求最长回文子串与最长重复子串。

长沙雅礼中学 何林

【介绍】

问题的提出：

问题 1 最长回文子串

顺序和逆序读起来完全一样的串叫做回文串。比如 acbca 是回文串，而 abc 不是（abc 的顺序为“abc”，逆序为“cba”，不相同）。

输入长度为 n 的串 S，求它最长回文子串。

问题 2 最长重复子串

如果一个串 x 在 S 中出现，并且 xx 也在 S 中出现，那么 x 就叫做 S 的重复子串。

输入长度为 n 的串 S，求它的最长重复子串。

本文主要涉及“最长回文子串”和“最长重复子串”这两个经典的信息学问题。把他们放在一起讨论，是因为两者的解法具有惊人的类似性。

从算法的最优性上说，两者都存在线性时间复杂度的算法——使用后缀树。无庸置疑，后缀树已经成了优化字符串处理类问题的不二法门。但是它有两个致命缺点。

- **后缀树的时空复杂度和字符串涉及的字符集有直接关系**。称后缀树是“线性数据结构”也是建立在字符集规模为常数的假设上。因此，所谓“线性算法”，准确的说，只是“伪线性”。
- **实践后缀树的编程复杂度极高**。如果说上一点是后缀树在理论上的硬伤，那么这一点就是后缀树在实践上的致命弱点。对时间要求很高的信息学竞赛，是不允许选手花数个小时去编写一个长而容易出错的程序的。最重要的一点是，因为字符集比较大，后缀树的实际运行效果往往不佳，甚至很容易发生空间上的爆炸。

以上两个原因限制了后缀树在竞赛中的应用，虽然它在理论上的价值是不可取代的。

一种折衷的数据结构——后缀数组——可以很好的平衡后缀树的缺点。但是其编程复杂度也不低。要求任意两个串的最长公共前缀，或者用 RMQ 算法、或者用线段树，这两只“老虎”都不是好惹的——几百行的程序一稍不留神就可能满盘皆错。

本文重点介绍的是一个有别于“后缀”系列的全新的算法：分治+扩展的 KMP 算法。它时空复杂度低，编程十分简单，而且算法原理非常好理解。更重要的是其解题思想有很深的可挖掘性。

【扩展的 KMP 算法】

问题的提出：

扩展的 KMP 问题

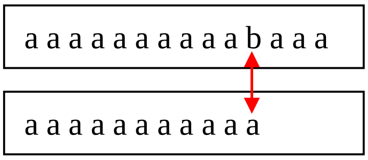
给定母串 S，和子串 T。定义 n=|S|, m=|T|，extend[i]=S[i..n] 与 T 的最长公共前缀长度。请在线性的时间复杂度内，求出所有的 extend[1..n]。

容易发现，如果有某个位置 i 满足 extend[i]=m，那么 T 就肯定在 S 中出现过，并且进一步知道出现首位置是 i——而这正是经典的 KMP 问题。

因此可见“扩展的 KMP 问题”是对经典 KMP 问题的一个扩充和加难。

来看一个例子 S=’aaaaaaaaaabaaa’, T=’aaaaaaaaaaa’。

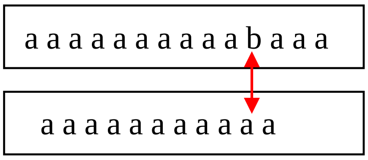
extend[1]=10



红色箭头表示失配
在第 11 个位置失配

这里为了计算 extend[1]，我们进行了 11 次比较运算。

然后我们要算 extend[2]:



红色箭头表示失配
在第 11 个位置失配

extend[2]=9。为了计算 extend[2]，我们是不是也要进行 10 次比较运算呢？不然。

因为通过计算 extend[1]=10，我们可以得到这样的信息：S[1..10]=T[1..10]➔S[2..10]=T[2..10]。

计算 extend[2]的时候，实际上是 S[2]开始匹配 T。因为 S[2..10]=T[2..10]，所以在匹配的开头阶段是“以 T[2..10]为母串，T 为子串”的匹配。

