字符串

Repetitions in Strings

transition: fade-out

今日内容

- border
- Z 数组
- 回文子串
- 周期

记号和约定

- 字符串下标从 1 开始。
- 设 x 是字符串，记号 $\left|x\right|$ 表示 x 的长度。
- 符号 ϵ 表示空串。

border

设 x, y 是字符串。若 y 是 x 的前缀也是 x 的后缀，并且 $y \neq x$，就称 y 是 x 的一个 **border**。

- 例：`aabaabaa` 的 border 有 ϵ，`a`，`aa`，`aabaa`。
- 例：`ab` 的 border 有 ϵ。
- 任何非空字符串至少有一个 border，ϵ。
- x 的 border 的 border 也是 x 的 border。

$\mathrm{Border}(x)$

设 x 是非空字符串。定义函数

$$\mathrm{Border}(x) = x \text{ 的最长 } \mathrm{border}.$$

- 例 : $\mathrm{Border}(\texttt{aabaabaa}) = \texttt{aabaa}$
- 例 : $\mathrm{Border}(\texttt{aa}) = \texttt{a}$
- 例 : $\mathrm{Border}(\texttt{a}) = \epsilon$

令 x 是一个非空字符串，m 是使得 $\mathrm{Border}^k(x)$ 有定义的最大整数 k（因此 $\mathrm{Border}^m(x) = \epsilon$）。那么 x 的所有 border，按长度递减顺序列出来，是

$$\mathrm{Border}(x), \mathrm{Border}^2(x), \ldots, \mathrm{Border}^m(x).$$

- 例：`aabaabaa` 的 border 有 `aabaa`，`aa`，`a`，ϵ。

border 数组

设 x 是一个长度为 n 的非空字符串。我们定义数组 $\border$：
$\border[1], \ldots, \border[n]$。
对 $i = 1, \ldots, n$,

$$\border[i] := |\mathrm{Border}(x[1..i])|.$$

我们把数组 $\border$ 称为 x 的 **border 数组**。

- x 的 border 数组记录了 x 的每个前缀的最长 border 的长度。
- $0 \leq \border[i] < i$。

$x =$ aabaabaa 的 border 数组。

i	$x[1..i]$	$\border[i]$
1	a	0
2	aa	1
3	aab	0
4	aaba	1
5	aabaa	2
6	aabaab	3
7	aabaaba	4
8	aabaabaa	5

求 border 数组

给定字符串 x，求 x 的 border 数组。

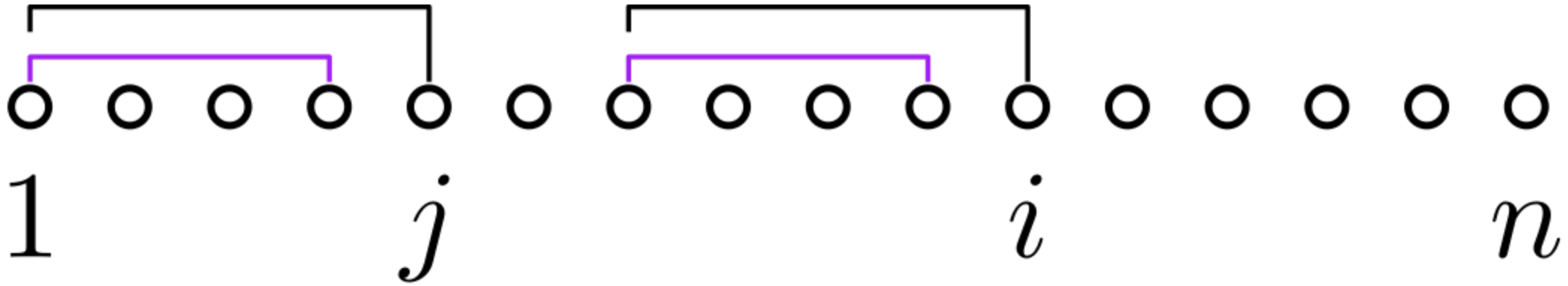
用**递推法**求 x 的 border 数组。

设 x 的长度是 n。

根据定义，$\border[1] = 0$。
对于 $i = 2, \ldots, n$，我们利用 $\border[1], \ldots, \border[i-1]$ 来算出 $\border[i]$。

