<https://leetcode.com/problems/shortest-palindrome/description/>

You are given a string s. You can convert s to a palindrome by adding characters in front of it.

Return *the shortest palindrome you can find by performing this transformation*.

**Example 1:**

Input: s = "aacecaaa"

Output: "aaacecaaa"

**Example 2:**

Input: s = "abcd"

Output: "dcbabcd"

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of lowercase English letters only.

**Attempt 1: 2023-07-18**

**Solution 1:  Brute Force (10min, TLE 120/123)**

**'We can convert it to an alternative problem"find the longest palindrome substring starts from index 0".'**

class Solution {

public String shortestPalindrome(String s) {

// Set as -1 is convenient for no palindrome found and whole

// string has to be considered as compensation to be added

// in front of s, work together with s.substring(-1 + 1)

int cutpoint = -1;

for(int i = s.length() - 1; i >= 0; i--) {

if(isPalindrome(s, 0, i)) {

cutpoint = i;

break;

}

}

// 'cutpoint + 1' means the substring should start from the

// 1st character after cutpoint

StringBuffer sb = new StringBuffer(s.substring(cutpoint + 1));

return sb.reverse().toString() + s;

}

private boolean isPalindrome(String s, int i, int j) {

while(i <= j) {

if(s.charAt(i) == s.charAt(j)) {

i++;

j--;

} else {

return false;

}

}

return true;

}

}

**Solution 2:  Rolling Hash (360 min)**

**Wrong Solution**

**We cannot scan backward as for(int i = n - 1; i >= 0; i--) like what we did in Brute Force solution, because not like Brute Force, we can do backward traverse and try to identify a largest i if s[0, i] is a palindrome, if cutpoint 'i' more close to last position, it will find more quickly than scanning forward.**

**But here is totally different, in Rolling Hash, we have to maintain an iterative way to calculate a 'multiplier' kind of 'forward\_hash' as what we did in L1044, we deal with each digit individually, not like Brute Force treat as a substring as s[0, i], e.g if we scan backward, the last digit in "abcd" as 'd' will be treated individually and result is 'd' has same forward & backward hash, then we wrongly update 'cutpoint' from -1 to 3, even "abcd" is not a palindrome at all.**

**Hence we have to scan from beginning like how we di Rolling Harsh usually, gradually increment a new digit on scanned substring, then calculate a new forward & backward hash**

class Solution {

public String shortestPalindrome(String s) {

int n = s.length();

int cutpoint = -1;

long mod = 1000000007;

int base = 256;

long pow = 1;

long forward\_hash = 0;

long backward\_hash = 0;

for(int i = n - 1; i >= 0; i--) {

forward\_hash = (forward\_hash \* base + s.charAt(i)) % mod;

backward\_hash = (backward\_hash + s.charAt(i) \* pow) % mod;

pow = pow \* base % mod;

if(forward\_hash == backward\_hash) {

cutpoint = i;

break;

}

}

return new StringBuilder().append(s.substring(cutpoint)).reverse().append(s).toString();

}

}

**Note: Compare to L1044, the Rolling Hash in L214 is more like calculate the 'multipler' in L1044, since we don't need to build a sliding window to find same substring, instead we just need to add one character in each loop iteration forward and backward**

class Solution {

public String shortestPalindrome(String s) {

int n = s.length();

int cutpoint = -1;

long mod = 1000000007;

int base = 256;

long pow = 1;

long forward\_hash = 0;

long backward\_hash = 0;

for(int i = n - 1; i >= 0; i--) {

forward\_hash = (forward\_hash \* base + s.charAt(i)) % mod;

backward\_hash = (backward\_hash + s.charAt(i) \* pow) % mod;

pow = pow \* base % mod;

if(forward\_hash == backward\_hash) {

cutpoint = i;

break;

}

}

return new StringBuilder().append(s.substring(cutpoint)).reverse().append(s).toString();

}

}

**Refer to**

<https://leetcode.com/problems/shortest-palindrome/solutions/60153/8-line-o-n-method-using-rabin-karp-rolling-hash/>

This problem is indeed computing the longest palindromic prefix of a string s. A naive approach would be computing all the prefixes of s and its reverse, and then finding the longest pair of prefixes that are equal.

