数据结构和算法实习

郭 炜

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

后缀数组

后缀数组的概念

贵州黄果树瀑布

后缀数组的概念

- 长度为 n 的字符串 S，有 n 个后缀，各不相同
- 对 n 个后缀排序，每个后缀都有个名次。名次从0到 n – 1
- S 对应于一个有 n 个元素的后缀数组 sa (下标从0到 n – 1)
- sa[i] 表示名次为 i 的后缀在 S 中的起始位置（即起始下标，以后简称位置)

后缀数组的概念

```
        012345
```
字符串 **banana** 的后缀数组

sa[0] = 5	a
sa[1] = 3	ana
sa[2] = 1	anana
sa[3] = 0	banana
sa[4] = 4	na
sa[5] = 2	nana

后缀数组的概念

笨办法求后缀数组：

所有后缀排序 o(nlogn)

排序时比较两个字符串 O(n)

总复杂度 n^2logn

要寻求 nlogn的解法

辅助概念 —— j-后缀

- 对字符串 S 的每个后缀，取左边 j 个字符（若后缀长度不足 j，则全取），即得到一个 j-后缀(j-后缀的长度 <= j)。

- 若S长度为 n，则有 n 个 j-后缀，位置从 0 到 n - 1

banana的 1-后缀： b,a,n,a,n,a
banana的 2-后缀： ba,an,na,an,na,a
banana的 4-后缀： bana,anan,nana,ana,na,a

辅助概念 —— 排名和名次

- j-后缀的排名：可并列，相同的j-后缀排名一样，与j-后缀的位置无关

  排名数组 pm^j ：$pm^j[i]$ 表示位置为 i 的 j-后缀的排名

- j-后缀的名次：不可并列，相同的j-后缀，位置靠左的名次在前

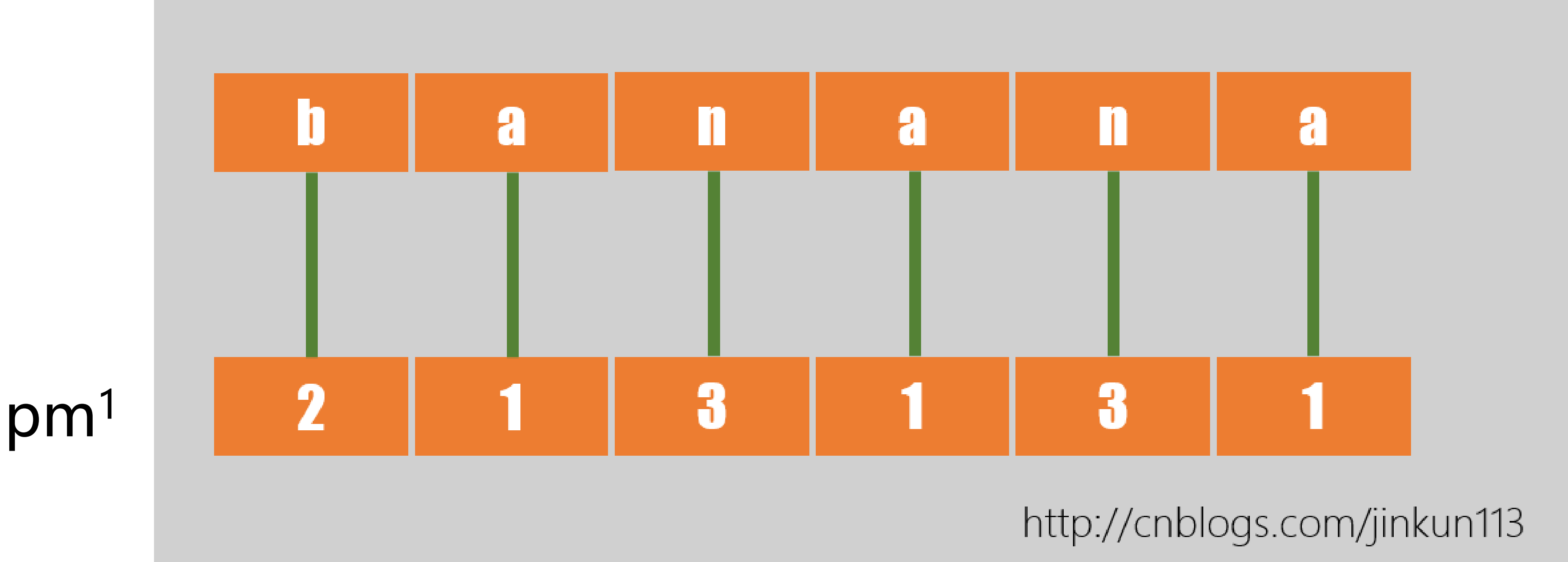
  sa[i]就是名次为 i 的n-后缀的位置 （n是原字符串长度)