对于正整数 $j = 1, 2, \ldots, i-1$，我们注意到

$x[1..j]$ 是 $x[1..i]$ 的 border $\iff$ $x[1..j-1]$ 是 $x[1..i-1]$ 的 border 且 $x[j] = x[i]$



{width=60%}

为了求 $\border[i]$，我们从大到小**枚举** $x[1..i-1]$ 的每个 border 的长度 k，检查是否有 $x[k+1] = x[i]$。

枚举 $x[1..i-1]$ 的 border 长度

设 $x[1..i-1]$ 有 m 个 border，那么这 m 个 border 的长度从大到小依次是

$$\border[i-1], \border^2[i-1], \ldots, \border^m[i-1].$$

代码

```
vector<int> border_array(string x) {
  int n = x.size();
  vector<int> border(n);
  border[0] = 0;
  for (int i = 1; i < n; i++) {
    int j = border[i - 1];
    while (j > 0 && x[j] != x[i]) {
      j = border[j - 1];
    }
    if (x[j] == x[i])
      j++;
    border[i] = j;
  }
  return border;
}
```

layout: two-cols-header

时间复杂度

::left::

```cpp
vector<int> border_array(string x) {
  int n = x.size();
  vector<int> border(n);
  border[0] = 0;
  for (int i = 1; i < n; i++) {
    int j = border[i - 1];
    while (j > 0 && x[j] != x[i]) {
      j = border[j - 1];
    }
    if (x[j] == x[i])
      j++;
    border[i] = j;
  }
  return border;
}
```

```cpp
vector<int> border_array(string x) {
  int n = x.size();
  vector<int> border(n);
  border[0] = 0;
  int j = 0;
  for (int i = 1; i < n; i++) {
    while (j > 0 && x[j] != x[i]) {
      j = border[j - 1];
```

设 x, y, z 是字符串。若 z 是 x 的前缀且 z 是 y 的前缀，则称 z 是 x 和 y 的**公共前缀**（common prefix）。x 和 y 的最长公共前缀记作 $\lcp(x, y)$。

- 例：$\lcp(\texttt{ab}, \texttt{ac}) = \texttt{a}$
- 例：$\lcp(\texttt{ba}, \texttt{ac}) = \epsilon$

Z 数组

设 x 是一个长度为 n 的非空字符串。我们定义数组 Z：$Z[2], Z[3], \ldots, Z[n]$。
对 $i = 2, 3, \ldots, n$ ，

$$Z[i] := |\lcp(x, x[i..n])|.$$

我们把数组 Z 称为 x 的 **Z 数组**。

x 的 Z 数组记录了 x 和它自己的每个后缀的最长公共前缀的长度。

我们用不到 $Z[1]$ ，所以干脆不定义它。实际上 $Z[1]$ 也没有合适的定义。
有的资料把 $Z[1]$ 定义为 $\lcp(x, x[1..n])$ ，也就是 x 的长度。有的资料把 $Z[1]$ 定义为 0。

$x =$ aabaabaa 的 Z 数组。

i	$x[i..8]$	$Z[i]$
1	aabaabaa	
2	abaabaa	1
3	baabaa	0
4	aabaa	5
5	abaa	1
6	baa	0
7	aa	2
8	a	1