Unfortunately, this method requires quadratic time and space since the length sum of all prefixes is 1+2+...+|s| = Θ(|s|^2).

Via the help of the Rolling Hash method, the above process can be optimized down to linear time. For more details, you can visit [here](https://en.wikipedia.org/wiki/Rolling_hash) and [here](http://courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf).

public String shortestPalindrome(String s) {

int n = s.length(), pos = -1;

long B = 29, MOD = 1000000007, POW = 1, hash1 = 0, hash2 = 0;

for (int i = 0; i < n; i++, POW = POW \* B % MOD) {

hash1 = (hash1 \* B + s.charAt(i) - 'a' + 1) % MOD;

hash2 = (hash2 + (s.charAt(i) - 'a' + 1) \* POW) % MOD;

if (hash1 == hash2) pos = i;

}

return new StringBuilder().append(s.substring(pos + 1, n)).reverse().append(s).toString();

}

<https://leetcode.com/problems/shortest-palindrome/solutions/60153/8-line-o-n-method-using-rabin-karp-rolling-hash/comments/61282>

Consider a decimal example (base = 10). Say we are given a number 7134. If we read it from left to right, we get 7134. And 4317 if we read it from right to left.

hash1 is the left--to-right fashion:

* hash1 = 0
* hash1 = 0 \* 10 + 7 = 7
* hash1 = 7 \* 10 + 1 = 71
* hash1 = 71 \* 10 + 3 = 713
* hash1 = 713 \* 10 + 4 = 7134

hash2 is the right-to-left fashion:

* hash2 = 0
* hash2 = 0 + 7 \* 1 = 7
* hash2 = 7 + 1 \* 10 = 17
* hash2 = 17 + 3 \* 100 = 317
* hash2 = 317 + 4 \* 1000 = 4317

A palindrome must be read the same from left to right and from right to left. So in this case, 7134 is not a palindrome.

**Above is an example for the decimal case, and for rolling hashing, the only differences are:**

1. **Base is not 10, but any constant >= 26.**
2. **hash1 and hash2 are not the exact value, but the exact value modulo a big prime. (Since the exact value is too large to fit in a 32-bit integer.)**

As you may notice, the rolling hash function is not an injection, which means that two different strings may share the same hash code.(This is also commonly called a conflict.) But in real, it is very difficult to trigger many conflicts (sometimes not even a single one) unless there are sufficiently many strings given. Therefore, if hash1 is not equal to hash2 for some string, then definitely it is not a palindrome. On the other hand, if they are equal, it means the string is a palindrome with extreme high probability.

**Refer to**

<https://leetcode.wang/leetcode-214-Shortest-Palindrome.html>

在字符串开头补充最少的字符，使得当前字符串成为回文串。

# **解法一 暴力**

先判断整个字符串是不是回文串，如果是的话，就直接将当前字符串返回。不是的话，进行下一步。

判断去掉末尾 1 个字符的字符串是不是回文串，如果是的话，就将末尾的 1 个字符加到原字符串的头部返回。不是的话，进行下一步。

判断去掉末尾 2 个字符的字符串是不是回文串，如果是的话，就将末尾的 2 个字符倒置后加到原字符串的头部返回。不是的话，进行下一步。

判断去掉末尾 3 个字符的字符串是不是回文串，如果是的话，就将末尾的 3 个字符倒置后加到原字符串的头部返回。不是的话，进行下一步。

...

