后缀数组(SA)

一些约定

字符串相关的定义请参考字符串基础

字符串下标从 1 开始。

“ 后缀 i”代指以第 i 个字符开头的后缀。

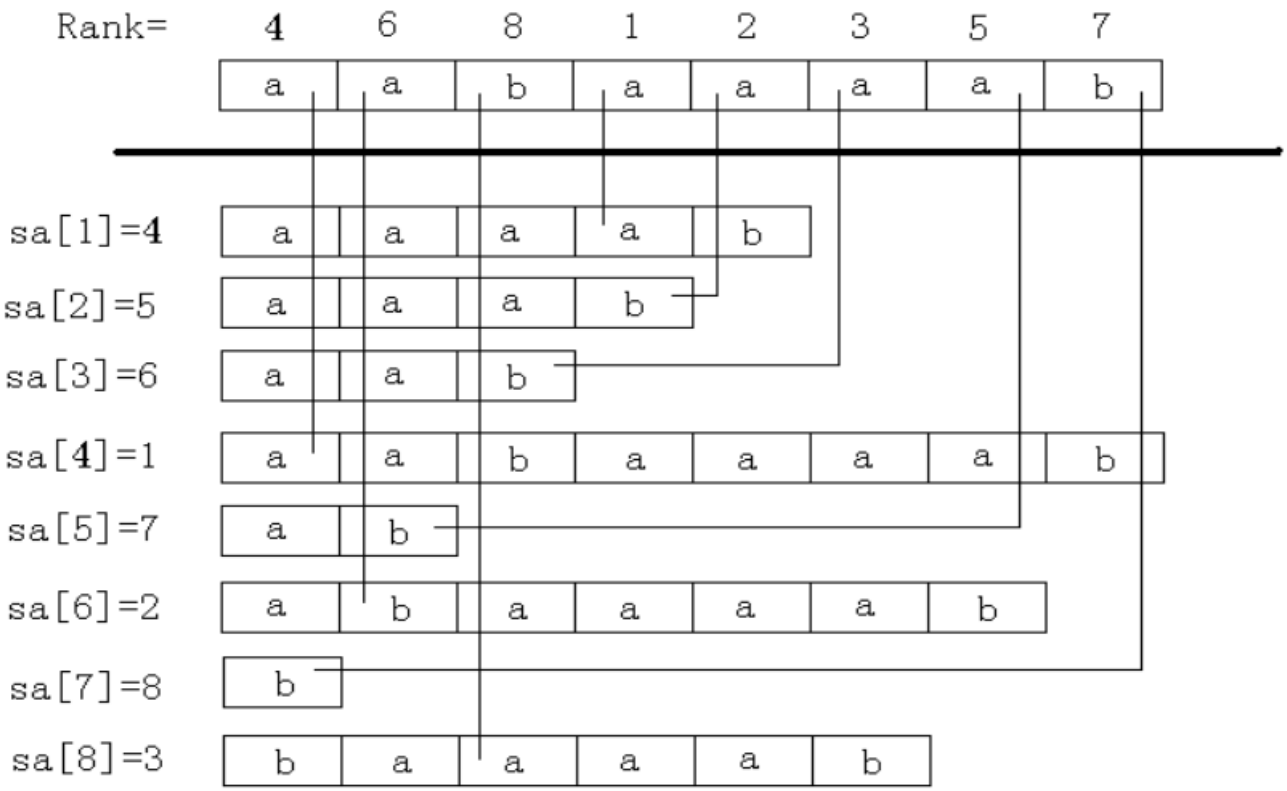
后缀数组是什么？

后缀数组(Suffix Array)主要是两个数组□sa 和 rk□

其中□$sa[i]$ 表示将所有后缀排序后第 i 小的后缀的编号□$rk[i]$ 表示后缀 i 的排名。

这两个数组满足性质□$sa[rk[i]]=rk[sa[i]]=i$□

后缀数组示例：



后缀数组怎么求？

$O(n^2\log n)$ 做法

我相信这个做法大家还是能自己想到的，用 `string` + `sort` 就可以了。由于比较两个字符串是 $O(n)$ 的，所以排序是 $O(n^2\log n)$ 的。

