```
	private void heapInsert(int index) {
		while (index != 0) {
			int parent = (index - 1) / 2;
			if (heap[index].times < heap[parent].times) {
				swap(parent, index);
				index = parent;
			} else {
				break;
			}
		}
	}

	private void heapify(int index, int heapSize) {
		int l = index * 2 + 1;
		int r = index * 2 + 2;
		int smallest = index;
		while (l < heapSize) {
		    if (heap[l].times < heap[index].times) {
		        smallest = l;
		    }
		    if (r < heapSize && heap[r].times < heap[smallest].times) {
		        smallest = r;
		    }
		    if (smallest != index) {
		        swap(smallest, index);
		    } else {
		        break;
		    }
		    index = smallest;
		    l = index * 2 + 1;
		    r = index * 2 + 2;
		}
	}

	private void swap(int index1, int index2) {
		nodeIndexMap.put(heap[index1], index2);
		nodeIndexMap.put(heap[index2], index1);
		Node tmp = heap[index1];
		heap[index1] = heap[index2];
		heap[index2] = tmp;
	}

}
```

Manacher 算法

【题目】

给定一个字符串 str，返回 str 中最长回文子串的长度。

【举例】

str="123"，其中的最长回文子串为"1"、"2"或者"3"，所以返回 1。

str="abc1234321ab"，其中的最长回文子串为"1234321"，所以返回 7。

【进阶题目】

给定一个字符串 str，想通过添加字符的方式使得 str 整体都变成回文字符串，但要求只能在 str 的末尾添加字符，请返回在 str 后面添加的最短字符串。

【举例】

str="12"。在末尾添加"1"之后，str 变为"121"，是回文串。在末尾添加"21"之后，str 变为"1221"，也是回文串。但"1"是所有添加方案中最短的，所以返回"1"。

【要求】

如果 str 的长度为 N，解决原问题和进阶问题的时间复杂度都达到 $O(N)$。

【难度】

将 ★★★★

【解答】

本文的重点是介绍 Manacher 算法，该算法是由 Glenn Manacher 于 1975 年首次发明的。Manacher 算法解决的问题是在线性时间内找到一个字符串的最长回文子串，比起能够解决该问题的其他算法，Manacher 算法算比较好理解和实现的。

先来说一个很好理解的方法。从左到右遍历字符串，遍历到每个字符的时候，都看看以这个字符作为中心能够产生多大的回文字符串。比如 str="abacaba"，以 str[0]=='a'为中心的回文字符串最大长度为 1，以 str[1]=='b'为中心的回文字符串最大长度为 3，……其中最大的回文子串是以 str[3]=='c'为中心的时候。这种方法非常容易理解，只要解决奇回文和偶回文寻找方式的不同就可以。比如"121"是奇回文，有确定的轴'2'。"1221"是偶回文，没有确定的轴，回文的虚轴在"22"中间。但是这种方法有明显的问题，之前遍历过的字符完全无法指导后面遍历的过程，也就是对每个字符来说都是从自己的位置出发，往左右两个方向扩出去检查。这样，对每个字符来说，往外扩的代价都是一个级别的。举一个极端的例

子"aaaaaaaaaaaaaaa"，对每一个'a'来讲，都是扩到边界才停止。所以每一个字符扩出去检查的代价都是 $O(N)$，所以总的时间复杂度为 $O(N^2)$。Manacher 算法可以做到 $O(N)$的时间复杂度，精髓是之前字符的“扩”过程，可以指导后面字符的“扩”过程，使得每次的“扩”过程不都是从无开始。以下是 Manacher 算法解决原问题的过程：