计算 Z 数组

给定长为 n 的字符串 x，求 x 的 Z 数组。

- 采用暴力比较的方法求 $Z[2]$。

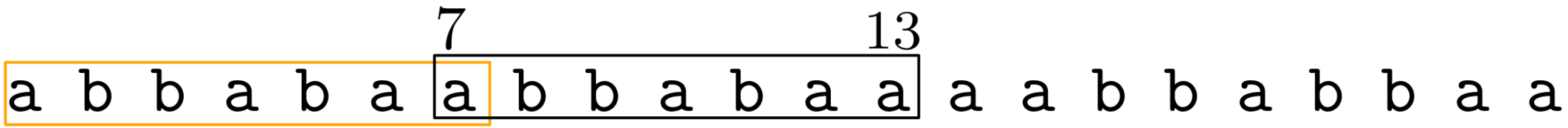
  aabaabaa

  aabaabaa
- 用**递推法**求 $Z[3], \ldots, Z[n]$

$x =$ abbabaabbabaaaabbabbaa。
假设我们已经算出 $Z[2]$ 到 $Z[9]$，现在要计算 $Z[10]$。

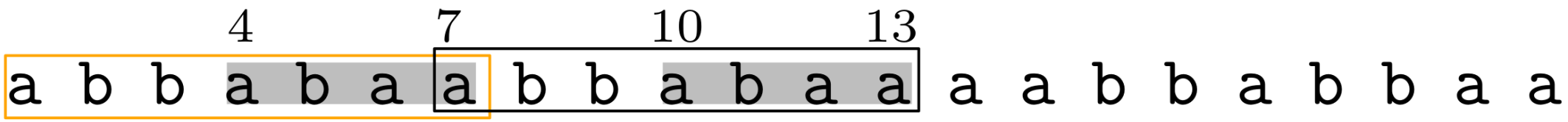
我们做一些**推理**

- 我们知道 $Z[7] = 7$，这意味着 $x[7..13] = x[1..7]$



{width=60%}

- $x[7..13] = x[1..7] \implies x[10..13] = x[4..7]$

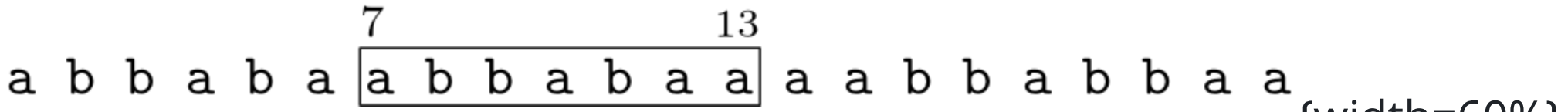


{width=60%}

- 我们又知道 $Z[4] = 2$，这意味着 $x[4..5] = x[1..2]$ 且 $x[6] \neq x[3]$

Z-box

设 x 是一个长为 n 的字符串，对每个 $i = 2, \ldots, n$，我们把区间 $[i, i + Z[i] - 1]$ 称为一个 **Z-box**。



{width=60%}

$Z[7] = 7$，$[7, 13]$ 是一个 Z-box。

Z-box 标志着一个等于前缀的子串。

或者说 Z-box 标志着一个再次出现的前缀。

假设我们已经算出了 $Z[2], \ldots, Z[i-1]$，现在要计算 $Z[i]$。

我们维护目前发现的右端点最大的 Z-box，设它是 $[l, r]$，那么有

$$r = l + Z[l] - 1 \quad \text{且} \quad r = \max_{2 \le j < i}(j + Z[j] - 1)$$

- 如果 $i > r$，那么我们采用暴力比较的方法算出 $Z[i]$。
- 如果 $i \le r$，那么根据 $x[l..r] = x[1..r-l+1]$ 可知

  $$x[i..r] = x[i-l+1..r-l+1]$$

  也就是说，子串 $x[i..r]$ **在前面出现过**。
  - 如果 $Z[i-l+1] < r-i+1$，那么 $Z[i] = Z[i-l+1]$。
  - 如果 $Z[i-l+1] > r-i+1$，那么 $Z[i] = r-i+1$。
  - 如果 $Z[i-l+1] = r-i+1$，那么 $Z[i] \ge r-i+1$。
    我们从 $x[r+1]$ 与 $x[r-i+2]$ 开始继续比较。

```
vector<int> z_array(string x) {
  int n = x.size();
  vector<int> z(n); // 字符串下标从 0 开始。
  int l = 0, r = 0; // [l,r) 是目前发现的右端点最大的 Z-box.
  for (int i = 1; i < n; i++) {
      z[i] = i < r ? min(z[i - l], r - i) : 0;
    while (i + z[i] < n && x[z[i]] == x[i + z[i]])
      z[i]++;
    if (r < i + z[i]) {
      l = i;
      r = i + z[i];
    }
  }
  return z;
}
```

- 根据前一页的分析，只有 `z[i - l] == r - i` 时，第 9 行 `z[i]++` 才可能执行。
- 此时 `z[i]` 的初始值是 `r - i`，所以 `z[i]++` 执行多少次，`r` 就增大多少。
- 最初 `r == 0`，又 `r` 始终不超过 `n`，所以第 9 行 `z[i]++` 执行至多 `n` 次。

字符串匹配

给定长为 m 的字符串 P 和长为 n 的字符串 T，求 P 在 T 里出现的位置。

如果 $T[i..i+m-1]=P$，我们说 P 在 T 的位置 i 出现。

我们称 P 为**模式串**（pattern），称 T 为**文本串**（text）。

我们总是假设 $m \leq n$。

用 border 数组进行字符串匹配

1. 算出文本串 P 的 border 数组。
2. 算一个数组 $a = a[1], \ldots, a[n]$。
   - $a[i]$ 的定义：既是 $T[1..i]$ 的后缀又是 P 的前缀的字符串，最长有多长。
   - P 在 T 的位置 j 出现 $\iff a[j+m-1] = m$

$P =$ a b c a b c a c a b, $m = 1$

$T =$ b a b c b a b c a b c a a b c a b c a b c a c a b c, $n = 2$

$a =$ 0 1 2 3 0 1 2 3 4 5 6 7 1 2 3 4 5 6 7 5 6 7 8 9 10 3.