不妨设辅助函数 next[i]表示 T[i..m]与 T 的最长公共前缀长度。

对于这个例子，next[2]=10。也就是说：

T[2..11]=T[1..10]➔T[2..10]=T[1..9]➔S[2..10]=T[1..9]。

这就是说前 9 位的比较是完全可以避免的！我们直接从 S[11]⇔T[10]开始比较。这时候一比较就发现失配，因此 extend[2]=9。

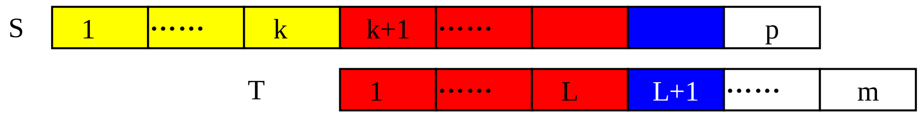
以上的例子是有代表性。下面提出一般的算法。

设 extend[1..k]已经算好，并且在以前的匹配过程中到达的最远位置是 p。最远位置严格的说就是 i+extend[i]-1 的最大值，其中 i=1,2,3,…,k；不妨设这个取最大值的 i 是 a。(下图黄色表示已经求出来了 extend 的位置)

S	1	……	k	k+1			p		

根据定义 S[a..p]=T[1..p-a+1]➔S[k+1..p]=T[k-a+2..p-a+1]，令 L=next[k-a+2]。有两种情况。

第一种情况 k+L<p，如下图：



上面的红色部分是相等的。蓝色部分肯定不相等，否则就违反了“next[i]表示 T[i..m]与 T 的**最长**公共前缀长度”的定义。（因为 next[k-a+2]=L，如果蓝色部分相等的话，那么就有 next[k-a+2]=L+1 或者更大，矛盾）。

这时候我们无需任何比较就可以知道 extend[k+1]=L。同时 a, p 的值都保持不变，k←k+1，继续上述过程。

第二种情况 k+L>=p。如下图：

上图的紫色部分是未知的。因为在计算 extend[1..k]的时候，到达过的最远地方是 p，所以 p 以后的位置从未被探访过，我们也就无从紫色部分是否相等。

这种情况下，就要从 S[p+1]⇔T[p-k+1]开始匹配，直到失配为止。匹配完之后，比较 extend[a]+a 和 extend[k+1]+(k+1)的大小，如果后者大，就更新 a。

整个算法描述结束。

上面的算法为什么是线性的呢？

很容易看出，在计算的过程中，凡是访问过的点，都不需要重新访问了。一旦比较，都是比较以前从不曾探访过的点开始。因此总的时间复杂度是 O(n+m)，是线性的。

还剩下一个问题：next[]这个辅助数组怎么计算？复杂度是多少？

我们发现计算 next 实际上以 T 为母串、T 为子串的一个特殊“扩展的 KMP”。用上文介绍的完全相同的算法计算 next 即可。（用 next 本身计算 next，具体可以参考标准 KMP 或者作者的程序）此不赘述。

请读者认真领会上面算法的思想，即：已经访问过的点绝不再访问，充分利用已经得到的信息。

本文最后还会对此进行一定的总结。

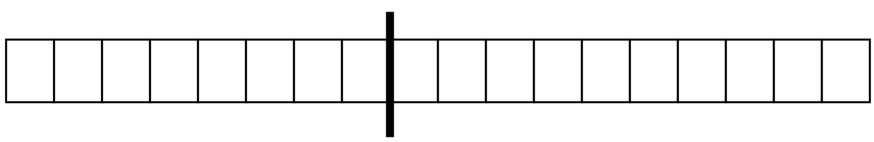
【最长回文子串的分治算法】

问题 1 最长回文子串

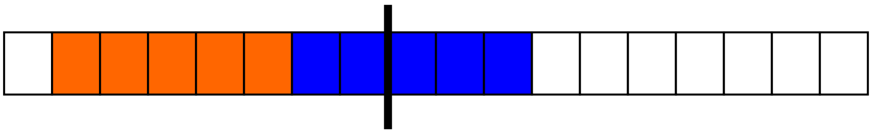
顺序和逆序读起来完全一样的串叫做回文串。比如 acbca 是回文串，而 abc 不是（abc 的顺序为“abc”，逆序为“cba”，不相同）。

