第4章 字符串

> Talk is cheap, show me the code. -- Linus Torvalds

4.1

> 假设以链表结构 LString（定义如下）作为串的存储结构，试编写判别给定串 S 是否具有对称性的算法，并要求算法的时间复杂度为 O(StrLength(S))。
>
> ```
> struct ListNode
> {
>   Char data; //存放数据;
>   ListNode * next; //存放指向后继结点的指针;
> };
>
> typedef ListNode * ListPtr;
>
> struct LString {
>   ListPtr head; //链表的表头指针
>   int strLen; //串的长度
> }
> ```

注：题目里用的是“`Char data`”，另外在 `struct Lstring{...}` 之后忘记打分号 `;` 了，所以没法过编译。

答：判断一个长度为n的字符串s是否为回文串，相当于判断是否【长度为 $\lfloor n/2 \rfloor$ 的前缀】与【长度为 $\lfloor n/2 \rfloor$ 的后缀的翻转】相等即可，即判断是否有`s[0:n/2] == reversed(s[n-n/2:n])`：

```
ListPtr reverseList(ListPtr head) { // O(1)空间，O(n)时间的单向链表翻转算法
  ListPtr prev = nullptr, cur = head;
  while (cur) {
    ListPtr nxt = cur->next;
    cur->next = prev;
    prev = cur;
    cur = nxt;
  }
  return prev;
}

bool isPalindrome(LString s) {
  int n = s.strLen;
  if (n <= 1) return true;
  ListPtr head = s.head;
  for (int i = 0; i < (n-n/2); ++i) { // 找到s[n-n/2]
    head = head->next;
  }
  ListPtr tail = reverseList(head); // 计算reversed(s[n-n/2:n])
  ListPtr tail_backup = tail;
  head = s.head;
```

```
  bool flag = true;
  for (int i = 0; i < n/2; ++i) { // 判断 s[0:n/2] == reversed(s[n-n/2:n])
    if (head->data != tail->data) flag = false;
    head = head->next;
    tail = tail->next;
  }
  reverseList(tail_backup); // 复原
  return flag;
}
```

在修改过题目中的错误之后，你可以使用

```
char s[100];

int main() {
    scanf("%s", s);
    int n = strlen(s);
    LString t;
    t.strLen = n;
    ListPtr *cur = &t.head;
    for (int i = 0; i < n; ++i) {
        *cur = new ListNode();
        (*cur)->data = s[i];
        cur = &((*cur)->next);
    }
    printf("%d", isPalindrome(t));
}
```

来验证以上代码的正确性。

4.2

> 写出一个线性时间的算法，判断字符串 **T** 是否是另一个字符串 **T'** 的循环旋转。例如 **arc** 和 **car** 是彼此的循环旋转。

答：**T** 是 **T'** 的循环旋转等价于 **T'** 是 **T + T** 的某个字串，故只需要进行一次字符串匹配即可。

```
#include <vector>
#include <string>

using namespace std;

void init(const string &s, vector<int> &nxt, int n) { // 计算nxt数组
    nxt[0] = -1;
    for (int i = 1; i <= n; i++) {
        int j = nxt[i-1];
        while (s[j+1] != s[i] && j != -1) j = nxt[j];
        nxt[i] = j+1;
    }
```

```
}

bool kmp(string s, string t) {
  int n = s.size();
  s = "#" + s;
  vector<int> nxt(n+1);
  init(s, nxt, n);

  int m = t.size();
  t = "#" + t;
  for (int i = 1, j = 0; i <= m; ++i) {
    for (; j != -1 && s[j+1] != t[i] ; j = nxt[j] );
    j++;
    if (j == n) return true;
  }
  return false;
}

bool solve(string t, string t_) {
  t += t;
  return kmp(t_, t);
}
```

你可以加上

```
#include <iostream>

int main() {
    string s, t;
    cin >> s >> t;
    cout << solve(s, t);
}
```

来测试它的正确性。

4.3

> 请证明教材中 **KMP** 数组(优化和非优化两种)算法正确性。

非优化

先从PPT直接抄过来算法（注：这个算法的返回值应该是 `int*` 而不是 `int`，课件里打错了）：

```
int findNext(string P) {
  int j, k;
  int m = P.length(); // m为模式P的长度
  assert(m > 0); // 若m=0，退出
  int *next = new int[m]; // 动态存储区开辟整数数组
```

```
  assert(next != 0); // 若开辟存储区域失败，退出
  next[0] = -1;
  j=0; k=-1;
  while (j < m-1) {
    while (k >= 0 && P[k] != P[j]) // 不等则采用 KMP 自找首尾子串
      k = next[k]; // k 递归地向前找
    j++; k++; next[j] = k;
  }
  return next;
}
```