倍增法求后缀数组

美国黄石公园

"倍增法"求后缀数组

- 先求所有 1-后缀的排名，得到排名数组 pm^1



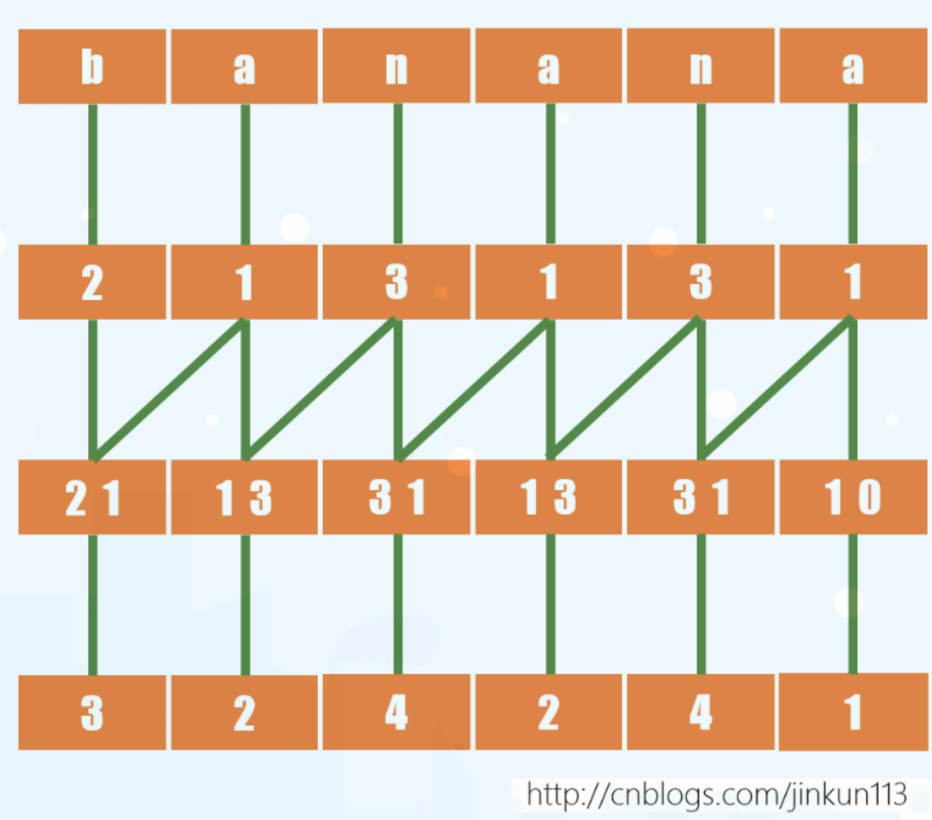
"倍增法"求后缀数组

- 再求所有 2-后缀的排名，得到排名数组 pm^2

  - 每个 2j-后缀，由两个 j-后缀拼成，其相对大小可以用这两个两个j-后缀排名的拼接体来代表
  - 对2j-后缀排名，就是对拼接体排名
  - 对拼接体排名，可以用“基数排序"
  - 由于拼接体是由2个数构成, 基数排序复杂度O(n)