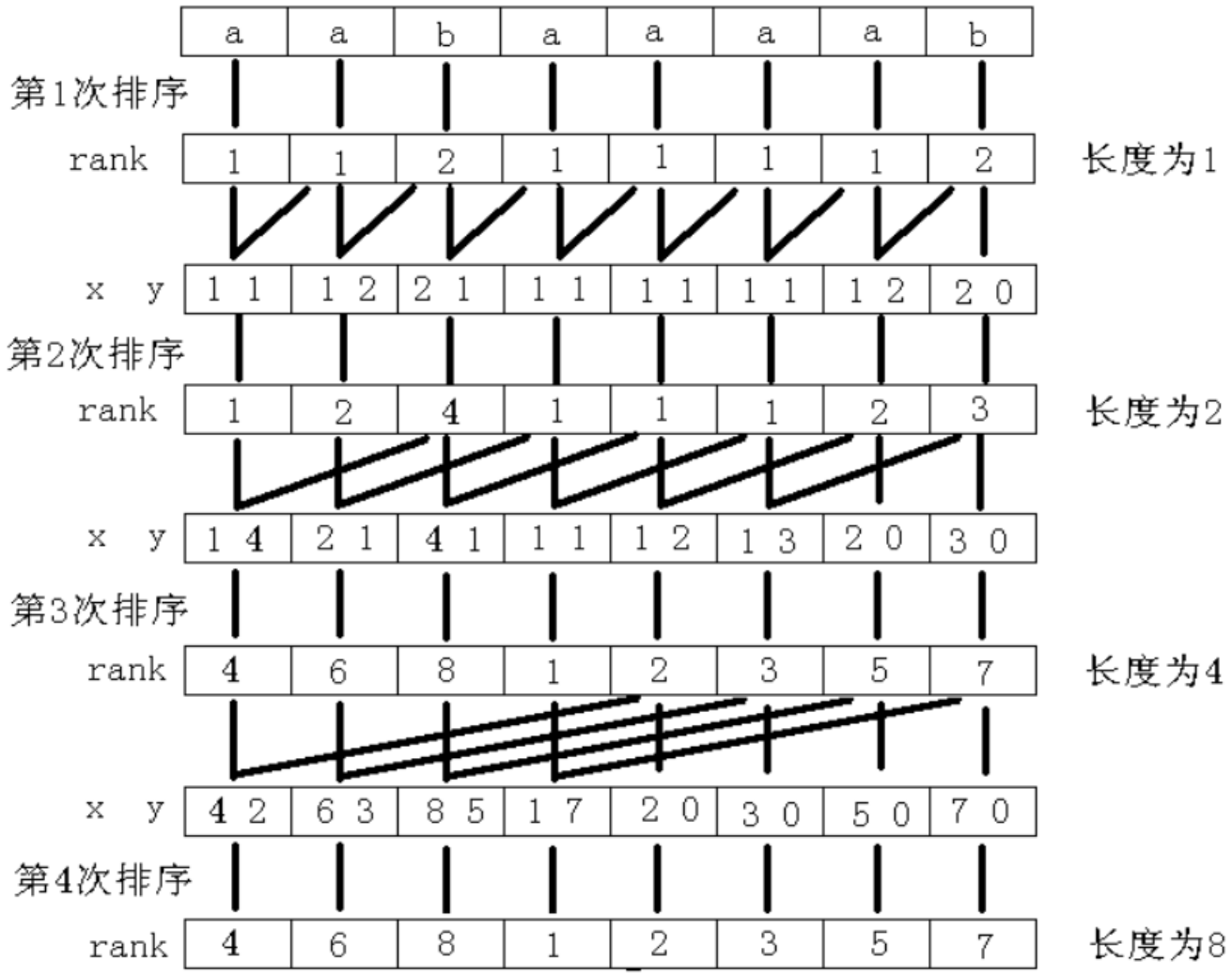
$O(n\log^2n)$ 做法

这个做法要用到倍增的思想。

先对每个长度为 1 的子串（即每个字符）进行排序。

假设我们已经知道了长度为 w 的子串的排名 $rk_w[1..n]$ ，即，$rk_w[i]$ 表示 $s[i..\min(i+w-1,n)]$ 在 $\{s[x..\min(x+w-1,n)]|x\in[1,n]\}$ 中的排名），那么，以 $rk_w[i]$ 为第一关键字，$rk_w[i+w]$ 为第二关键字（若 $i+w>n$ 则令 $rk_w[i+w]$ 为无穷小）进行排序，就可以求出 $rk_{2w}[1..n]$ 。