用**递推法**计算 a 数组

假设我们已经算出 $a[1], \ldots, a[i-1]$，现在要计算 $a[i]$。

计算 $a[4]$.

$P = \texttt{a b c a b c a c a b},$ $m = 10$

$T = \texttt{b a b c b a b c a b c a a b c a b c a b c a c a b c},$ $n = 26$

$a = \texttt{0 1 2 ?}$

- 注意到 $a[4] \le a[3] + 1$.
- 我们先看 $T[4]$ 是否等于 $P[a[3]+1]$，也就是比较 $T[4]$ 和 $P[3]$.
- $T[4] = P[3]$，所以 $a[4] = a[3] + 1 = 3$.

计算 $a[20]$.

$P =$ a b c a b c a c a b, $m = 10$

$T =$ b a b c b a b c a b c a a b c a b c a b c a c a b c, $n = 26$

$a =$ 0 1 2 3 0 1 2 3 4 5 6 7 1 2 3 4 5 6 7 ?

- $a[19] = 7$ 而 $T[20] \neq P[8]$，所以 $a[20] \leq 7$。
- 对于 $j = 1, \ldots, 7$，我们注意到

  $T[20 - j + 1..20] = P[1..j] \iff P[1..j-1]$ 是 $P[1..7]$ 的 border 且 $T[20] = P[j]$
- 借助 P 的 border 数组，我们从大到小枚举 $P[1..7]$ 的每个 border 的长度 k，检查 $T[20]$ 和 $P[k+1]$ 是否相等。
- $\backslash\text{border}[7] = 4$ 而 $T[20] = P[5]$，所以 $a[20] = 5$。

用**递推法**计算 a 数组

假设我们已经算出 $a[1], \ldots, a[i-1]$，现在要计算 $a[i]$。

- 注意到 $a[i] \le a[i-1] + 1$。
- 如果 $t[i] = p[a[i-1]+1]$ 那么 $a[i] = a[i-1] + 1$。
- 否则如果 $a[i] > 0$ 那么 $p[1..a[i]-1]$ 是 $p[1..a[i-1]]$ 的一个后缀，
  也就是说 $p[1..a[i]-1]$ 是 $p[1..a[i-1]]$ 的一个 border。
  - 借助 p 的 border 数组，我们可以从大到小枚举 $p[1..a[i]-1]$ 的每个 border 的长度 k，检查是否有 $p[k+1] = p[i]$。

上面讲的用 border 数组进行字符串匹配的算法，我们称之为 **KMP 算法。**

KMP 是 Knuth，Morris，Pratt 三个姓的缩写。

```cpp
vector<int> kmp_search(string t, string p, const vector<int>& border) {
  vector<int> res;
  int n = t.size();
  int m = p.size();
  int k = 0;
  for (int i = 0; i < n; i++) {
    while (k > 0 && t[i] != p[k])
      k = border[k - 1];
    if (t[i] == p[k])
      k++;
    if (k == m) {
      res.push_back(i - m + 1);
      k = border[k - 1];
    }
  }
  return res;
}
```

```cpp
vector<int> kmp_search(string t, string p, const vector<int>& border) {
  vector<int> res;
  int n = t.size();
  int m = p.size();
  int k = 0;
  for (int i = 0; i < n; i++) {
    while (k > 0 && (k == m || t[i] != p[k])) // 把 k = border[k-1] 的操作写在一处
      k = border[k - 1];
    if (t[i] == p[k])
      k++;
    if (k == m) {
      res.push_back(i - m + 1);
    }
  }
  return res;
}
```

```cpp
vector<int> kmp_search(string t, string p, vector<int> border) {
  vector<int> res;
  int n = t.size();
  int m = p.size();
  int k = 0;
  for (int i = 0; i < n; i++) {
    while (k > 0 && t[i] != p[k]) // p[m] == '\0'，通常 t[i] != '\0'. 也就是说，通常 t[i] != p[m].
      k = border[k - 1];
    if (t[i] == p[k])
      k++;
    if (k == m) {
      res.push_back(i - m + 1);
    }
  }
  return res;
}
```

