

## 添加最少字符使字符串整体都是回文字符串

### 【题目】

给定一个字符串 `str`，如果可以在 `str` 的任意位置添加字符，请返回在添加字符最少的情况下，让 `str` 整体都是回文字符串的一种结果。

### 【举例】

`str="ABA"`。`str` 本身就是回文串，不需要添加字符，所以返回 `"ABA"`。

`str="AB"`。可以在 `'A'` 之前添加 `'B'`，使 `str` 整体都是回文串，故可以返回 `"BAB"`。也可以在 `'B'` 之后添加 `'A'`，使 `str` 整体都是回文串，故也可以返回 `"ABA"`。总之，只要添加的字符数最少，只返回其中一种结果即可。

### 【进阶题目】

给定一个字符串 `str`，再给定 `str` 的最长回文子序列字符串 `strlps`，请返回在添加字符最少的情况下，让 `str` 整体都是回文字符串的一种结果。进阶问题比原问题多了一个参数，请做到时间复杂度比原问题的实现低。

### 【举例】

`str="A1B21C"`，`strlps="121"`，返回 `"AC1B2B1CA"` 或者 `"CA1B2B1AC"`。总之，只要是添加的字符数最少，只返回其中一种结果即可。

### 【难度】

校 ★★☆☆

### 【解答】

原问题。在求解原问题之前，我们先来解决下面这个问题，如果可以在 `str` 的任意位置添加字符，最少需要添几个字符可以让 `str` 整体都是回文字符串。这个问题可以用动态规划的方法求解。如果 `str` 的长度为  $N$ ，动态规划表是一个  $N \times N$  的矩阵记为 `dp[][]`。`dp[i][j]` 值的含义代表子串 `str[i..j]` 最少添加几个字符可以使 `str[i..j]` 整体都是回文串。那么，如果求 `dp[i][j]`

的值呢？有如下三种情况：

- 如果字符串  $\text{str}[i..j]$  只有一个字符，此时  $\text{dp}[i][j]=0$ ，这是很明显的，如果  $\text{str}[i..j]$  只有一个字符，那么  $\text{str}[i..j]$  已经是回文串了，自然不必添加任何字符。
- 如果字符串  $\text{str}[i..j]$  只有两个字符。如果两个字符相等，那么  $\text{dp}[i][j]=0$ 。比如，如果  $\text{str}[i..j]$  为 "AA"，两字符相等，说明  $\text{str}[i..j]$  已经是回文串，自然不必添加任何字符。如果两个字符不相等，那么  $\text{dp}[i][j]=1$ 。比如，如果  $\text{str}[i..j]$  为 "AB"，只用添加一个字符就可以令  $\text{str}[i..j]$  变成回文串，所以  $\text{dp}[i][j]=1$ 。
- 如果字符串  $\text{str}[i..j]$  多于两个字符。如果  $\text{str}[i]==\text{str}[j]$ ，那么  $\text{dp}[i][j]=\text{dp}[i+1][j-1]$ 。比如，如果  $\text{str}[i..j]$  为 "A124521A"， $\text{str}[i..j]$  需要添加的字符数与  $\text{str}[i+1..j-1]$ （即 "124521"）需要添加的字符数是相等的，因为只要能把 "124521" 整体变成回文串，然后在左右两头加上字符 'A'，就是  $\text{str}[i..j]$  整体变成回文串的结果。如果  $\text{str}[i] \neq \text{str}[j]$ ，要让  $\text{str}[i..j]$  整体变为回文串有两种方法，一种方法是让  $\text{str}[i..j-1]$  先变成回文串，然后在左边加上字符  $\text{str}[j]$ ，就是  $\text{str}[i..j]$  整体变成回文串的结果。另一种方法是让  $\text{str}[i+1..j]$  先变成回文串，然后在右边加上字符  $\text{str}[i]$ ，就是  $\text{str}[i..j]$  整体变成回文串的结果。两种方法中哪个代价最小就选择哪个，即  $\text{dp}[i][j] = \min \{ \text{dp}[i][j-1], \text{dp}[i+1][j] \} + 1$ 。