输入长度为 n 的串 S，求它最长回文子串。

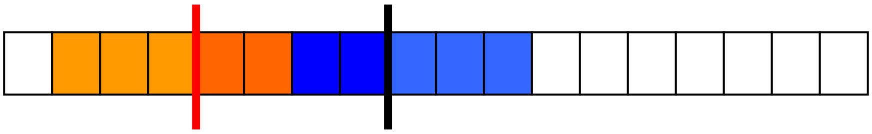
我们把串分成均匀的两部分：



对左边、右边分别递归求最长回文子串，下面的工作只要考虑那些“跨越”粗线的回文串即可。



假设上面是一个回文串，深桔色和深蓝色的部分对称相等。



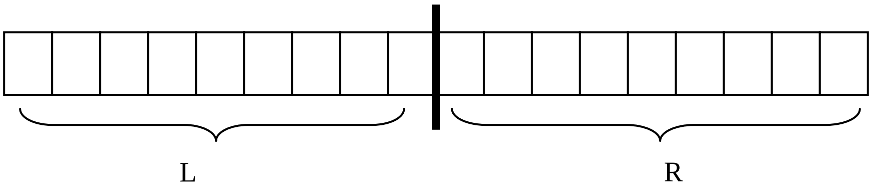
如上，实际上就是桔色和蓝色对称相等、且浅桔色和浅蓝色对称相等。桔色和浅桔色交接的地方（也就是粗红线），称之为这个回文串的“***对称分界点***”。

一个回文串必然满足：

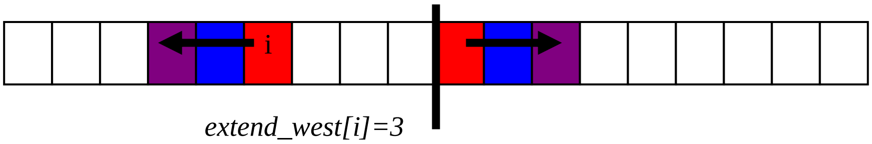
1、对称分界点到二分点（上图两条粗线条之间的部分）之间，是回文。

2、从对称分界点向左扩展、从二分点向右扩展，必须是完全相等的。

设原串为 S，左边的串记为 L、右边的串记为 R。字符串 S 的逆序记为 r(S)。



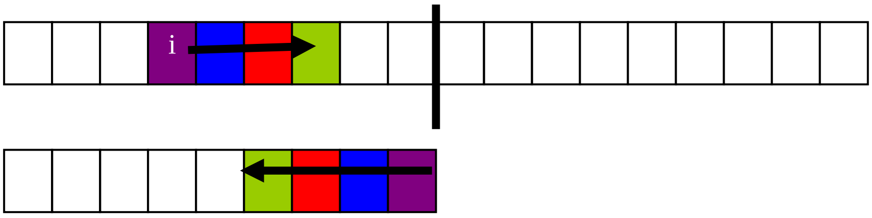
用 extend_west[i]表示第 i 个字符向左，和 R 匹配，最远可以匹配多远。



extend_west[i]=3

显然 ***extend_west[i]就是以 r(L)为母串，R 为子串的“扩展 KMP”。O(n)内可以解决***。

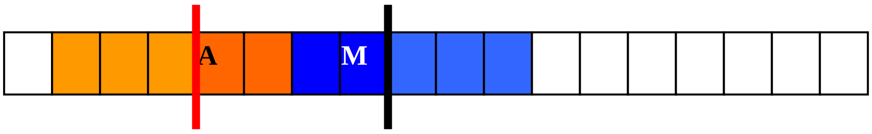
类似的 extend_east[i]表示从第 i 个字符向右扩展，和 r(L)匹配，最远可以匹配多远。



extend_east[i]=4

显然 ***extend_east[i]是以 L 为母串，r(L)为子串的“扩展 KMP”。O(n)内可以解决***。

求出来 extend_west 和 extend_east 有什么用呢？



我们枚举“对称分界点”，设为 A；设二分点为 M。根据条件，只要满足：

extend_east[A]*2>=M-A+1

就肯定存在以 A 为对称分界点的回文串。S[A..M]称为基本回文串。

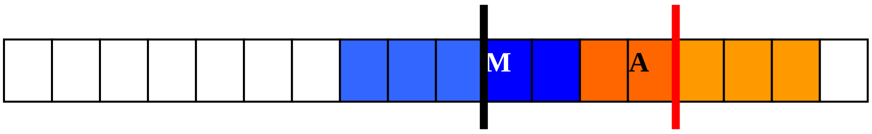
因为要求长度最大，我们在基本回文串的基础上向两边扩展，容易发现扩展的长度就是 extend_west[A-1]（规定 extend_west[0]=0），所以此时的长度：

