - 1] + k]) {
18             ++k;
19         }
20         height[rk[i]] = k;
21     }
22     return height;
23 }
```

---