直到判断去掉末尾的 n - 1 个字符，整个字符串剩下一个字符，把末尾的 n - 1 个字符倒置后加到原字符串的头部返回。

举个例子，比如字符串 abbacd。

原字符串 abbacd

先判断 abbacd 是不是回文串, 发现不是, 执行下一步

判断 abbac 是不是回文串, 发现不是, 执行下一步

判断 abba 是不是回文串, 发现是，将末尾的 2 个字符 cd 倒置后加到原字符串的头部,

即 dcabbacd

代码的话，判断是否是回文串的话可以用 [125 题](https://leetcode.wang/leetcode-125-Valid-Palindrome.html) 的思想，利用双指针法。

//判断是否是回文串, 传入字符串的范围

public boolean isPalindromic(String s, int start, int end) {

char[] c = s.toCharArray();

while (start < end) {

if (c[start] != c[end]) {

return false;

}

start++;

end--;

}

return true;

}

public String shortestPalindrome(String s) {

int end = s.length() - 1;

//找到回文串的结尾, 用 end 标记

for (; end > 0; end--) {

if (isPalindromic(s, 0, end)) {

break;

}

}

//将末尾的几个倒置然后加到原字符串开头

return new StringBuilder(s.substring(end + 1)).reverse() + s;

}

遗憾的是超时了（前几天这个方法还能是过的，今天突然就超时了，官方应该是增加了 case）。



上边字符串的长度达到了四万多，类似于这样的 aaaaaaaaaaaaaacdaaaaaaaaaaaaaa，每次调用判断字符串是否是回文串的时候，都需要判断很多次，最终时间复杂度达到了 O(n²)，造成了超时。

# **解法二**

参考 [这里](https://leetcode.com/problems/shortest-palindrome/discuss/60098/My-7-lines-recursive-Java-solution)。

根据解法一，我们其实就是在**寻找从开头开始的最长回文串**（这个很关键，后边所有的解法都是基于这个了），然后将末尾的除去最长回文串部分的几个字符倒置后加到原字符串开头即可。

我们只需要两个指针， i 和 j，i 初始化为 0，j 初始化为字符串长度减 1。然后依次判断 s[i] 和 s[j] 是否相同，相同的话， i 就进行加 1，j 进行减 1。 s[i] 和 s[j] 不同的话，只将 j 进行减 1。

看几个例子。

abbacde

a b b a c d e

^ ^

i j

如上所示, s[i] != s[j], j--

a b b a c d e

^ ^

i j

如上所示, s[i] != s[j], j--

a b b a c d e

^ ^

i j

如上所示, s[i] != s[j], j--

a b b a c d e

^ ^

i j

如上所示, s[i] == s[j], i++, j--

a b b a c d e

^ ^

i j

如上所示, s[i] == s[j], i++, j--

a b b a c d e

^ ^

j i

如上所示, s[i] == s[j], i++, j--

a b b a c d e

^ ^

j i

如上所示, s[i] == s[j], i++, j--

a b b a c d e

^ ^

j i

如上所示, j < 0, 结束循环。

此时 i 指向最长回文串的下一个字符串,我们只需要把 i 到 最后的字符倒置加到开头即可。

当然，上边是最理想的情况，如果 j 在最长回文串外提前出现了和 i 相同的字符会有影响吗？

abbacba

a b b a c b a

^ ^

i j

如上所示, s[i] == s[j], i++, j--

a b b a c b a

^ ^

i j

如上所示, s[i] == s[j], i++, j--

a b b a c b a

^ ^

i j

如上所示, s[i] != s[j], j--

a b b a c b a

^ ^

i j

如上所示, s[i] != s[j], j--

a b b a c b a

^

i

j

如上所示, s[i] == s[j], i++, j--

a b b a c b a

^ ^

j i

如上所示, s[i] != s[j], j--

a b b a c b a

^ ^

j i

如上所示, s[i] == s[j], i++, j--

a b b a c d e

^ ^

j i

如上所示, j < 0, 结束循环。

会发现此时 i 和之前一样, 依旧指向最长回文串的下一个字符,我们只需要把 i 到最后的字符倒置加到开头即可。

可以看到上边的两种情况，只要 j 进入了最长回文子串，一定会使得 i 走出最长回文子串。所以我们可以利用双指针写一下代码了。

public String shortestPalindrome(String s) {

int i = 0, j = s.length() - 1;

char[] c = s.toCharArray();

while (j >= 0) {

if (i == j){

continue;

}

if (c[i] == c[j]) {

i++;

}

j--;

}

//此时代表整个字符串是回文串

if (i == s.length()) {

return s;

}

//后缀

String suffix = s.substring(i);

