

后缀数组

MengChunlei

February 27, 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
- 应用

1 定义

对于一个长度为 n 的字符串 $s : s[0], s[1], \dots, s[n-2], s[n-1]$, 它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 个后缀中, 排第 i 名的后缀是哪个 (第一个字母在 s 中的位置). 比如 $s = \text{aabaaaab}$, 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: aaaab

第 1 名: aaab

第 2 名: aab

第 3 名: aabaaaab

第 4 名: ab

第 5 名: abaaaab

第 6 名: b

第 7 名: baaaab

另外还有一个数组称之为 rank 数组 rk , 它的长度也是 n , $rk[i]$ 表示以 $s[i]$ 开始的后缀的排名. 那么有以下关系: $sa[rk[i]] = rk[sa[i]] = i$

2 一般做法

很容易想到 $n^2 \log(n)$ 的做法. 代码如下:

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.

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 $O(n + C)$. 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 }

```

由 14-16 行可以看出, 这是一个稳定排序算法.

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 $n \log^2(n)$ 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.

Listing 5: Last Improvement

```

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);
5      std::vector<int> rk(n);
6      std::vector<int> cnt(m, 0);
7      for (int i = 0; i < n; ++i) {
8          ++cnt[rk[i] = s[i]];
9      }
10     for (int i = 1; i < m; ++i) {
11         cnt[i] += cnt[i - 1];
12     }
13     for (int i = n - 1; i >= 0; --i) {
14         sa[--cnt[rk[i]]] = i;
15     }
16     for (int w = 1; w < n; w <= 1) {
17         sa = RadixSort<int>(sa, 2, {m, m}, [&](const int &x, int index) {
18             if (index == 0) {

```

```
19         return rk[x];
20     } else {
21         return x + w >= n ? 0 : rk[x + w] + 1;
22     }
23 });
24 rk = UpdateRank(sa, rk, n, w);
25 }
26 return sa;
27 }
```

这个算法的复杂度为 $O(n\log(n))$