拼接体

pm^2



"倍增法"求后缀数组

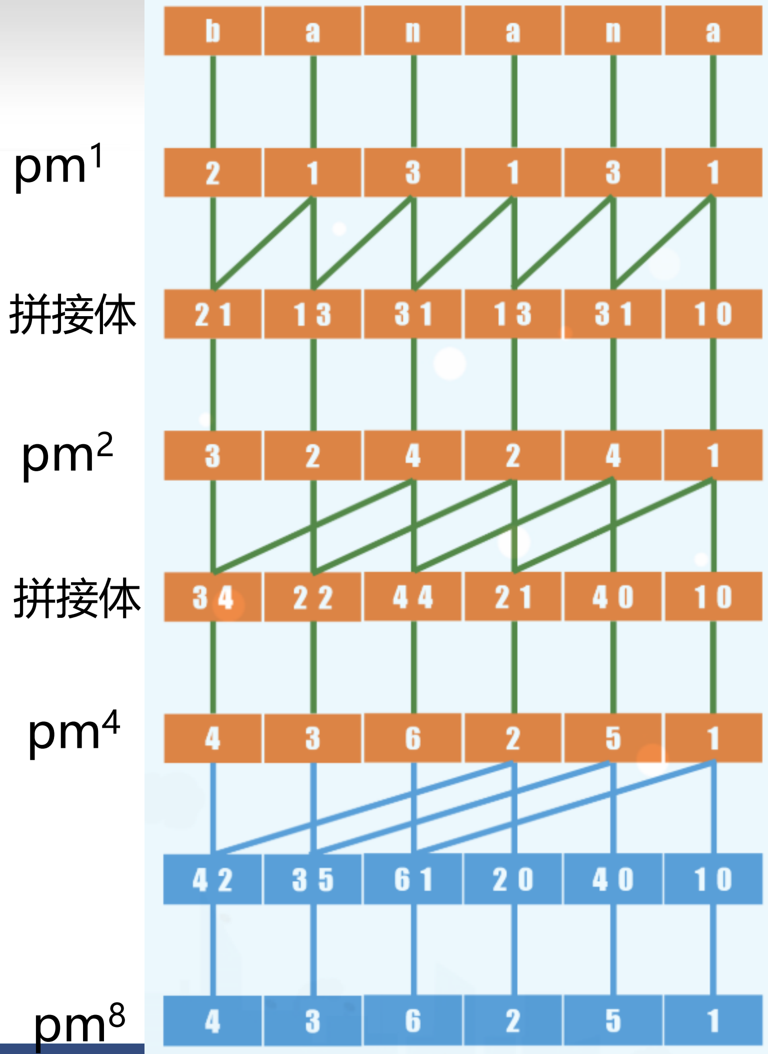
- 再求所有 4-后缀的排名 pm^4 (pm数组一个就够，可以复用)

- 发现pm^4各不相同，即所有的4-后缀都不相同时，就没必要再求 pm^8了。因每个后缀最多只要比前4个字符就能分出胜负，多比没必要，故所有的后缀的排名，就是所有的4-后缀的排名。

- 由最终的pm数组（即pm^4)计算出sa:

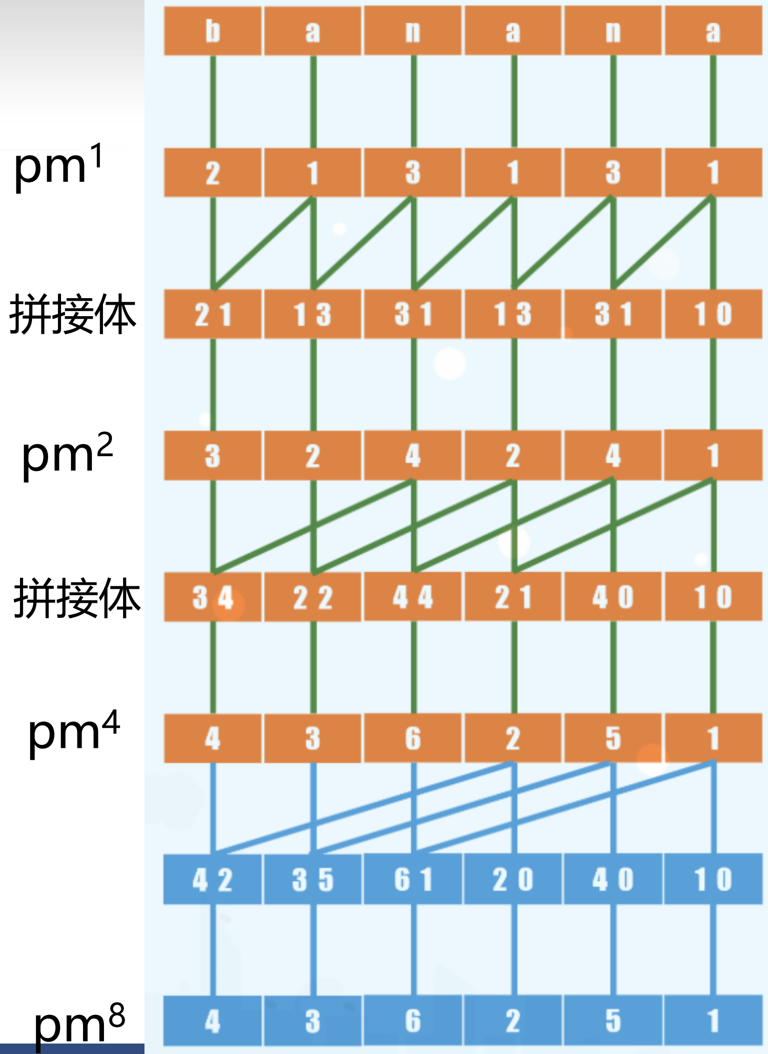
```
for(int i = 0;i < n; ++i)
        sa[pm[i]-1] = i;
```

