**数据：**是描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合。

**数据元素：**是组成数据的、有一定意义的基本单位，再计算机中通常作为整体处理。也被称为记录。

**数据项：**一个数据元素可以由若干个数据项组成。数据项是不可分割的最小单位。

**数据对象：**是性质相同的数据元素的集合，是数据的子集。

****

**数据结构：是相互之间存在一种或多种特定关系的数据元素的集合。**

按照视点的不同，我们把数据结构分为**逻辑结构**和**物理结构**。



**逻辑结构**：是指数据对象中数据元素之间的相互关系。

逻辑结构分为以下四种：集合结构、线性结构、树形结构、图形结构。

1. 集合结构：集合结构中的数据元素除了同属于一个集合外，它们之间没有其他关系。
2. 线性结构：线性结构中的数据元素之间是一对一的关系。
3. 树形结构：树形结构中的数据元素之间存在一种一对多的层次关系。
4. 图形结构：图形结构的数据元素是多对多的关系。

**物理结构**：是指数据的逻辑结构在计算机中的存储形式。

数据元素的存储结构形式有两种：**顺序存储**和**链式存储**。

**顺序存储结构**：是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。

**链式存储结构**：是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的。

逻辑结构是面向问题的，而物理结构就是面向计算机的，其基本的目标就是将数据及其逻辑关系存储到计算机的内存中。

**数据类型**：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

抽象数据类型(Abstract Data Type，ADT)：是指一个数学模型及定义在该模型上的一组操作。

**算法**是解决特定问题求解步骤的描述，再计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

**算法具有五个基本特性**：输入、输出、有穷性、确定性和可行性。

**算法设计的要求**：正确性、可读性、健壮性、时间效率高和存储量低。

**算法效率的度量方法**：事后统计方法(不科学、不准确)、事前分析估算方法。

**函数的渐进增长**：给定两个函数f(n)和g(n)，如果存在一个整数N，使得对于所有的n>N，f(n)总是比g(n)大，那么，我们说f(n)的增长渐近快于g(n)。

判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项(最高阶项)的阶数。

某个算法，随着n的增大，它会越来越优于另一算法，或者越来越差于另一算法。这其实就是事前估算方法的理论依据，通过算法时间复杂度来估算算法时间效率。

在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级。算法的时间复杂度，也就是算法的时间量度，记作：T(n)=O(f(n))。它表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐进时间复杂度，简称为时间复杂度。其中f(n)是问题规模n的某个函数。

这样用大写O()来体现算法时间复杂度的记法，我们称之为大O记法。





**算法的空间复杂度**通过计算算法所需的存储空间实现，算法空间复杂度的计算公式记作：S(n)=O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数。

计算机中的常用字符是使用标准的ASCII编码，更准确一点，由7位二进制数表示一个字符，总共可以表示128个字符。后来发现一些特殊符号的出现，128个不够用，于是扩展ASCII码由8位二进制数表示一个字符，总共可以表示256个字符。

Unicode编码比较常用的是由16位的二进制数表示一个字符，这样总共就可以表示216个字符，约是65万多个字符。为了个ASCII码兼容,Unicode的前256个字符与ASCII码完全相同。

**KMP模式匹配算法**

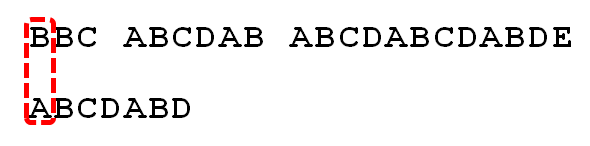
字符串匹配是计算机的基本任务之一。

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？

许多算法可以完成这个任务，Knuth-Morris-Pratt算法（简称KMP）是最常用的之一。它以三个发明者命名，起头的那个K就是著名科学家Donald Knuth。

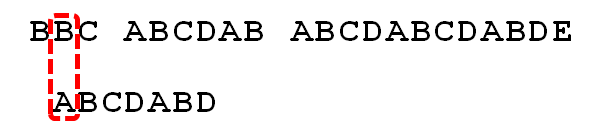
这种算法不太容易理解，网上有很多解释，但读起来都很费劲。直到读到[Jake Boxer](http://jakeboxer.com/blog/2009/12/13/the-knuth-morris-pratt-algorithm-in-my-own-words/)的文章，我才真正理解这种算法。下面，我用自己的语言，试图写一篇比较好懂的KMP算法解释。

1.



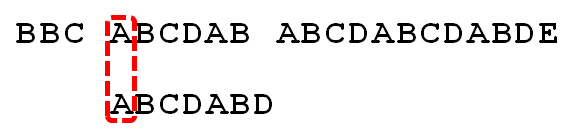
首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。

2.



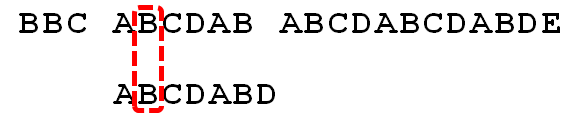
因为B与A不匹配，搜索词再往后移。

3.



就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

4.



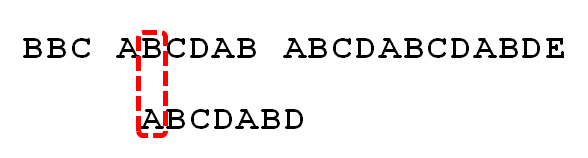
接着比较字符串和搜索词的下一个字符，还是相同。

5.