用 Z 数组进行字符串匹配

1. 算出模式串 P 的 Z 数组。
2. 计算一个数组 $b = b[1], \ldots, b[n]$。
   - $b[i]$ 的定义：$T[i..n]$ 和 P 的最长公共前缀的长度。
   - P 在 T 的位置 j 出现 $\iff a[j] = m$。

$P =$ a b c a b c a c a b, $m = 1$

$T =$ b a b c b a b c a b c a a b c a b c a b c a c a b c, $n = 2$

$b =$ 0 3 0 0 0 7 0 0 4 0 0 1 7 0 0 10 0 0 4 0 0 1 0 3 0 0.

计算数组 b 和计算模式串 P 的 Z 数组的方法是类似的。

```
vector<int> z_search(string p, string t, vector<int> z) {
  vector<int> res;
  int m = p.size();
  int n = t.size();
  int l = 0, r = 0;
  for (int i = 0; i < n; i++) {
    int k = i < r ? min(r - i, z[i - l]) : 0;
    while (k < m && i + k < n && p[k] == t[i + k]) {
      k++;
    }
    if (r < i + k) {
      l = i;
      r = i + k;
    }
    // k 就是我们说的 b[i]
    if (k == m) {
      res.push_back(i);
    }
  }
  return res;
}
```

字符串匹配问题的简便写法

- `border_array()` 和 `kmp_search()` 的代码差不多。
- `z_array()` 和 `z_search()` 的代码也差不多。

实际上，我们可以用 `border_array()` 或 `z_array()` 来解决字符串匹配问题。

1. 用一个在模式串 P 和文本串 T 里都没出现过的特殊字符（比如 #）把 P 和 T 连接起来，得到一个新字符串

$$S = P\#T$$

2. 计算 S 的 border 数组或 Z 数组。
   - $T[i, i+m-1] = P \iff S[i+m+1, i+2m] = P \iff Z[i+m+1]$

.

设 $S = a_1 a_2 \ldots a_n$ 是一个长为 n 的字符串。
我们把 S 倒转过来得到的字符串记作 $\mathrm{rev}S$，也就是 $\mathrm{rev}S := a_n \ldots a_2 a_1$.

如果 $S = \mathrm{rev}S$，就称 S 是**回文串**（palindrome）。

特别的，空串是回文串。

- 例：`noon`，`madam`，`level`，`radar`，`racecar` 是回文串。`apple`，`aab`，`qop` 不是回文串。

给你一个长为 n 的字符串 x。求 x 有多少个子串是回文串。
也就是有多少对整数 l, r 满足：$1 \le l \le r \le n$ 且 $x[l..r]$ 是回文串。