1．因为奇回文和偶回文在判断时比较麻烦，所以对 str 进行处理，把每个字符开头、结尾和中间插入一个特殊字符'#'来得到一个新的字符串数组。比如 str="bcbaa"，处理后为"#b#c#b#a#a#"，然后从每个字符左右扩出去的方式找最大回文子串就方便多了。对奇回文来说，不这么处理也能通过扩的方式找到，比如"bcb"，从'c'开始向左右两侧扩出去能找到最大回文。处理后为"#b#c#b#"，从'c'开始向左右两侧扩出去依然能找到最大回文。对偶回文来说，不处理而直接通过扩的方式是找不到的，比如"aa"，因为没有确定的轴，但是处理后为"#a#a#"，就可以通过从中间的'#'扩出去的方式找到最大回文。所以通过这样的处理方式，最大回文子串无论是偶回文还是奇回文，都可以通过统一的“扩”过程找到，解决了差异性的问题。同时要说的是，这个特殊字符是什么无所谓，甚至可以是字符串中出现的字符，也不会影响最终的结果，就是一个纯辅助的作用。

具体的处理过程请参看如下代码中的 manacherString 方法。

```
public char[] manacherString(String str) {
        char[] charArr = str.toCharArray();
        char[] res = new char[str.length() * 2 + 1];
        int index = 0;
        for (int i = 0; i != res.length; i++) {
                res[i] = (i & 1) == 0 ? '#' : charArr[index++];
        }
        return res;
}
```

2．假设 str 处理之后的字符串记为 charArr。对每个字符（包括特殊字符）都进行“优化后”的扩过程。在介绍“优化后”的扩过程之前，先解释如下三个辅助变量的意义。

- 数组 pArr。长度与 charArr 长度一样。pArr[i]的意义是以 i 位置上的字符(charArr[i])作为回文中心的情况下，扩出去得到的最大回文半径是多少。举个例子来说明，对"#c#a#b#a#c#"来说，pArr[0..9]为[1,2,1,2,1,6,1,2,1,2,1]。我们的整个过程就是在从左到右遍历的过程中，依次计算每个位置的最大回文半径值。
- 整数 pR。这个变量的意义是之前遍历的所有字符的所有回文半径中，最右即将到达的位置。还是以"#c#a#b#a#c#"为例来说，还没遍历之前 pR，初始设置为-1。charArr[0]=='#'的回文半径为 1，所以目前回文半径向右只能扩到位置 0，回文半

径最右即将到达的位置变为 1(pR=1)。charArr[1]=='~~#~~c'的回文半径为 2，此时所有的回文半径向右能扩到位置 2，所以回文半径最右即将到达的位置变为 3(pR=3)。charArr[2]=='#'的回文半径为 1，所以位置 2 向右只能扩到位置 2，回文半径最右即将到达的位置不变，仍是 3(pR=3)。charArr[3]=='a'的回文半径为 2，所以位置 3 向右能扩到位置 4，所以回文半径最右即将到达的位置变为 5(pR=5)。charArr[4]=='#'的回文半径为 1，所以位置 4 向右只能扩到位置 4，回文半径最右即将到达的位置不变仍是 5(pR=5)。charArr[5]=='b'的回文半径为 6，所以位置 4 向右能扩到位置 10，回文半径最右即将到达的位置变为 11(pR=11)。此时已经到达整个字符数组的结尾，所以之后的过程中 pR 将不再变化。换句话说，pR 就是遍历过的所有字符中向右扩出来的最大右边界。只要右边界更往右，pR 就更新。

- 整数 index。这个变量表示最近一次 pR 更新时，那个回文中心的位置。以刚刚的例子来说，遍历到 charArr[0]时 pR 更新，index 就更新为 0。遍历到 charArr[1]时 pR 更新，index 就更新为 1……遍历到 charArr[5]时 pR 更新，index 就更新为 5。之后的过程中，pR 将不再更新，所以 index 将一直是 5。

3．只要能够从左到右依次算出数组 pArr 每个位置的值，最大的那个值实际上就是处理后的 charArr 中最大的回文半径，根据最大的回文半径，再对应回原字符串的话，整个问题就解决了。步骤 3 就是从左到右依次计算出 pArr 数组每个位置的值的过程。

1）假设现在计算到位置 *i* 的字符 charArr[i]，在 *i* 之前位置的计算过程中，都会不断地更新 pR 和 index 的值，即位置 *i* 之前的 index 这个回文中心扩出了一个目前最右的回文边界 pR。