既然  $\text{dp}[i][j]$  值代表子串  $\text{str}[i..j]$  最少添加几个字符可以使  $\text{str}[i..j]$  整体都是回文串，所以根据上面的方法求出整个  $\text{dp}$  矩阵之后，我们就得到了  $\text{str}$  中任何一个子串添加几个字符后可以变成回文串。具体请参看如下代码中的 `getDP` 方法。

```
public int[][] getDP(char[] str) {
    int[][] dp = new int[str.length][str.length];
    for (int j = 1; j < str.length; j++) {
        dp[j - 1][j] = str[j - 1] == str[j] ? 0 : 1;
        for (int i = j - 2; i > -1; i--) {
            if (str[i] == str[j]) {
                dp[i][j] = dp[i + 1][j - 1];
            } else {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j - 1]) + 1;
            }
        }
    }
    return dp;
}
```

下面介绍如何根据  $\text{dp}$  矩阵，求在添加字符最少的情况下，让  $\text{str}$  整体都是回文字符串的一种结果。首先， $\text{dp}[0][N-1]$  的值代表整个字符串最少需要添加几个字符，所以，如果最

后的结果记为字符串 `res`, `res` 的长度=`dp[0][N-1]`+`str` 的长度, 然后依次设置 `res` 左右两头的字符。具体过程如下:

1. 如果 `str[i..j]` 中 `str[i]==str[j]`, 那么 `str[i..j]` 变成回文串的最终结果=`str[i]+str[i+1..j-1]` 变成回文串的结果+`str[j]`, 此时 `res` 左右两头的字符为 `str[i]`(也是 `str[j]`), 然后继续根据 `str[i+1..j-1]` 和矩阵 `dp` 来设置 `res` 的中间部分。

2. 如果 `str[i..j]` 中 `str[i]!=str[j]`, 看 `dp[i][j-1]` 和 `dp[i+1][j]` 哪个小。如果 `dp[i][j-1]` 更小, 那么 `str[i..j]` 变成回文串的最终结果=`str[j]+str[i..j-1]` 变成回文串的结果+`str[j]`, 所以此时 `res` 左右两头的字符为 `str[j]`, 然后继续根据 `str[i..j-1]` 和矩阵 `dp` 来设置 `res` 的中间部分。而如果 `dp[i+1][j]` 更小, 那么 `str[i..j]` 变成回文串的最终结果=`str[i]+str[i+1..j]` 变成回文串的结果+`str[i]`, 所以此时 `res` 左右两头的字符为 `str[i]`, 然后继续根据 `str[i+1..j]` 和矩阵 `dp` 来设置 `res` 的中间部分。如果一样大, 任选一种设置方式都可以得出最终结果。

3. 如果发现 `res` 所有的位置都已设置完毕, 过程结束。

原问题解法的全部过程请参看如下代码中的 `getPalindrome1` 方法。

```
public String getPalindromel(String str) {
    if (str == null || str.length() < 2) {
        return str;
    }
    char[] chas = str.toCharArray();
    int[][] dp = getDP(chas);
    char[] res = new char[chas.length + dp[0][chas.length - 1]];
    int i = 0;
    int j = chas.length - 1;
    int resl = 0;
    int resr = res.length - 1;
    while (i <= j) {
        if (chas[i] == chas[j]) {
            res[resl++] = chas[i++];
            res[resr--] = chas[j--];
        } else if (dp[i][j - 1] < dp[i + 1][j]) {
            res[resl++] = chas[j];
            res[resr--] = chas[j--];
        } else {
            res[resl++] = chas[i];
            res[resr--] = chas[i++];
        }
    }
    return String.valueOf(res);
}
```

求解 `dp` 矩阵的时间复杂度为  $O(N^2)$ , 根据 `str` 和 `dp` 矩阵求解最终结果的过程为  $O(N)$ , 所以原问题解法中总的时间复杂度为  $O(N^2)$ 。

进阶问题。如果有最长回文子序列字符串 `strlps`，那么求解的时间复杂度可以加速到  $O(N)$ 。如果 `str` 的长度为  $N$ ，`strlps` 的长度为  $M$ ，则整体回文串的长度应该是  $2 \times N - M$ 。本书提供的解法类似“剥洋葱”的过程，给出示例来具体说明：