- 例：`aaa` 有 6 个回文子串。`banana` 有 10 个回文子串。

枚举 l, r，判断 $x[l..r]$ 是不是回文串。时间 $O(n^3)$。

回文子串

设 x 是一个长为 n 的字符串。若子串 $x[l..r]$ 是回文串，则称之为 x 的**回文子串**。

回文子串 $x[l..r]$ 的**中心**是 $(l+r)/2$，**半径**是 $\lfloor (r-l+1)/2 \rfloor$。

- 例：`a` 的半径是 0，`aa` 的半径是 1，`aaa` 的半径是 1，`noon` 的半径是 2，`madam` 的半径是 2。

有些资料里，回文子串的半径定义是 $\lceil (r-l+1)/2 \rceil$。

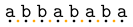
回文中心、回文半径

设 x 是一个长为 n 的字符串。我们把 x 里每个字符的位置称为一个**奇回文中心**，用 $1, 2, \ldots, n$ 表示。

把 x 里相邻两个字符之间的位置称为一个**偶回文中心**，用 $1.5, 2.5, \ldots, n - 0.5$ 表示。

奇偶回文中心统称为**回文中心**。

对于 x 的一个回文中心，我们把以它为中心的最长回文子串的半径称为它的**回文半径**。



{width=200}

黑色的点是奇回文中心，橙色的点是偶回文中心。

回文中心	a		b		b		a		b		a		b		a
回文半径	0	0	0	2	0	0	1	0	2	0	2	0	1	0	1

- 字符串 x 的一个回文中心是 x 里的一个位置。

- 虽然 x 最多有 $O(n^2)$ 个回文子串，但是回文子串的中心只有 $O(n)$ 个。
- 知道了每个回文中心的回文半径，就知道了 x 的全部回文子串。

设 x 是一个长为 n 的字符串，定义数组 $\rad = \rad[1], \ldots, \rad[2n-1]$

$$\rad[i] := \text{回文中心 } (i+1)/2 \text{ 的回文半径.}$$

我们把数组 $\rad$ 称为字符串 x 的**回文半径数组**。

枚举每个回文中心，求它的回文半径。时间 $O(n^2)$。

中心延伸法的程序实现

统一处理奇回文中心和偶回文中心

```
vector<int> palindrome_radius(string s) {
  int n = s.size();
  vector<int> rad(2 * n - 1);
  for (int i = 0; i < 2 * n - 1; i++) {
    int p = i / 2;
    int q = (i + 1) / 2;
    int k = 0;
    while (0 <= q - k - 1 && p + k + 1 < n && s[q - k - 1] == s[p + k + 1]) {
      k++;
    }
    rad[i] = k;
  }
  return rad;
}
```

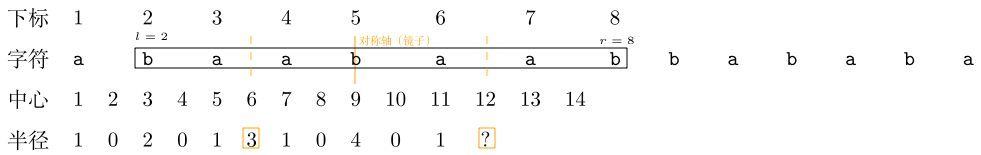
我们寻求更快的计算每个中心的回文半径的方法。

Manacher 算法

- 输入：一个长为 n 的字符串 x。
- 输出：一个长为 $2n-1$ 的数列，即 x 的回文半径数组。
- 时间：$O(n)$。

概述：

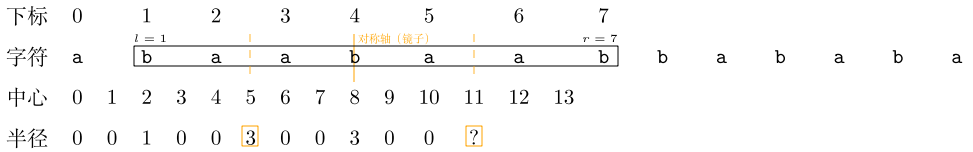
- 从左到右计算每个回文中心的回文半径。
- 维护两个下标 l 和 r 表示已经找到的右端点最大的回文子串的左、右端点。
- 在用中心延伸法求当前中心点的回文半径时，利用 (i) $x[l..r]$ 是回文串和 (ii) 已经求出的回文半径，不检查已知是回文子串的部分。



{width=900}

Manacher 算法的程序实现

下标从 0 开始

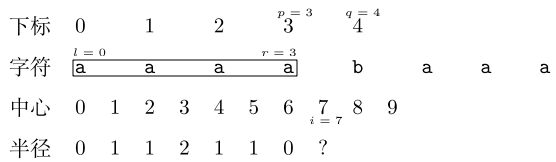


{width=900}

- $l+r$ 就是回文子串 $x[l..r]$ 的中心（的编号），也就是上图中的对称轴。
- 当前回文中心点 i 关于对称轴 $l+r$ 的对称点是 $2(l+r)-i$。

```cpp
vector<int> manacher(string s) {
  int n = s.size();
  vector<int> rad(2 * n - 1);
  int l = 0, r = 0; // l,r 是目前找到的右端点最大的回文子串的左右端点
  for (int i = 0; i < 2 * n - 1; i++) {
    int p = i / 2;
    int q = (i + 1) / 2;
    int k = (r <= p ? 0 : min(r - p, rad[2 * (l + r) - i]));
    while (0 <= q - k - 1 && p + k + 1 < n && s[q - k - 1] == s[p + k + 1])
      k++;
    rad[i] = k;
  }
}
```

第 8 行，`r <= p` 不能写成 `r < p`。因为当 `r == p` 时 `2 * (l + r)` 可能小于 `i`。例如



{width=450}

周期