//后缀倒置

String reverse = new StringBuilder(suffix).reverse().toString();

//加到开头

return reverse + s;

}

看起来没什么问题，但还有一种情况，那就是 i 提前走出了最长回文子串，看下边的例子。

ababbcefbbaba

a b a b b c e f b b a b a

^ ^

i j

i 和 j 同时移动, 一直是相等, 直到下边的情况

a b a b b c e f b b a b a

^ ^

i j

然后继续移动, 最后就变成了下边的样子

a b a b b c e f b b a b a

^ ^

j i

会发现此时 0 到 i - 1 并不是一个回文串, 所以我们需要递归的去解决这个问题

此时我们并没有找到最长回文串，但是我们可以肯定最长回文串一定在 0 到 i 之间，所以我们只需要递归的从s[0, i) 中继续寻找最长回文串即可。

因为上边的所有情况，都保证了 i 一定可以走出最长回文串，只不过可能超出一部分，所以用递归解决即可。代码的整体框架不需要改变。

public String shortestPalindrome(String s) {

int i = 0, j = s.length() - 1;

char[] c = s.toCharArray();

while (j >= 0) {

if (c[i] == c[j]) {

i++;

}

j--;

}

//此时代表整个字符串是回文串

if (i == s.length()) {

return s;

}

//后缀

String suffix = s.substring(i);

//后缀倒置

String reverse = new StringBuilder(suffix).reverse().toString();

//递归 s[0,i),寻找开头开始的最长回文串，将其余部分加到开头和结尾

return reverse + shortestPalindrome(s.substring(0, i)) + suffix;

}

这个解法相对解法一会好一些，但对于某些极端情况，时间复杂度依旧会达到 O(n²)。比如下边的例子。

aababababa

a a b a b a b a b a

^ ^

i j

如上所示, s[i] == s[j], i++, j--

a a b a b a b a b a

^ ^

i j

如上所示, s[i] != s[j], j--

a a b a b a b a b a

^ ^

i j

如上所示, 此时 i 和 j 之间是一个回文串，所以 i 和 j 最终会变成下边的样子

a a b a b a b a b a

^ ^

j i

结合上边的代码，接下来去掉末尾字符，将对下边的字符串进行递归

a a b a b a b a

此时会发现和最开始的结构一样，最终结果是去掉末尾的两个字符，继续对下边的字符串递归

a a b a b a

此时会发现和最开始的结构一样，最终结果是去掉末尾的两个字符，继续对下边的字符串递归

a a b a

此时会发现和最开始的结构一样，最终结果是去掉末尾的两个字符，继续对下边的字符串递归

a a

此时是回文串了，递归结束

所以每次递归只会减少两个字符，递归路径如下

a a b a b a b a b a

a a b a b a b a

a a b a b a

a a b a

a a

如果初始字符串是上边的结构，即 aaba...ba...ba...ba，有几万个 ba 的话，和解法一一样会造成超时。由于 leetcode 中没有这种 case ，所以这个解法也就 AC 了。

# **解法三**

寻找开头开始的最长回文串，我们回到更暴力的方法。

将原始字符串逆序，然后比较对应的子串即可判断是否是回文串。举个例子。

abbacd

原s: abbacd, 长度记为 n

逆r: dcabba, 长度记为 n

判断 s[0,n) 和 r[0,n)

abbacd != dcabba

判断 s[0,n - 1) 和 r[1,n)

abbac != cabba

判断 s[0,n - 2) 和 r[2,n)

abba == abba

从开头开始的最长回文串也就找到了, 接下来只需要使用之前的方法。

将末尾不是回文串的部分倒置加到原字符串开头即可。

代码的话，也很好写了。

public String shortestPalindrome(String s) {

String r = new StringBuilder(s).reverse().toString();

int n = s.length();

int i = 0;

for (; i < n; i++) {

if (s.substring(0, n - i).equals(r.substring(i))) {

break;

}

}

return new StringBuilder(s.substring(n - i)).reverse() + s;