`str="A1BC22DE1F"`，`strlps = "1221"`。`res=...`长度为  $2 \times N - M...$

洋葱的第 0 层由 `strlps[0]` 和 `strlps[M-1]` 组成，即“1...1”。从 `str` 最左侧开始找字符‘1’，发现‘A’是 `str` 第 0 个字符，‘1’是 `str` 第 1 个字符，所以左侧第 0 层洋葱圈外的部分为“A”，记为 `leftPart`。从 `str` 最右侧开始找字符‘1’，发现右侧第 0 层洋葱圈外的部分为“F”，记为 `rightPart`。把（`leftPart+rightPart` 的逆序）复制到 `res` 左侧未设值的部分，把（`rightPart+leftPart` 逆序）复制到 `res` 的右侧未设值的部分，即 `result` 变为“AF...FA”。把洋葱的第 0 层复制进 `res` 的左右两侧未设值的部分，即 `result` 变为“AF1...1FA”。至此，洋葱第 0 层被剥掉。洋葱的第 1 层由 `strlps[1]` 和 `strlps[M-2]` 组成，即“2...2”。从 `str` 左侧的洋葱第 0 层往右找“2”，发现左侧第 1 层洋葱圈外的部分为“BC”，记为 `leftPart`。从 `str` 右侧的洋葱第 0 层往左找“2”，发现右侧第 1 层洋葱圈外的部分为“DE”，记为 `rightPart`。把（`leftPart+rightPart` 的逆序）复制到 `res` 左侧未设值的部分，把（`rightPart+leftPart` 逆序）复制到 `res` 的右侧未设值的部分，`res` 变为“AF1BCED...DECB1FA”。把洋葱的第 1 层复制进 `res` 的左右两侧未设值的部分，即 `result` 变为“AF1BCED2...2DECB1FA”。第 1 层被剥掉，洋葱剥完了，返回“AF1BCED22DECB1FA”。整个过程就是不断找到洋葱圈的左部分和右部分，把（`leftPart+rightPart` 的逆序）复制到 `res` 左侧未设值的部分，把（`rightPart+leftPart` 逆序）复制到 `res` 的右侧未设值的部分，洋葱剥完则过程结束。具体请参看如下的 `getPalindrome2` 方法。

```
public String getPalindrome2(String str, String strlps) {
    if (str == null || str.equals("")) {
        return "";
    }
    char[] chas = str.toCharArray();
    char[] lps = strlps.toCharArray();
    char[] res = new char[2 * chas.length - lps.length];
    int chasl = 0;
    int chasr = chas.length - 1;
    int lpsl = 0;
    int lpsr = lps.length - 1;
    int resl = 0;
    int resr = res.length - 1;
    int tmp1 = 0;
    int tmp2 = 0;
    while (lpsl <= lpsr) {
        tmp1 = chasl;
        tmp2 = chasr;
        while (chas[chasl] != lps[lpsl]) {

```

```

        chasl++;
    }
    while (chas[chasr] != lps[lpsr]) {
        chasr--;
    }
    set(res, resl, resr, chas, tmp1, chasl, chasr, tmpr);
    resl += chasl - tmp1 + tmpr - chasr;
    resr -= chasl - tmp1 + tmpr - chasr;
    res[resl++] = chas[chasl++];
    res[resr--] = chas[chasr--];
    lpsl++;
    lpsr--;
}
return String.valueOf(res);
}

public void set(char[] res, int resl, int resr, char[] chas, int ls,
               int le, int rs, int re) {
    for (int i = ls; i < le; i++) {
        res[resl++] = chas[i];
        res[resr--] = chas[i];
    }
    for (int i = re; i > rs; i--) {
        res[resl++] = chas[i];
        res[resr--] = chas[i];
    }
}
}

```

## 括号字符串的有效性和最长有效长度

### 【题目】

给定一个字符串 `str`，判断是不是整体有效的括号字符串。

### 【举例】

`str="()"`，返回 `true`；`str="(())"`，返回 `true`；`str="()())"`，返回 `true`。

`str="())"`。返回 `false`；`str="()("`，返回 `false`；`str="()a()"`，返回 `false`。

### 【补充题目】

给定一个括号字符串 `str`，返回最长的有效括号子串。