LENGTH=extend_west[A-1]*2+M-A+1

枚举所有可能的“对称分界点”(总共不超过 n 个)，对每个分界点，根据上面的分析，只要用 O(1)的时间复杂度就能判定它的合法性、以及求出以该点为分界点时的最大回文串长度。

最后取最大值即可。

注意到上面的讨论中，回文串的“重心”在 L 中——所谓重心就是回文串中点。所以“对称分界点”也在 L 中。实际上还有可能是下面的情况：



类似处理即可，此不赘述。

以上我们就在 O(n)的时间复杂度内（n=|S|），求出了跨越“二分点”的最长回文串长度。

分析一下时间复杂度。设计算长度为 n 的串的时间复杂度是 f(n)，一个粗略的递推可以写成：

$f(n)=2f(n/2)+n$ (其中 n 是求跨越二分点的最长回文子串的复杂度，f(n/2)分别是递归处理两边的复杂度)

$f(1)=1$

很容易算出：

$f(n)\sim nlogn$

也就是说该算法复杂度是 O(nlogn)。

【最长重复子串的分治算法】

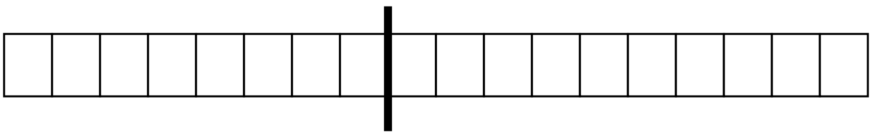
问题 2 最长重复子串

如果一个串 x 在 S 中出现，并且 xx 也在 S 中出现，那么 x 就叫做 S 的重复子串。

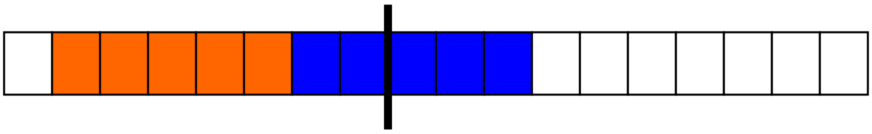
输入长度为 n 的串 S，求它的最长重复子串。

有了最长回文串的解题基础，研究最长重复子串就要简单多了。

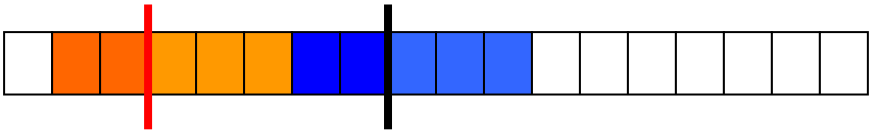
我们把串分成均匀的两部分：



对左边、右边分别递归求最长重复子串，下面的工作只要考虑那些“跨越”粗线的重复子串即可。

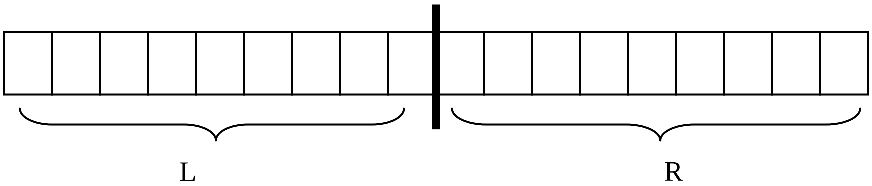


假设深桔色和深蓝色的部分完全相等。（也就是说存在一个长度为 5 的重复子串）

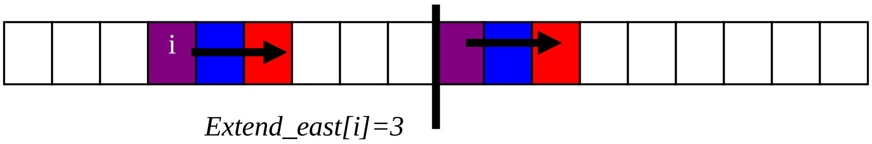


如上，实际上就是桔色和蓝色相等、且浅桔色和浅蓝色相等。桔色和浅桔色交接的地方（也就是粗红线），称之为这个重复子串的“***重复分界点***”。(重复分界点到二分点的距离，实际上就是重复子串的长度)