sa: 5,3,1,0,4,2



"倍增法"求后缀数组的复杂度

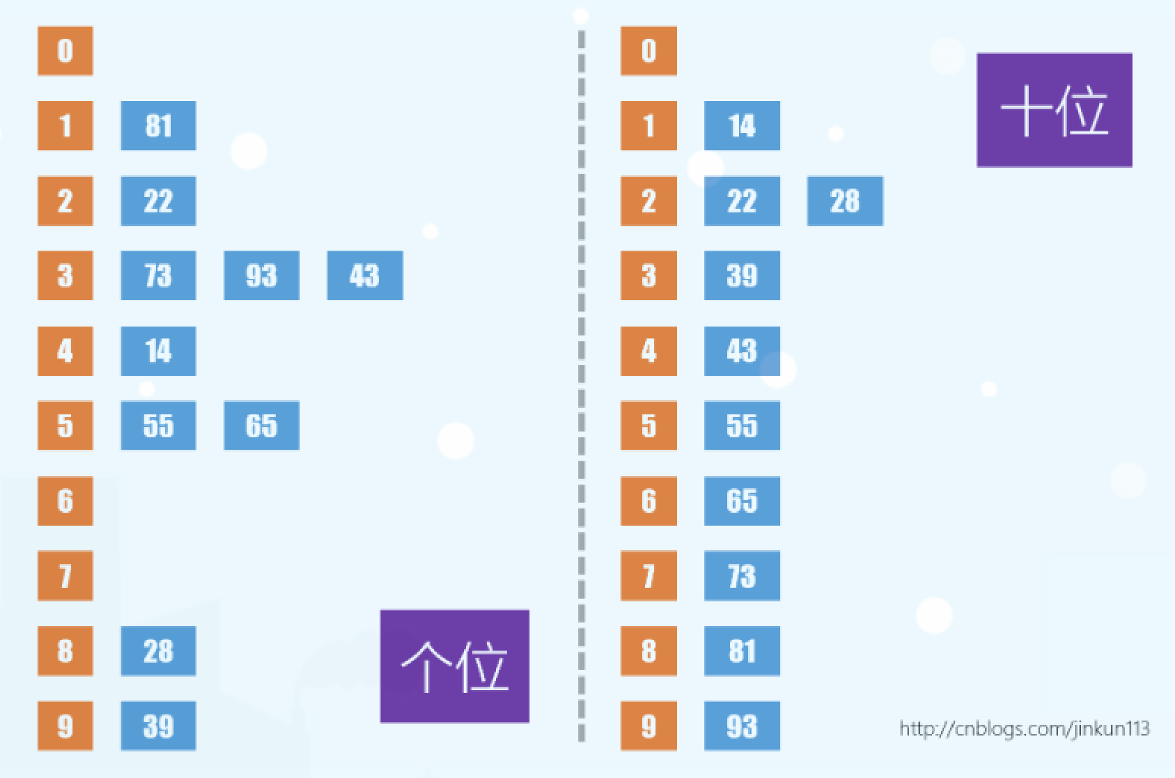
- 设字符串长度为n, 哪怕一直要求到 pm^n, 一共也只要求 logn 轮
- 每轮都做基数排序，复杂度O(n)
- 总复杂度O(nlogn)
- 如果每轮不用基数排序，用快速排序，复杂度 $O(n\log^2 n)$



基数排序(桶排序)

- **对序列 73,22,93,43,55,14,28,65,39,81 排序**

- 根据个位数，将每个数分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列1：
81,22,73,93,43,14,55,65,28,39
- 按顺序将序列1中的每个数，根据十位数，重新分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列2：
14,22,28,39,43,55,65,73,81,93



基数排序(桶排序)

- **基数排序的复杂度是 O(d*(n+radix))**
  n： 要排序的元素的个数(假设每个元素由若干个原子组成)
  radix: 桶的个数，即组成元素的原子的种类数
  d: 元素最多由多少个原子组成

对序列 73,22,93,43,55,14,28,65,39,81 排序:
n = 10, d = 2, radix = 10(或9)

- ➢ 一共要做 d 轮分配和收集
- ➢ 每一轮, 分配的复杂度 O(n)，收集的复杂度O(radix)
  (一个桶里的元素可以用链表存放，便于快速搜集)
- ➢ 总复杂度 O(d * (n + radix))

后缀数组的实现

河北草原天路

```
const int MAXN = 100010;
int wa[MAXN], wb[MAXN], wv[MAXN], Ws[MAXN]; //辅助数组
//注意，wv和Ws的元素个数应该同时超过字符串的字符种类数和字符串的长度
int sa[MAXN]; //sa[i]是名次为i的后缀的位置
void BuildSA(const char * s, int sa[], int n, int m)  {
		int i, j, p, *pm = wa, *k2sa = wb, *t;
		for (i = 0; i<m; i++) Ws[i] = 0;
		for (i = 0; i<n; i++) Ws[pm[i] = s[i]]++;
		for (i = 1; i<m; i++) Ws[i] += Ws[i - 1];
		for (i = n - 1; i >= 0; i--) sa[--Ws[pm[i]]] = i;
		for (j = p = 1; p < n; j <<= 1, m = p)   {
				for (p = 0, i = n - j; i < n; i++) k2sa[p++] = i;
				for (i = 0; i < n; i++) if (sa[i] >= j) k2sa[p++] = sa[i] - j;
				for (i = 0; i < m; i++) Ws[i] = 0;
				for (i = 0; i < n; i++) Ws[wv[i] = pm[k2sa[i]]]++;
				for (i = 1; i < m; i++) Ws[i] += Ws[i - 1];
				for (i = n - 1; i >= 0; i--) sa[--Ws[wv[i]]] = k2sa[i];
```

```
            for (t = pm, pm = k2sa, k2sa = t, pm[sa[0]] = 0, p = i = 1; i<n; i++) {
                int a = sa[i - 1], b = sa[i];
                if (k2sa[a] == k2sa[b] && k2sa[a + j] == k2sa[b + j])
                    pm[sa[i]] = p - 1;
                else
                    pm[sa[i]] = p++;
            }
        }
        return;
}
```

