后缀数组

MengChunlei

February 27, 2021

1 定义

对于一个长度为 n 的字符串 $s: s[0], s[1], ..., s[n-2], s[n-1]$, 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=aabaaaab, 那么对应的 sa 为:

$index$	0	1	2	3	4	5	6	7
s	a	a	b	a	a	a	a	b
sa	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 $n^2log(n)$ 的做法. 代码如下:

Listing 1: Normal

```
std::vector<int> ComputeSA1(const std::string &s) {
  int n = static_cast<int>(s.size());
  std::vector<int> sa(n);
  for (int i = 0; i < n; ++i) {
    sa[i] = i;
  }
  std::sort(sa.begin(), sa.end(), [&](int p1, int p2) {
    for (; p1 < n && p2 < n && s[p1] == s[p2]; ++p1, ++p2);
    return p1 < n && p2 < n ? s[p1] < s[p2] : p1 == n;
  });
  return sa;
}
```

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 $s[i] \to s[i+w-1]$. 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

```
std::vector<int> ComputeSA2(const std::string &s) {
  int n = static_cast<int>(s.size());
  std::vector<int> sa(n);
  std::vector<int> rk(n);
  for (int i = 0; i < n; ++i) {
    sa[i] = i;
    rk[i] = s[i];
  }
  for (int w = 1; w < n; w <<= 1) {
    std::sort(sa.begin(), sa.end(), [&](int x, int y) {
      return rk[x] == rk[y]
                 ? (x + w < n && y + w < n ? rk[x + w] < rk[y + w] : x + w >= n)
                 : rk[x] < rk[y];
    });
    std::vector<int> rk_tmp = rk;
    rk[sa[0]] = 0;
    for (int p = 0, i = 1; i < n; ++i) {
      if (rk_tmp[sa[i]] == rk_tmp[sa[i - 1]] &&
          ((sa[i] + w >= n && sa[i - 1] + w >= n) ||
           (sa[i] + w < n && sa[i - 1] + w < n &&
            rk_tmp[sa[i] + w] == rk_tmp[sa[i - 1] + w]))) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }
    }
  }
  return sa;
}
```

这个算法的复杂度为 $O(nlog^2(n))$. 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:

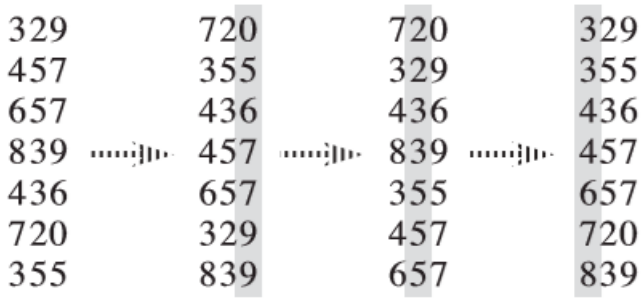
Listing 3: Counting Sort

```
template <typename Elem>
std::vector<Elem> CountingSort(const std::vector<Elem> &eles,
                               std::function<int(const Elem &)> functor,
                               int C) {
  int n = static_cast<int>(eles.size());
  std::vector<int> bucket(C, 0);
  for (int i = 0; i < n; ++i) {
    ++bucket[functor(eles[i])];
  }
  for (int i = 1; i < C; ++i) {
    bucket[i] += bucket[i - 1];
  }
  std::vector<Elem> result(n);
  for (int i = n - 1; i >= 0; --i) {
    result[--bucket[functor(eles[i])]] = eles[i];
  }
  return result;
}
```

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:



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 $nlog^2(n)$ 的方法进行进行改进. 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