设原串为 S，左边的串记为 L、右边的串记为 R。字符串 S 的逆序记为 r(S)。



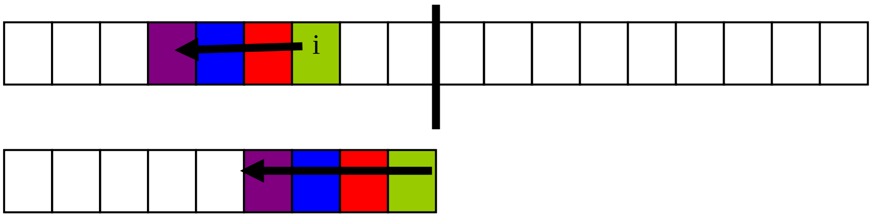
用 extend_east[i]表示第 i 个字符向右，和 R 匹配，最远可以匹配多远。



Extend_east[i]=3

显然 ***extend_east[i]就是以 L 为母串，R 为子串的“扩展 KMP”。O(n)内可以解决***。

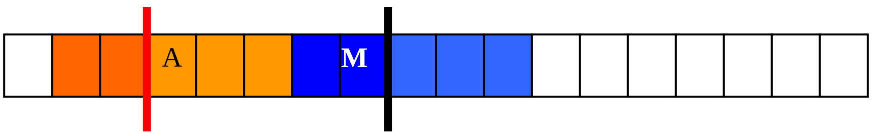
类似的 extend_west[i]表示从第 i 个字符向左扩展，和 r(L)匹配，最远可以匹配多远。



extend_west[i]=4

显然 ***extend_west[i]是以 r(L)为母串，r(L)为子串的“扩展 KMP”。O(n)内可以解决***。

求出来 extend_west 和 extend_east 有什么用呢？



我们枚举“重复分界点”，设为 A；设二分点为 M。根据条件，只要满足：

extend_east[A]+extend_west[A-1]>=M-A+1

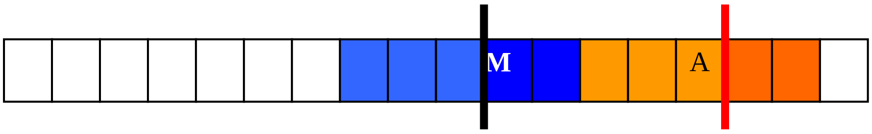
就肯定存在以 A 为重复分界点的重复子串。此时的串长度是

LENGTH=M-A+1

枚举所有可能的“重复分界点”（总共不超过 n 个），对每个分界点，根据上面的分析，只要用 O(1)的时间复杂度就能判定它的合法性、以及求出以该点为分界点时的最大重复串长度。

最后取最大值即可。

注意到上面的讨论中，重复子串是“偏左”的，实际上还有可能“偏右”，如下：



类似处理即可，此不赘述。

至此问题 2——最长重复子串——也解决了。时间复杂度和第一个问题相同，也是 O(nlogn)。

【优化的本质——减少冗余】

我们回顾一下扩展的 KMP，它为什么高效？

在计算 extend[1..k]的时候，已经可以得到一些关于 S 和 T 的信息；在计算 extend[k+1]的时候，正是充分利用了之前得到的信息，将所有可以避免的比较都避免了，所以最后得到了一个很高效的算法。