倍增排序示意图：



如果用 `sort` 进行排序，复杂度就是 $O(n\log^2n)$的。

参考代码：

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, w, sa[N], rk[N << 1], oldrk[N << 1];
//为了防止访问 rk[i+w] 导致数组越界，开两倍数组。
//当然也可以在访问前判断是否越界，但直接开两倍数组方便一些。

int main() {
  int i, p;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) sa[i] = i, rk[i] = s[i];

  for (w = 1; w < n; w <<= 1) {
    sort(sa + 1, sa + n + 1, [](int x, int y) {
      return rk[x] == rk[y] ? rk[x + w] < rk[y + w] : rk[x] < rk[y];
    });  //这里用到了 lambda
    memcpy(oldrk, rk, sizeof(rk));
    //由于计算 rk 的时候原来的 rk 会被覆盖，要先复制一份
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }  //若两个子串相同，它们对应的 rk 也需要相同，所以要去重
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

$O(n\log n)$ 做法

在刚刚的 $O(n\log^2n)$ 做法中，单次排序是 $O(n\log n)$ 的，如果能 $O(n)$ 排序，就能在 $O(n\log n)$ 计算后缀数组了。

前置知识：计数排序，基数排序。

由于计算后缀数组的过程中排序的关键字是排名，值域为 $O(n)$并且是一个双关键字的排序，可以使用基数排序优化至 $O(n)$

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];

int main() {
  int i, m, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  m = max(n, 300);
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1) {
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) id[i] = sa[i];
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) id[i] = sa[i];
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }
    }
  }
```

```
  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

一些常数优化

如果你把上面那份代码交到 LOJ #111: 后缀排序 上：

测试点 #1	Accepted	得分：100	用时：20 ms	内存：11900 KiB
测试点 #2	Accepted	得分：100	用时：27 ms	内存：12004 KiB
测试点 #3	Accepted	得分：100	用时：27 ms	内存：11900 KiB
测试点 #4	Accepted	得分：100	用时：37 ms	内存：12112 KiB
测试点 #5	Accepted	得分：100	用时：36 ms	内存：12112 KiB
测试点 #6	Accepted	得分：100	用时：103 ms	内存：13200 KiB
测试点 #7	Accepted	得分：100	用时：111 ms	内存：13248 KiB
测试点 #8	Time Limit Exceeded	得分：0	用时：4018 ms	内存：24576 KiB
测试点 #9	Time Limit Exceeded	得分：0	用时：4014 ms	内存：24576 KiB
测试点 #10	Time Limit Exceeded	得分：0	用时：4018 ms	内存：24576 KiB

这是因为，上面那份代码的常数的确很大。

第二关键字无需计数排序

实际上，像这样就可以了：

```
for (p = 0, i = n; i > n - w; --i) id[++p] = i;
for (i = 1; i <= n; ++i) {
  if (sa[i] > w) id[++p] = sa[i] - w;
}
```

意会一下，先把 $s[i+w..i+2w-1]$ 为空串（即第二关键字为无穷小）的位置放前面，再把剩下的按排好的顺序放进去。

优化计数排序的值域

每次对 rk 进行去重之后，我们都计算了一个 p□这个 p 即是 rk 的值域，将值域改成它即可。

将 rk[id[i]] 存下来，减少不连续内存访问

这个在数据范围较大时效果非常明显。

用函数 cmp 来计算是否重复

同样是减少不连续内存访问，在数据范围较大时效果比较明显。

把 `oldrk[sa[i]] == oldrk[sa[i - 1]] && oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]` 替换成 `cmp(sa[i], sa[i - 1], w)`，`bool cmp(int x, int y, int w) { return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w]; }`。