}

然后它竟然 AC 了，当然这个时间复杂度是 O(n²)，之所以通过了，还是取决于 test cases 。

# **解法四**

在解法三倒置的基础上进行一下优化，参考 [这里](https://leetcode.com/problems/shortest-palindrome/discuss/60153/8-line-O(n)-method-using-Rabin-Karp-rolling-hash)。

用到了字符串匹配算法 RK 算法的思想，也就是滚动哈希。

解法三中，每次比较两个字符串是否相等都需要一个字符一个字符比较，如果我们把字符串通过 hash 算法映射到数字，就可以只判断数字是否相等即可。

而 hash 算法，这里的话，我们将 a 看做 1，b 看做 2 ... 以此类推，然后把字符串看做是 26 进制的一个数字，将其转为十进制后的值作为 hash 值。

也许需要一些进制转换的知识，可以参考 [这里](https://www.zhihu.com/question/357414448/answer/949086536)。

举个例子，对于 abcd。

a b c d

1 2 3 4

26^3 26^2 26 1

那么 abcd的 hash值就是 4+3∗26+2∗262+1∗2634+3∗26+2∗26^2+1∗26^3。

这样做的好处是，我们可以通过前一个字符串的 hash值，算出当前字符串的 hash值。

举个例子。

对于字符串 abb，如果我们知道了它的 hash值是 x，那么对于 abba的 hash值，因为新增加的数字 a对应 1，所以 abba的 hash值就是 (x \* 26 + 1)。

所以代码可以写成下边的样子。

public String shortestPalindrome(String s) {

int n = s.length(), pos = -1;

int b = 26; // 基数

int pow = 1; // 为了方便计算倒置字符串的 hash 值

char[] c = s.toCharArray();

int hash1 = 0, hash2 = 0;

for (int i = 0; i < n; i++, pow = pow \* b) {

hash1 = hash1 \* b + (c[i] - 'a' + 1);

// 倒置字符串的 hash 值, 新增的字符要放到最高位

hash2 = hash2 + (c[i] - 'a' + 1) \* pow;

if (hash1 == hash2) {

pos = i;

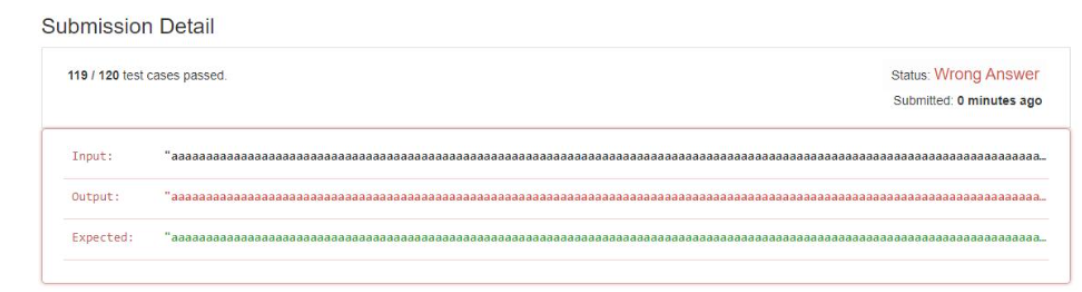
}

}

return new StringBuilder(s.substring(pos + 1)).reverse() + s;

}

理论上，上边的代码是可行的，但会发现出现了 wrong answer。



我猜测下原因，不是十分确定。

最直接的问题肯定是由于我们用 int 存储 hash 值，所以一定会出现溢出的情况。溢出以后，接着带来了 hash 冲突，从而使得相同的 hash 值，但是字符串并不相同。

基于上边的分析，我们可以在 pos = i 之前判断一下当前是否是回文串。

public boolean isPalindromic(String s, int start, int end) {

char[] c = s.toCharArray();

while (start < end) {

if (c[start] != c[end]) {

return false;

}

start++;

end--;

}

return true;

}

public String shortestPalindrome(String s) {

int n = s.length(), pos = -1;

int b = 26; // 基数

int pow = 1; // 为了方便计算倒置字符串的 hash 值

char[] c = s.toCharArray();

int hash1 = 0, hash2 = 0;

for (int i = 0; i < n; i++, pow = pow \* b) {

hash1 = hash1 \* b + (c[i] - 'a' + 1);