辅助概念 —— 第一关键字和第二关键字

- 每个 2j-后缀，由两个 j-后缀拼成

  这两个 j-后缀分别称为该 2j-后缀的“第一关键字”和“第二关键字"。长度 <= j 的2j后缀，第二关键字为 NULL

  banana的 4-后缀： bana,anan,nana,ana,na,a
  4-后缀 bana 的第一关键字： ba 第二关键字： na
  4-后缀 ana 的第一关键字： an 第二关键字： a
  4-后缀 na 的第一关键字： na 第二关键字： NULL
  4-后缀 a 的第一关键字： a 第二关键字： NULL

辅助概念

- 排名数组 pm^j ： $pm^j[i]$ 是位置为 i 的 j-后缀的排名(排名可并列)

  相同的 j-后缀，排名相同

- 名次数组sa^j : $sa^j[i]$ 是名次为 i 的 j-后缀的位置(名次不可并列)

  相同的 j-后缀，位置靠左的名次在前

pm^j 和sa^j 后都可以重复使用(例如pm^{2j} 可覆盖pm^j)，因此实际上只需要一个pm数组和一个 sa 数组

辅助概念

- 第二关键字名次数组 $k2sa^j$：

$k2sa^j[i]$ 是所有2j-后缀按照第二关键字（第二个j-后缀）排序后，名次为 i 的 2j-后缀的起始位置

$k2sa^j$ 同样可复用，因此只需要一个k2sa数组

倍增算法求后缀数组的程序实现

```
const int MAXN = 100010;
int wa[MAXN], wb[MAXN], wv[MAXN], Ws[MAXN]; //辅助数组
//注意，wv和Ws的元素个数应该同时超过字符串的字符种类数和字符串长度
int sa[MAXN]; //sa[i]是名次为i的后缀的位置，即后缀数组
void buildSA(const char * s, int * sa, int n, int m)
{     //n: 字符串s的长度
      //m: 开始是字符串s中字符的种类数，后来变成不同j-后缀的个数
      //调用方法： buildSA("banana",sa,6, 127);
      int i, j, p, *pm = wa, *k2sa = wb, *t;
      for (i = 0; i<m; i++) Ws[i] = 0;
      for (i = 0; i<n; i++) Ws[pm[i] = s[i]]++; //(1)
  //此时Ws[i]是字符i(编码为i的字符)出现的次数 ，pm是原字符串s的复制品
  //字符的编码就是字符的排名，此处排名可不连续，相对大小正确即可
  //Ws[i]也可以看作排名为i的1-后缀的出现次数
      for (i = 1; i<m; i++) Ws[i] += Ws[i - 1];
      //此时Ws[i]是编码不大于i的字符出现的次数，
      //也可以说是排名不大于 i 的字符出现的次数
```

```
for (i = n - 1; i >= 0; i--) sa[--Ws[pm[i]]] = i;  //(2)
//循环中  i 代表位置
//将名次未确定的字符（1-后缀)称为 U字符
//此时sa[k]是名次为k的1-后缀的位置
//下标为i的那个字符，名次是 Ws[pm[i]]-1。因为:
//下标为i的那个字符，就是pm[i]。现在，有Ws[pm[i]]个U字符排名不大于pm[i]
//即名次不大于pm[i]
//那么pm[i]显然是这些字符里面排名最大的
//虽然可能有多个字符和pm[i]相同，但由于i是从大到小遍历的
//pm[i]就是所有和pm[i]相同的U字符里，位置最大的，也就是名次最大的