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], rk[N], oldrk[N << 1], id[N], px[N], cnt[N];
// px[i] = rk[id[i]]，用于排序的数组所以叫 px。

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, m = 300, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {  // m=p 就是优化计数排序值域
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);
```

```
  return 0;
}
```

$O(n)$ 做法

在一般的题目中，常数较小的倍增求后缀数组是完全够用的，求后缀数组以外的部分也经常有 $O(n\log n)$ 的复杂度，倍增求解后缀数组不会成为瓶颈。

但如果遇到特殊题目、时限较紧的题目，或者是你想追求更短的用时，就需要学习 $O(n)$ 求后缀数组的方法。

SA-IS

可以参考 诱导排序与 SA-IS 算法

DC3

可以参考 2009 后缀数组——处理字符串的有力工具 by. 罗穗骞

后缀数组的应用

寻找最小的循环移动位置

将字符串 S 复制一份变成 SS 就转化成了后缀排序问题。

例题： JSOI2007字符加密

在字符串中找子串

任务是在线地在主串 T 中寻找模式串 S在线的意思是，我们已经预先知道主串 T但是当且仅当询问时才知道模式串 S我们可以先构造出 T 的后缀数组，然后查找子串 S若子串 S 在 T 中出现，它必定是 T 的一些后缀的前缀。因为我们已经将所有后缀排序了，我们可以通过在 sa 数组中二分查找来实现。比较子串 S 和当前后缀的时间复杂度为 $O(|S|)$因此找子串的时间复杂度为 $O(|S|\log |T|)$注意，如果该子串在 T 中出现了多次，每次出现都是在 sa 数组中相邻的。因此出现次数可以通过再次二分找到，输出每次出现的位置也很轻松。

从字符串首尾取字符最小化字典序

例题：USACO07DECBest Cow Line

题意：给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中最小的一个。

> 题解：暴力做法就是每次最坏 $O(n)$ 地判断当前应该取首还是尾（即比较取首得到的字符串与取尾得到的反串的大小），只需优化这一判断过程即可。
>
> 由于需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如 #，代码中可以直接使用空字符），求后缀数组，即可 $O(1)$ 完成这一判断。

参考代码：

```
<hidden>
#include <cctype>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], id[N], oldrk[N << 1], rk[N << 1], px[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, w, m = 200, p, l = 1, r, tot = 0;

  cin >> n;
  r = n;

  for (i = 1; i <= n; ++i)
    while (!isalpha(s[i] = getchar()))
      ;
  for (i = 1; i <= n; ++i) rk[i] = rk[2 * n + 2 - i] = s[i];

  n = 2 * n + 1;

  for (i = 1; i <= n; ++i) ++cnt[rk[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
```

```
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }

  while (l <= r) {
    printf("%c", rk[l] < rk[n + 1 - r] ? s[l++] : s[r--]);
    if ((++tot) % 80 == 0) puts("");
  }

  return 0;
}
```

</hidden>

height 数组

LCP（最长公共前缀）

两个字符串 S 和 T 的 LCP 就是最大的 x ($x\le\min(|S|,|T|)$) 使得 $S_i=T_i(\forall1\le i\le x)$

下文中以 $lcp(i,j)$ 表示后缀 i 和后缀 j 的最长公共前缀（的长度）。

height 数组的定义

$height[i]=lcp(sa[i],sa[i-1])$即第 i 名的后缀与它前一名的后缀的最长公共前缀。

$height[1]$ 可以视作 0。

O(n) 求 height 数组需要的一个引理

$height[rk[i]]\ge height[rk[i-1]]-1$

证明：

略

O(n) 求 height 数组的代码实现

利用上面这个引理暴力求即可：

```
for (i = 1, k = 0; i <= n; ++i) {
  if (k) --k;
  while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;
  ht[rk[i]] = k;  // height太长了缩写为ht
```

```
}
```

k 不会超过 n最多减 n 次，所以最多加 $2n$ 次，总复杂度就是 $O(n)$

未完待续

参考链接

参考链接