// 倒置字符串的 hash 值, 新增的字符要放到最高位

hash2 = hash2 + (c[i] - 'a' + 1) \* pow;

if (hash1 == hash2) {

//确认下当前是否是回文串

if (isPalindromic(s,0,i)) {

pos = i;

}

}

}

return new StringBuilder(s.substring(pos + 1)).reverse() + s;

}

但是超时了。



然后就是换 hash 算法，我们可以把每次的结果取模，这样就不会溢出了。

public String shortestPalindrome(String s) {

int n = s.length(), pos = -1;

int b = 26; // 基数

int pow = 1; // 为了方便计算倒置字符串的 hash 值

char[] c = s.toCharArray();

int hash1 = 0, hash2 = 0;

int mod = 1000000;

for (int i = 0; i < n; i++, pow = (pow \* b) % mod) {

hash1 = (hash1 \* b + (c[i] - 'a' + 1)) % mod;

// 倒置字符串的 hash 值, 新增的字符要放到最高位

hash2 = (hash2 + (c[i] - 'a' + 1) \* pow)% mod;

if (hash1 == hash2) {

pos = i;

}

}

return new StringBuilder(s.substring(pos + 1)).reverse() + s;

}

虽然这种方法 AC 了，但我觉得是侥幸的，我觉得即使每次取模，并不能保证不会出现 hash 冲突，只是当前的 test case 没有出现 hash 冲突。当然这是我的想法，并不是很确定，大家有其他想法欢迎和我交流。

感谢 @[franklinqin0](https://www.zhihu.com/people/franklinqin7) 指出，上边确认当前是否是回文串的时候，我们调用了 isPalindromic ，但超时了，这里的话我们还可以和它的逆置字符串进行比较。

public String shortestPalindrome(String s) {

int n = s.length(), pos = -1;

int b = 26; // 基数

int pow = 1; // 为了方便计算倒置字符串的 hash 值

char[] c = s.toCharArray();

String rev = new StringBuilder(s).reverse().toString();

int hash1 = 0, hash2 = 0;

for (int i = 0; i < n; i++, pow = pow \* b) {

hash1 = hash1 \* b + (c[i] - 'a' + 1);

// 倒置字符串的 hash 值, 新增的字符要放到最高位

hash2 = hash2 + (c[i] - 'a' + 1) \* pow;

if (hash1 == hash2) {

if (s.substring(0, i + 1).equals(rev.substring(n - i - 1))) {

pos = i;

}

}

}

return new StringBuilder(s.substring(pos + 1)).reverse() + s;

}

这样做的话就不会超时了，但如果分析时间复杂度的话其实是一样的，很神奇。

# **解法五**

参考 [这里](https://leetcode.com/problems/shortest-palindrome/discuss/60113/Clean-KMP-solution-with-super-detailed-explanation)。

这个解法的前提是你熟悉另一种字符串匹配算法，即 KMP 算法。推荐两个链接，大家可以先学习一下，我就不多说了。KMP 算法代码简单，但理解求 next 数组的话，确实有些麻烦。

<http://jakeboxer.com/blog/2009/12/13/the-knuth-morris-pratt-algorithm-in-my-own-words/>

<https://learnku.com/articles/10622/introduction-of-kmp-algorithm-and-derivation-of-next-array>

如果熟悉了 KMP 算法，下边就简单了。

再回想一下解法三，倒置字符串的思路，依次比较对应子串。

abbacd

原s: abbacd, 长度记为 n

逆r: dcabba, 长度记为 n

我们把两个字符串写在一起

abbacd dcabba

判断 abbacd 和 dcabba 是否相等

判断 abbac 和 cabba 是否相等

判断 abba 和 abba 是否相等

如果我们把 abbacd dcabba看成一个字符串，中间加上一个分隔符 #，abbacd#dcabba。

回味一下上边的三条判断，判断 XXX 和 XXX 是否相等，按列看一下。

左半部分 abbacd，abbac , abba 其实就是 abbacd#dcabba 的一些前缀。

右半部分dcabba，cabba，abba 其实就是 abbacd#dcabba 的一些后缀。

寻找前缀和后缀相等。

想一想 KMP 算法，这不就是 next 数组做的事情吗。

而我们中间加了分隔符，也就保证了前缀和后缀相等时，前缀一定在 abbacd 中。

换句话说，我们如果求出了 abbacd#dcabba 的 next 数组，因为我们构造的字符串后缀就是原字符串的倒置，前缀后缀相等时，也就意味着当前前缀是一个回文串，而 next 数组是寻求最长的前缀，我们也就找到了开头开始的最长回文串。