//那么pm[i]的名次就是 Ws[pm[i]] - 1 （名次从0开始算）
//--Ws[pm[i]]是因为新确定了一个字符的名字，U字符的个数就要减一
```

```
for (j = p = 1; p < n; j <<= 1, m = p) {   //烧脑循环
 //在一次循环中,已知j-后缀相关信息,要求 2j-后缀相关信息
 //此时，sa[i]是名次为i的j-后缀的位置,pm[i]是位置为i的j-后缀的排名
 // j > 1时，m 是不同j-后缀的个数
        for (p = 0, i = n - j; i < n; i++) k2sa[p++] = i;
//k2sa[i]是所有2j-后缀按照第二关键字（第二个j-后缀）排序后，名次为i的2j-后缀的位置
        //从位置 n-j开始的2j-后缀，第二关键字为NULL，NULL的排名最小，
        //共j个2j-后缀的第二关键字是NULL
        //执行完此循环后，p = j, k2sa[0] - k2sa[j-1]值确定
        for (i = 0; i < n; i++) //按名次从小到大遍历n个j-后缀
            if (sa[i] >= j) k2sa[p++] = sa[i] - j;
//剩余的n-j个2j-后缀，都是第二关键字不为NULL的，第二关键字都是一个j-后缀
//所有第二关键字的位置，自然都是 >= j 的 ，
//位置为 x (x>=j)的每个j-后缀，都会是一个2j-后缀的第二关键字
//若名次为p的第二关键字xx的位置是sa[i],则xx所属的2j-后缀的位置，就是sa[i] - j;执行完此循环，p = n
```

```
for (i = 0; i < m; i++) Ws[i] = 0;
for (i = 0; i < n; i++)
    //按第二关键字名次遍历2j-后缀的第一关键字
    Ws[ wv[i] = pm[k2sa[i]] ]++;
//此时Ws[i]表示排名为i的j-后缀的个数
//j = 1时，Ws和 （1）处的 Ws是一样的
//pm[k2sa[i]]:  位置为k2sa[i]的j-后缀的排名
//Ws[i], 排名为 i 的 j-后缀的个数(j>1时，i从0到m-1)
//wv[i]: 位置为k2sa[i]的j-后缀的排名,第一次执行的时候，和s[k2sa[i]]的编码一样
for (i = 1; i < m; i++) Ws[i] += Ws[i - 1];
//此时 Ws[i]是排名不大于i的j-后缀的个数
```

```
for (i = n - 1; i >= 0; i--)
    //按第二关键字的名次从大到小遍历所有2j-后缀
    sa[--Ws[wv[i]]] = k2sa[i];//求位置为k2sa[i]的2j-后缀的名次
```

//k2sa[i]是所有2j-后缀按照第二关键字排序后，名次为i的 2j-后缀的位置

//位置为 k2sa[i] 的那个U 2j-后缀(简称xx)名次是多少呢?

//wv[i]是xx的第一关键字的排名

//Ws[wv[i]]: 排名不大于wv[i]的j-后缀的个数(每个j-后缀都对应一个2j-后缀)

//比较2j-后缀时，先比较第一关键字，再比较第二关键字

//此处可以看作，所有第一关键字相同的2j-后缀，排名都相同，但是第二关键字名次大的，名次应该更大

//看这些 2j-后缀时，是按第二关键字名次从大到小看的，此处第二关键字的名次，类似于 (2) 处的下标i, xx是排名不大于wv[i]的U 2j-后缀中名次最大的

//循环结束后:

//sa[i]是名次为i的2j-后缀的位置

//pm[i]一直是位置为i的j-后缀的排名，没变化

```
        //下面要把pm[i]变成位置为i的2j-后缀的排名，排名从0开始算
        //下面要把p变成不同的2j-后缀的个数
        for (t = pm, pm = k2sa, k2sa = t,
              pm[sa[0]] = 0, p = i = 1; i<n; i++) {//按名次遍历2j-后缀
              //p为目前发现的，不同的 2j-后缀的个数
              int a = sa[i - 1], b = sa[i];
              //a,b是名次相邻的两个2j-后缀的位置
              //pm,k2sa换了，所以此时k2sa[i]是位置为i的j-后缀的排名
              if (k2sa[a] == k2sa[b] && a + j < n && b + j < n &&
                          k2sa[a + j] == k2sa[b + j])
//看名次为i的那个2j-后缀是不是新的，即和名次为i-1的那个2j-后缀是否一样
//如果两个2j-后缀，他们的第一关键字排名相同（即第一关键字相同）
//且第二关键字排名也相同（即第二关键字相同），则这俩个2j-后缀相同
                    pm[sa[i]] = p - 1; //未发现新的2j-后缀
//位置为sa[i](即排名为i的那个2j-后缀的位置)的2j-后缀的排名是 p - 1
              else
                    pm[sa[i]] = p++; //发现新的2j-后缀
        } //当p达到n时，说明已经有了n个不同的2j-后缀，并且都在sa里排好了序。
          //因此p达到n时，循环即可终止。
    } //烧脑循环结束
    return;  }
```

后缀数组应用

- 在本来就要求母串后缀数组 sa的情况下，求出sa后，可以顺便用sa做模式串匹配

  设母串S长度为 n,模式串长度为m，则模式匹配复杂度: O(mlogn)

- 做法：根据sa，用二分的办法让模式串和S的后缀的前m个字符进行匹配。

  模式串和名次为k的后缀匹配，如果因模式串小而失败，则找个名次更小的后缀试试，如果因模式大而失败， 则找个名次更大的后缀试试.....

  一共需要比较 logn个后缀。每次比较复杂度O(m)

后缀数组应用

To be continued……

RMQ问题

倍增思想另一应用：RMQ问题

- RMQ（Range Minimum/Maximum Query），区间最值查询

  对于长度为n的数列A，回答若干询问RMQ（A,i,j）(i,j<=n)，返回数列A[i...j] 这一段的最小/大值。

- 线段树处理：建好线段树 (O(n))后，查询后复杂度O(logn)

- 倍增算法：预处理(O(nlogn))后，查询复杂度O(1), 更好写

倍增思想另一应用：RMQ问题

- 动态规划预处理：生成二维数组 dp

  dp[i][j] 表示: 从A[i]开始的 2^j 个元素中的最大值

则:

```
dp[i][0] = A[i]      (i= 0…n-1)
```

倍增思想另一应用：RMQ问题

```
dp[i][j]=
    max(dp[i][j-1], dp[i+(1 << (j-1))][j-1])
```

i 的取值范围： n
j 的取值范围: logn

故预处理复杂度 nlogn

倍增思想另一应用：RMQ问题

- 求 A[i...j] 的最大值

  称 dp[i][j] "管辖" 从 A[i] 开始的2^j个元素
  则以下2个dp元素的管辖范围(可能重叠)叠加，正好是 A[i...j]:

  dp[i][x]： 2^x 尽可能大且 $i + 2^x - 1 <= j$ ，管辖A[i... $i + 2^x - 1$]
  管辖范围 > A[i...j]的一半
  dp[k][x] : $k = j - 2^x + 1$,管辖A[k...j]

  故： max(A[i...j]) = max(dp[i][x], dp[k][x])

倍增思想另一应用：RMQ问题

- RMQ的局限性是A的元素不可修改