2）如果 pR-1 位置没有包住当前的 *i* 位置。比如"#c#a#b#a#c#"，计算到 charArr[1]=='c'时，pR 为 1。也就是说，右边界在 1 位置，1 位置为最右回文半径即将到达但还没有达到的位置，所以当前的 pR-1 位置没有包住当前的 *i* 位置。此时和普通做法一样，从 *i* 位置字符开始，向左右两侧扩出去检查，此时的“扩”过程没有获得加速。

3）如果 pR-1 位置包住了当前的 *i* 位置。比如"#c#a#b#a#c#"，计算到 charArr[6...10]时，pR 都为 11，此时 pR-1 包住了位置 6~10。这种情况下，检查过程是可以获得优化的，这也是 manacher 算法的核心内容，如图 9-14 所示。

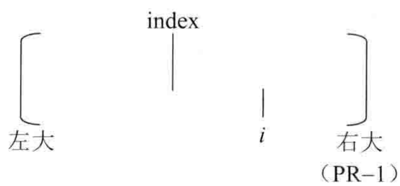

图 9-14

在图 9-14 中，位置 *i* 是要计算回文半径(pArr[i])的位置。pR-1 位置此时是包住位置 *i* 的。同时根据 index 的定义，index 是 pR 更新时那个回文中心的位置，所以如果 pR-1 位置以 index 为中心对称，即图 9-14 中的“左大”位置，那么从“左大”位置到 pR-1 位置一定是以 index 为中心的回文串，我们把这个回文串叫作大回文串，同时把 pR-1 位置称为“右大”位置。既然回文半径数组 pArr 是从左到右计算的，所以位置 *i* 之前的所有位置都已经算过回文半径。假设位置 *i* 以 index 为中心向左对称过去的位置为 *i'*，那么位置 *i'* 的回文半径也是计算过的。那么以 *i'* 为中心的最大回文串大小(pArr[i'])必然只有三种情况，我们依次来分析一下，假设以 *i'* 为中心的最大回文串的左边界和右边界分别记为“左小”和“右小”。

情况一，“左小”和“右小”完全在“左大”和“右大”内部，即以 *i'* 为中心的最大回文串完全在以 index 为中心的最大回文串的内部，如图 9-15 所示。

图 9-15

图 9-15 中，a'是“左小”位置的前一个字符，b'是“右小”位置的后一个字符，b 是 b'以 index 为中心的对称字符，a 是 a'以 index 为中心的对称字符。“左小'”是“左小”以 index 为中心的对称位置，“右小'”是“右小”以 index 为中心的对称位置。如果处在情况一下，那么以位置 *i* 为中心的最大回文串可以直接确定，就是从“右小'”到“左小'”这一段。这是什么原因呢？首先，“左小”到“右小”这一段如果以 index 为回文中心，对应过去就是“右小'”到“左小'”这一段，那么“右小'”到“左小'”这一段就完全是“左小”到“右小”这一段的逆序。同时有“左小”到“右小”这一段又是回文串(以 *i'* 为回文中心)，所以“右小'”到“左小'”这一段一定也是回文串，也就是说，以位置 *i* 为中心的最大回文串起码是“右小'”到“左小'”这一段。另外，以位置 *i'* 为中心的最大回文串只是“右小'”到“左小'”这一段，说明 a'!=b'。那么与 a'相等的 a 也必然不等于与 b'相等的 b，既然 a!=b，说明以位置 *i* 为中心的最大回文串就是“右小'”到“左小'”这一段，而不会扩得更大。