因为 next 数组的含义并不统一，但 KMP 算法本质上都是一样的，所以下边的代码仅供参考。

我的 next 数组 next[i] 所考虑的对应字符串不包含 s[i]。

public String shortestPalindrome(String s) {

String ss = s + '#' + new StringBuilder(s).reverse();

int max = getLastNext(ss);

return new StringBuilder(s.substring(max)).reverse() + s;

}

//返回 next 数组的最后一个值

public int getLastNext(String s) {

int n = s.length();

char[] c = s.toCharArray();

int[] next = new int[n + 1];

next[0] = -1;

next[1] = 0;

int k = 0;

int i = 2;

while (i <= n) {

if (k == -1 || c[i - 1] == c[k]) {

next[i] = k + 1;

k++;

i++;

} else {

k = next[k];

}

}

return next[n];

}

# **解法六**

参考 [这里](https://leetcode.com/problems/shortest-palindrome/discuss/60188/My-C++-O(n)-solution-based-on-Manacher's-algorithm) 。

大家还记得 [第 5 题](https://leetcode.wang/leetCode-5-Longest-Palindromic-Substring.html) 吗？求最长回文子串。

这里我们已经把题目转换成了求开头开始的最长回文子串，很明显这个问题只是第 5 题的子问题了。但这道题时间复杂度差不多只有 O(n) 才会通过。这就必须使用 [第 5 题](https://leetcode.wang/leetCode-5-Longest-Palindromic-Substring.html) 介绍的马拉车算法了。

直接把马拉车算法粘贴过来即可，然后在最后稍微修改一下即可。大家不熟悉的话，可以参考 [一文让你彻底明白马拉车算法](https://zhuanlan.zhihu.com/p/70532099)。

public String preProcess(String s) {

int n = s.length();

if (n == 0) {

return "^$";

}

String ret = "^";

for (int i = 0; i < n; i++)

ret += "#" + s.charAt(i);

ret += "#$";

return ret;

}

// 马拉车算法

public String shortestPalindrome(String s) {

String T = preProcess(s);

int n = T.length();

int[] P = new int[n];

int C = 0, R = 0;

for (int i = 1; i < n - 1; i++) {

int i\_mirror = 2 \* C - i;

if (R > i) {

P[i] = Math.min(R - i, P[i\_mirror]);// 防止超出 R

} else {

P[i] = 0;// 等于 R 的情况

}

// 碰到之前讲的三种情况时候，需要利用中心扩展法

while (T.charAt(i + 1 + P[i]) == T.charAt(i - 1 - P[i])) {

P[i]++;

}

// 判断是否需要更新 R

if (i + P[i] > R) {

C = i;

R = i + P[i];

}

}

//这里的话需要修改

int maxLen = 0;

int centerIndex = 0;

for (int i = 1; i < n - 1; i++) {

int start = (i - P[i]) / 2;

//我们要判断当前回文串是不是开头是不是从 0 开始的

if (start == 0) {

maxLen = P[i] > maxLen ? P[i] : maxLen;

}

}

return new StringBuilder(s.substring(maxLen)).reverse() + s;

}

# **总**

这道题太强了，六种解法，各有特色。把这道题捋下来着实不易，涉及到很多算法。但懂了之后确实心旷神怡。

花时间最多的地方其实在 KMP 那里，求 next 数组确实难理解一些。然后递归解法，看起来简单，其实理解起来的话也没那么容易。最后没想到又回到了马拉车算法，不得不再佩服一下这个解法，神仙操作，直接将时间复杂度优化到了 O(n)。