1

我编写的算法 C++ 代码如下：

```
bool isSymmetric(const LString& str){
    int t = str.strLen/2;
    char arr[t+1];    // VLA

    // move to the second half
    ListPtr p = str.head;
    for (int i = 0; i < t; ++i)
        p = p->next;

    // store the second half in arr
    int idx = 0;
    for (; p; ++idx){
        arr[idx] = p->data;
        p = p->next;
    }

    // char-wise comparison
    p = str.head;
    while (idx >= 0){
        if (p->data != arr[idx])
            return false;
        p = p->next;
        --idx;
    }
    return true;
}
```

这个算法的基本思路是：既然单链表无法倒序遍历，就把链表装进一个数组里，然后再倒序遍历. 于是，我将链表的后半部分存入一个数组，再与前半部分逐字符比较——这是标准的判断对称性的方法.

但是，这种方法虽然时间上是 $O(N)$ 的，却使用了 $O(N)$ 的额外空间. 我不知道如何只使用 $O(1)$ 的空间完成这道题目.

另外，`char arr[t+1];` 在标准 C++ 中是不合法的，我这里使用了 GCC 的 VLA 拓展. 为了避免误会，特此澄清.

2

将 T' 复制一份并接在其结尾，称为 S. 例如，$T' = \text{car}$ 时 $S = \text{carcar}$. 显然，T, T' 互为循环旋转的充分必要条件是 T, T' 长度相等且 T 是 S 的子串. 因此对 S, T 运行 KMP 算法即可得到答案.

由于复制和 KMP 都是线性时间的，因此这个算法总体是线性时间的.

3

next 数组有多种常见的定义，下面只对

$$\mathrm{next}[i] := \max\{0, \max\{0 < j < i \big| \mathrm{p}[1 \sim j] = \mathrm{p}[i-j+1 \sim i]\}\}, 1 \le i \le N$$

进行证明. 其中 p 为模式串，有效下标从 1 到 N.

为了方便叙述，称满足 $\mathrm{p}[1 \sim j] = \mathrm{p}[i-j+1 \sim i]$ 的所有 $0 < j < i$ 为 $\mathrm{next}[i]$ 的候选项. 特别地，也认为 0 是 $\mathrm{next}[i]$ 的侯选项. 显然，$\mathrm{next}[i]$ 是其所有候选项中最大的..

由定义，$\mathrm{next}[1] = 0$. 我们只需考虑如果已经计算出了 $\mathrm{next}[1], \cdots, \mathrm{next}[i-1]$，如何高效计算 $\mathrm{next}[i]$.

首先证明两个引理.

引理

引理 1：设 $0 < j_0 < i$ 为 $\mathrm{next}[i]$ 的一个侯选项，则 $\mathrm{next}[i]$ 的候选项中，小于 j_0 的最大的一个是 $\mathrm{next}[j_0]$.

证明：

记 $k = \mathrm{next}[j_0]$.

首先证明 k 是 $\mathrm{next}[i]$ 的候选项.

事实上，由于 k 是 $\mathrm{next}[j_0]$ 的候选项，有 $\mathrm{p}[1 \sim k] = \mathrm{p}[j_0 - k + 1 \sim j_0]$. 另一方面，由于 j_0 是 $\mathrm{next}[i]$ 的候选项，有 $\mathrm{p}[1 \sim j_0] = \mathrm{p}[i - j_0 + 1 \sim i]$，这隐含着 $\mathrm{p}[j_0 - k + 1 \sim j_0] = \mathrm{p}[i - k + 1 \sim i]$. 于是便有 $\mathrm{p}[1 \sim k] = \mathrm{p}[i - k + 1 \sim i]$，即 k 是 $\mathrm{next}[i]$ 的候选项.

再证明 $\forall k < k_1 < j_0$ 不是 i 的候选项.

反之，有 $\mathrm{p}[1 \sim k_1] = \mathrm{p}[i - k_1 + 1 \sim i]$. 又 $\mathrm{p}[1 \sim j_0] = \mathrm{p}[i - j_0 + 1 \sim i]$ 隐含着 $\mathrm{p}[j_0 - k_1 + 1 \sim j_0] = \mathrm{p}[i - k_1 + 1 \sim i]$，于是有 $\mathrm{p}[1 \sim k_1] = \mathrm{p}[j_0 - k_1 + 1 \sim j_0]$，即 k_1 是 $\mathrm{next}[j_0]$ 的候选项. 但是 $k_1 > \mathrm{next}[j_0]$ ，这便说明了矛盾.

证毕.

引理 2：$1 < j < i$ 是 $\mathrm{next}[i]$ 的候选项的充要条件是 $j-1$ 是 $\mathrm{next}[i-1]$ 的候选项，且$\mathrm{p}[j] = \mathrm{p}[i]$.

这个引理可以由定义直接导出，不再赘述证明.

算法实现

根据引理 1 和引理 2，$\mathrm{next}[i]$ 的候选项只可能是 $\mathrm{next}[i-1]+1, \mathrm{next}[\mathrm{next}[i-1]]+1, \cdots, 0$，称它们按顺序是 $\mathrm{next}[i]$ 的候选候选项.

因此，在计算 $\mathrm{next}[i]$ 时，只需逐个尝试这些候选候选项 j 是否也满足 $\mathrm{p}[j] = \mathrm{p}[i]$，如果满足，那么它就是 $\mathrm{next}[i]$ 的候选项. 由于 $\forall k, \mathrm{next}[k] < k$，第一个满足上述条件的候选候选项一定是 $\mathrm{next}[i]$ 的最大候选项，即 $\mathrm{next}[i]$.

其代码实现如下：

```
void getNext(){
    next[1] = 0;
    for (int i = 2, j = 0; i <= N; ++i){
        while (j && p[i] != p[j+1]) j = next[j];
        if (p[i] == p[j+1]) ++j;
        next[i] = j;
    }
}
```

优化

KMP 算法有一个经典的优化，即如果 $\mathrm{p}[\mathrm{next}[i]+1]=\mathrm{p}[\mathrm{i}+1]$，那么如果匹配时在 $i+1$ 处失配，那回溯后仍然会失配，在计算 next 数组时就排除这种情况，可以避免更多注定失败的回溯. 优化后的 next 数组一般称为 $\mathrm{nextval}$ 数组.

避免这种情况的方法很简单，在更新 $\mathrm{next}[i]$ 时判断上述条件，如果真的相等，就再次回溯, 利用与前一部分证明中完全平行的方法，也可以证明这个优化的正确性，这里不再赘述.

其代码实现如下：

```
void getNextval(){
    next[1] = 0;
    for (int i = 2, j = 0; i <= N; ++i){
        while (j && p[i] != p[j+1]) j = nextval[j];
        if (p[i] == p[j+1]) ++j;
        if (i < N && p[i+1] == p[j+1]) j = nextval[j];
        nextval[i] = j;
    }
}
```

事实上，这个优化的改进并不大，因为即使有冗余的回溯，每次回溯也只是比较了一个字符而已，如果文本串不充分长，优化效果是十分轻微的. 这个优化之所以经典，是因为其与 KMP 原版算法浑然一体，思想高度相似，而且实现几乎没有代价——只是加了一行代码而已.