情况一举例如图 9-16 所示。

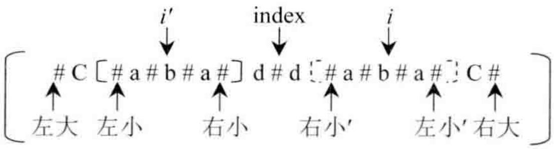

图 9-16

情况二，“左小”和“右小”的左侧部分在“左大”和“右大”的外部，如图 9-17 所示。

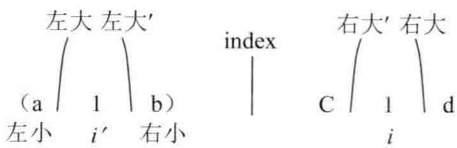

图 9-17

图 9-17 中，a 是“左大”位置的前一个字符，d 是“右大”位置的后一个字符，“左大'”是“左大”以位置 *i'* 为中心的对称位置，“右大'”是“右大”以位置 *i* 为中心的对称位置，b 是“左大'”位置的后一个字符，c 是“右大'”位置的前一个字符。如果处在情况二下，那么以位置 *i* 为中心的最大回文串可以直接确定，就是从“右大'”到“右大”这一段。这是什么原因呢？首先“左大”到“左大'”这一段和“右大'”到“右大”这一段是关于 index 对称的，所以“右大'”到“右大”这一段是“左大”到“左大'”这一段的逆序。同时“左小”到“右小”这一段是回文串(以 *i'* 位置为中心)，那么“左大”到“左大'”这一段也是回文串，所以“左大”到“左大'”这一段的逆序也是回文串，所以“右大'”到“右大”这一段一定是回文串。也就是说，以位置 *i* 为中心的最大回文串起码是“右大'”到“右大”这一段。另外，“左小”到“右小”这一段的是回文串，说明 a==b，b 和 c 关于 index 对称说明 b==c，“左大”到“右大”这一段没有扩得更大，说明 a!=d，所以 d!=c。说明以位置 *i* 为中心的最大回文串就是“右大'”到“右大”这一段，而不会扩得更大。

情况二举例如图 9-18 所示。

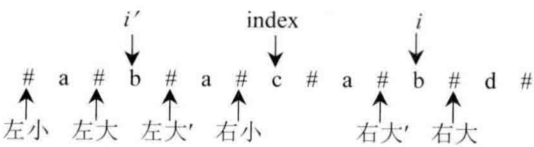

图 9-18

情况三，“左小”和“左大”是同一个位置，即以 i' 为中心的最大回文串压在了以 index 为中心的最大回文串的边界上，如图 9-19 所示。

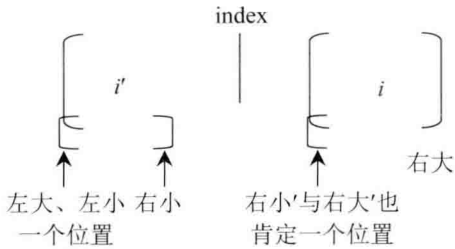

图 9-19

图 9-19 中，“左大”与“左小”的位置重叠，“右小'”是“右小”位置以 index 为中心的对称位置，“右大'”是“右大”位置以 i 为中心的对称位置，可以很容易的证明“右小'”和“右大'”位置也重叠。如果处在情况三下，那么以位置 i 为中心的最大回文串起码是“右大'”和“右大”这一段，但可能会扩得更大。因为“右大'”和“右大”这一段是“左小”和“右小”这一段以 index 为中心对称过去的，所以两段互为逆序关系，同时“左小”和“右小”这一段又是回文串，所以“右大'”和“右大”这一段肯定是回文串，但以位置 i 为中心的最大回文串是可能扩得更大的。比如图 9-20 的例子。

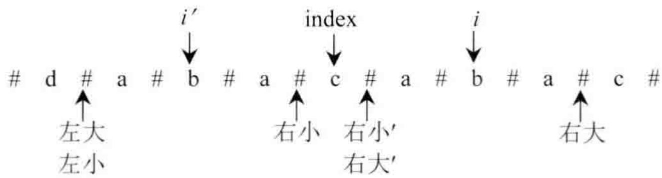

图 9-20

图 9-20 中，以位置 i 为中心的最大回文串起码是“右大'”到“右大”这一段，但可以扩得更大。说明在情况三下，扩出去的过程可以得到优化，但还是无法避免扩出去的检查。

4．按照步骤 3 的逻辑从左到右计算出 pArr 数组，计算完成后再遍历一遍 pArr 数组，找出最大的回文半径，假设位置 i 的回文半径最大，即 pArr[i]==max。但 max 只是 charArr 的最大回文半径，还得对应回原来的字符串，求出最大回文半径的长度（其实就是 max-1）。比如原字符串为"121"，处理成 charArr 之后为"#1#2#1#"。在 charArr 中位置 3 的回文半径最大，最大值为 4（即 pArr[3]==4），对应原字符串的最大回文子串长度为 4-1=3。