我们规定 $s[0:i]$ 表示 s 中下标从 0 到 $i-1$ 的元素组成的字串（换句话说就是Python的字符串切片`s[0:i]`）。

首先，考虑next数组的定义：

$$n_i = \max\{0 \le j < i \mid s[0:i].prefix(j) == s[0:i].suffix(j)\}$$

然后，证明对 $i = 1, 2, \dots, m-1$ 均有上式成立。

容易验证，当 $i = 1$ 时，$n_i = 0$，`next[i] == 0`。

假设当 $i \le l$ 时成立 $\textbf{next}[\textbf{i}] == n_i$，下证当 $i = l+1$ 时成立：

1. 在循环开始时，有 `j == l` 与 `k == next[j]` 。
2. 然后便是递归找的过程。
    - 为方便证明，我们设

      $$S_i = \{0 \le j < i \mid s[0:i].prefix(j) == s[0:i].suffix(j)\}$$

      为所有使得字串 $s[0:i]$ 中前缀等于后缀的长度，此时知 $n_i = \max S_i$ 。
    - 考虑 S_i 满足什么性质：
        - 显然，$\forall i > 0, 0 \in S_i$。
        - 我断言 $S_i = S_{n_i} \cup \{n_i\}$，理由如下：
            - $\supseteq$：由定义的前后缀相等性，易证 $S_{n_i} \subseteq S_i$ 与 $n_i \in S_i$。
            - $\subseteq$：由 $n_i \in S_i$ 且由它的最大性，可证 $S_i \backslash \{n_i\} \subseteq S_{n_i}$ ，得证。
    - 另外，若 $n_i \ne 0$ ，由定义可以证明 $n_i - 1 \in S_{i-1}$。
    - 由于 $n_i = \max S_i$ 与 $S_{n_i} = S_i \backslash \{n_i\}$ ，可知不断进行 $i := n_i$ 操作相当于从大到小遍历 S_i 。因此，若 $n_{i+1} \ne 0$，一定可以找到一个 $n_i \in S_i$ 使得 $n_i + 1 \in S_{i+1}$ 。第一个这样的 n_i 即为 $n_{i+1} - 1$。
    - 为方便处理 $n_{i+1} = 0$ 的情况，我们令 $n_0 = -1$。这相当于往所有的 S_i 里加了一个元素 -1 。此时，无论 n_{i+1} 是否为零，都一定可以找到一个 $n_i \in S_i$ 使得 $n_i + 1 \in S_{i+1}$ 。第一个这样的 n_i 即为 $n_{i+1} - 1$。
    - 回到代码中。
        - 因为循环开始时 `k = next[j]` ，知 `k = next[k]` 的过程实际上就是在遍历 S_j 。
        - 因为`P` 是0-下标的，`P[k] != P[j]` 实际上就是在判断第 `k+1` 个字符是否等于第 `j+1` 个字符，即判断是否有 $k+1 \in S_{j+1}$ 。
        - 找到的第一个满足 $k+1 \in S_{j+1}$ 的 $k+1$ 即为 n_{j+1} 。

因此，正确性得证。

优化

先从PPT直接抄过来算法（同上，这个算法的返回值应该是 `int*` 而不是 `int`，课件里打错了）：

```
int findNext(string P) {
  int j, k;
  int m = P.length(); // m为模式P的长度
  int *next = new int[m]; // 动态存储区开辟整数数组
  next[0] = -1;
  j=0; k=-1;
  while (j < m-1) { // 若写成 j < m 会越界
    while (k >= 0 && P[k] != P[j]) // 若不等，采用 KMP 找首尾子串
      k = next[k]; // k 递归地向前找
    j++; k++;
    if (P[k] == P[j])
      next[j] = next[k]; // 前面找 k 值，没有受优化的影响
    else next[j] = k; // 取消if判断，则不优化
  }
  return next;
}
```

这次的 n'_j 的定义变得不一样了，为满足后一项不一样前提下的最大值：

$$n'_i = \max\{0 \le j < i \mid \texttt{s[0:i].prefix(j) == s[0:i].suffix(j) \&\& s[i] != s[j]}\}$$

不过，我们每一轮维护的 `k` 值都是本来的 n_i。

我们对 j 来进行归纳：

若当前 $k = n_j$，在运行完 `k = next[k]` 这一轮之后有 $k+1 = n_{j+1}$：

- 实际上，`k = next[k]` 这个操作类似于从大到小遍历之前定义的集合 S_i，只是跳过了一定对匹配没有帮助的，`P[l] == P[k]` 的 `l`。
- 因此，在循环结束后依然有 $k+1 = n_{j+1}$

接下来是已知 n_j 求 n'_j：

- 假如第 $j+1$ 个字符 `P[j]` 与第 $k+1$ 个字符 `P[k]` 相等，那么这意味着需要跳过 $k \in S_j$。此时，由归纳假设，知 $l = n'_k$ 是跳过了所有满足第 $l+1$ 个元素 `P[l]` 等于第 $k+1$ 个元素 `P[k]` 的 l 之后最大的 $l \in S_k$。因为 `P[k] == p[l]`，所以也有 $n'_j = l = n'_k$。
- 假如第 $j+1$ 个字符 `P[j]` 与第 $k+1$ 个字符 `P[k]` 不等，由定义，$n'_j = k$。

得证。