再看求最长回文子串。我们很容易提出这样一个算法：枚举回文串的中点，然后向两边扩展。

这个算法的复杂度是 O(n^2)，远远高于 O(nlogn)。它到底差在哪？

比如 S=’aaaaaaaaaa……’。

当以倒数第二个 a 为中点，’aaaaaaaaaa……’，实际就是要从’aaaaaaaaaa……’中的两个蓝色字母开始分别向两边扩展，求最大扩展长度。

当以倒数第三个 a 为中点，’aaaaaaaaaa……’，实际就是要从’aaaaaaaaaa……’中的两个蓝色字母开始分别向两边扩展，求最大扩展长度。

……

最后归纳一下就是：

aaaaaaaaaa……

对每一个蓝的 a，都要求一次从红色的 a 向右、蓝色的 a 向左，最多可以扩展多远。因为采用的纯枚举，计算这个的复杂度是 O(n^2)。

但是我们可以轻松、而有点震惊的发现，这实际上就是一个“扩展的 KMP”问题！母串是蓝色部分的逆序串，子串是红色的 a 及其右边的所有字符。

对于这样一个标准的“扩展 KMP”，该算法采用的实际上是暴力穷举，这样的效率如何能不低！诚如上面的分析，这种暴力穷举等于是放弃了任何已经得到的有用信息。

从令一个角度说，二分法+扩展 KMP 算法之所有能够优秀的解决最长回文串

问题，也正是因为**减少了计算的冗余，充分利用了已知**。

【分治法提高效率的本质——集中类似状态】

前面我们说明了“充分利用已知信息”的重要性。但是很多时候情况并不尽如人意。

还是以最长回文子串为例。尽管我们知道了比较中存在大量冗余，我们又能做什么呢？

一种自然的思路就是定义一个新的数组 extend[i, j]，表示从第 i 位向左、第 j 位向右，最多可以扩展多远。

这样做是没有冗余的——每个 extend[i, j]的计算，都已经完全充分的利用到了已知信息。但是稍微想一下就知道这种做法多么愚蠢——仅仅把所有的 extend[i, j]算出来，就需要 $O(n^2)$的时间复杂度！

这种方法又是为什么低效呢？

原因是：**计算信息量太大，得到了许多无用信息。**

事实上我们并不需要把所有的 extend[i, j]都计算出来，只要一少部分就能解决问题了。

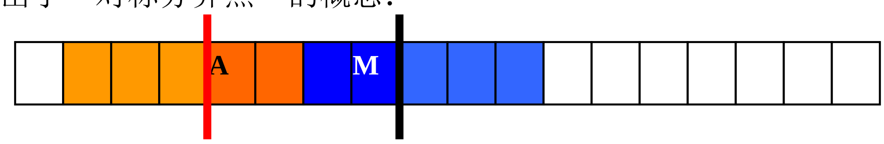
我们并不知道哪些信息是需要的、哪些信息是互相关联的，这就涉及到一个**利用信息成本**的问题。

比如计算 extend[1,x], extend[2,x],……,extend[x-1,x]，采用扩展的 KMP 算法，只要 O(n)的复杂度。因为 1, 2, ……, x 是连续自然数，我们可以很轻松的分析他们之间的关系，从而以很低的信息利用成本解决问题——这导致的就是高效的算法。

但是如果给出的是一大堆杂乱无章的数字，比如 extend[1, 10], extend[2,9], extend[7, 25], extend[8, 19]，那么我们就必须设法寻找它们之间的联系、寻找到底要如何才能利用已知信息来推导未知信息——这种寻找要付出成本——或者说他们之间因为杂乱无章，根本没关系，除了暴力穷举没有其他方法——这时候利用信息成本可以看作正无穷。

二分法正是一个很好的降低信息利用成本的方法。

我们回忆一下二分法，它一开始就把字符串分成了均匀的两部分。在然后的分析中给出了“对称分界点”的概念：



肯定有读者心里犯嘀咕“平白无故怎么造出一个分界点来了？”

实际上造出分界点之后，就可以得到：

1、两个深色部分对称。

2、两个浅色部分对称。

其中深蓝色是 r(L)的前缀、浅蓝色是 R 的前缀。这样问题就能转化成为求“某个串的***每一个***后缀与另一个串的最长公共前缀”——这正是“扩展 KMP”解决的问题。（请注意“每一个”。所谓“每一个”实际上等价于前面提到的“计算 extend[1,x], extend[2,x],……,extend[x-1,x]”。因为是连续自然数，所以可以用高效的扩展 KMP）

一条粗黑的二分线，创造出了两个蓝色的前缀——前缀进一步转化为扩展的KMP——这就是二分法高效的本质。

它从凌乱的状态中整理出了关系紧密的状态，这些“紧密的关系”满足了“扩展 KMP”解决问题的前提，使得问题可以在一个比较低的信息利用成本下，得到很好的解决——导致出高效的算法。

【关于信息利用的一些思考】

一个问题，给定的信息越多、已知的条件越多、解决它的复杂度就越低。

反过来为了降低复杂度，我们就要想法设法“创造条件”。

创造条件的第一条路是合理的组织。就是本文主要讨论的，把有***关联的、有用的***状态组织起来，分析他们的联系，从而充分的利用已经得到的信息。

第二条路是部分枚举。通过部分枚举创造已知信息，为以后的解题打开方便之门。本届集训队员楼天成的论文《部分搜索与匹配》就对这方面进行了研究和总结。

【鸣谢】

感谢栗师在最长回文子串问题上对我的大力帮助。
感谢楼天成在最长重复子串问题上对我的大力帮助。
感谢许智磊在后缀数组上对我的大力帮助。
感谢林希德在后缀树上的对我的大力帮助。
感谢饶向容在扩展的 KMP 算法上对我的大力帮助。