Manacher 算法时间复杂度是 $O(N)$ 的证明。虽然我们可以很明显地看到 Manacher 算法

与普通方法相比，在扩出去检查这一行为上有明显的优化，但如何证明该算法的时间复杂度就是 $O(N)$ 呢？关键之处在于估算扩出去检查这一行为发生的数量。原字符串在处理后的长度由 N 变为 $2N$，从步骤 3 的主要逻辑来看，要么在计算一个位置的回文半径时完全不需要扩出去检查，比如，步骤 3 的中 3）介绍的情况一和情况二，都可以直接获得位置 i 的回文半径长度；要么每一次扩出去检查都会导致 pR 变量的更新，比如步骤 3 中的 2）和 3）介绍的情况三，扩出去检查时都让回文半径到达更右的位置，当然会使 pR 更新。然而 pR 最多是从-1 增加到 $2N$（右边界），并且从来不减小，所以扩出去检查的次数就是 $O(N)$ 的级别。所以 Manacher 算法时间复杂度是 $O(N)$。具体请参看如下代码中的 maxLcpsLength 方法。

```java
public int maxLcpsLength(String str) {
	if (str == null || str.length() == 0) {
		return 0;
	}
	char[] charArr = manacherString(str);
	int[] pArr = new int[charArr.length];
	int index = -1;
	int pR = -1;
	int max = Integer.MIN_VALUE;
	for (int i = 0; i != charArr.length; i++) {
		pArr[i] = pR > i ? Math.min(pArr[2 * index - i], pR - i) : 1;
		while (i + pArr[i] < charArr.length && i - pArr[i] > -1) {
			if (charArr[i + pArr[i]] == charArr[i - pArr[i]])
				pArr[i]++;
			else {
				break;
			}
		}
		if (i + pArr[i] > pR) {
			pR = i + pArr[i];
			index = i;
		}
		max = Math.max(max, pArr[i]);
	}
	return max - 1;
}
```

进阶问题。在字符串的最后添加最少字符，使整个字符串都成为回文串，其实就是查找在必须包含最后一个字符的情况下，最长的回文子串是什么。那么之前不是最长回文子串的部分逆序过来，就是应该添加的部分。比如"abcd123321"，在必须包含最后一个字符的情况下，最长的回文子串是"123321"，之前不是最长回文子串的部分是"abcd"，所以末尾应该添加的部分就是"dcba"。那么只要把 manacher 算法稍作修改就可以。具体改成：从左

到右计算回文半径时，关注回文半径最右即将到达的位置（pR），一旦发现已经到达最后(pR==charArr.length)，说明必须包含最后一个字符的最长回文半径已经找到，直接退出检查过程，返回该添加的字符串即可。具体过程参看如下代码中的 shortestEnd 方法。

```
public String shortestEnd(String str) {
	if (str == null || str.length() == 0) {
		return null;
	}
	char[] charArr = manacherString(str);
	int[] pArr = new int[charArr.length];
	int index = -1;
	int pR = -1;
	int maxContainsEnd = -1;
	for (int i = 0; i != charArr.length; i++) {
		pArr[i] = pR > i ? Math.min(pArr[2 * index - i], pR - i) : 1;
		while (i + pArr[i] < charArr.length && i - pArr[i] > -1) {
			if (charArr[i + pArr[i]] == charArr[i - pArr[i]])
				pArr[i]++;
			else {
				break;
			}
		}
		if (i + pArr[i] > pR) {
			pR = i + pArr[i];
			index = i;
		}
		if (pR == charArr.length) {
			maxContainsEnd = pArr[i];
			break;
		}
	}
	char[] res = new char[str.length() - maxContainsEnd + 1];
	for (int i = 0; i < res.length; i++) {
		res[res.length - 1 - i] = charArr[i * 2 + 1];
	}
	return String.valueOf(res);
}
```

KMP 算法

【题目】

给定两个字符串 str 和 match，长度分别为 N 和 M。实现一个算法，如果字符串 str 中含有子串 match，则返回 match 在 str 中的开始位置，不含有则返回-1。