直到字符串有一个字符，与搜索词对应的字符不相同为止。

6.



这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

7.



一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

8.



怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

9.

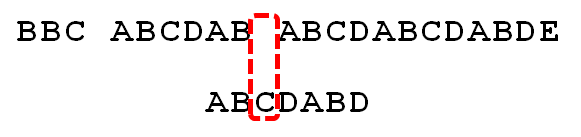


已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，因此按照下面的公式算出向后移动的位数：

**移动位数 = 已匹配的字符数 - 对应的部分匹配值**

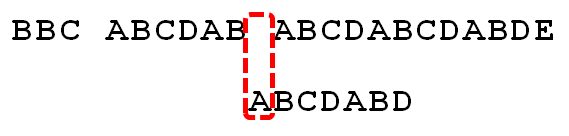
因为 6 - 2 等于4，所以将搜索词向后移动4位。

10.



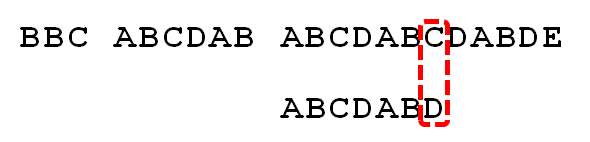
因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（"AB"），对应的"部分匹配值"为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移2位。

11.



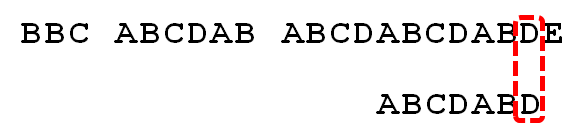
因为空格与A不匹配，继续后移一位。

12.



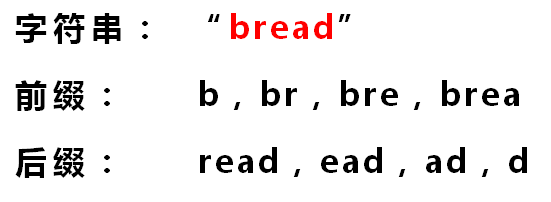
逐位比较，直到发现C与D不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。

13.



逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。

14.



下面介绍《部分匹配表》是如何产生的。

首先，要了解两个概念："前缀"和"后缀"。 "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

15.



"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

*－　"A"的前缀和后缀都为空集，共有元素的长度为0；*

*－　"AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；*

*－　"ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；*

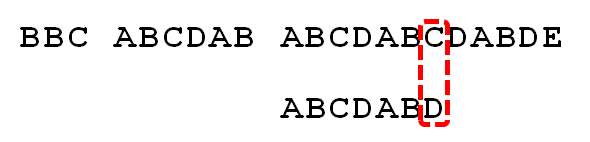
*－　"ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；*

*－　"ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；*

*－　"ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；*

*－　"ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。*

16.



"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2（"AB"的长度）。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

节点拥有的子树数称为节点的度(Degree)。

树的度是树内各节点的度的最大值。



树中节点的最大层次称为树的深度(Depth)或高度。



如果将树中节点的各子树看成从左往右是有次序的，不能互换的，则称该树为有序树，否则称为无序树。

森林(Forest)是m(m>=0)颗互不相交的树的集合。对树中每个节点而言，其子树的集合即为森林。

树的存储结构的表示：双亲表示法、孩子表示法、孩子兄弟表示法。

所有节点都只有左子树的二叉树叫左斜树。所有节点都只有右子树的二叉树叫右斜树。这两者统称为斜树。

在一棵二叉树中，如果所有分支节点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。



对一颗具有n个节点的二叉树按层序编号，如果编号为i(1<=i<=n)的节点与同样深度的满二叉树中编号为i的节点在二叉树中的位置完全相同，则这颗二叉树称为完全二叉树，如图6-5-6所示。



二叉树性质

性质1：在二叉树 的第i层上至多有个节点(i>=1)。

性质2：深度为k的二叉树至多有个节点(k>=1)。

性质3：对任何一颗二叉树T，如果其终端节点数为，度为2的节点数为，则。

性质4：具有n个节点的完全二叉树的深度为(表示不大于x的最大整数)。

性质5：

二叉树遍历方法：前序遍历、中序遍历、后序遍历

赫夫曼树

Prim算法和Kruskal算法都能从连通图找出最小生成树。区别在于Prim算法是挨个找，而Kruskal是先排序再找。

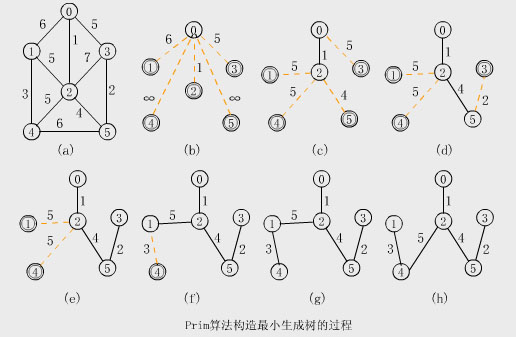
 一、Prim算法：

    Prim算法实现的是找出一个有权重连通图中的最小生成树，即：具有最小权重且连接到所有结点的树。(强调的是树，树是没有回路的)。

    Prim算法是这样来做的：

    首先以一个结点作为最小生成树的初始结点，然后以迭代的方式找出与最小生成树中各结点权重最小边，并加入到最小生成树中。加入之后如果产生回路则跳过这条边，选择下一个结点。当所有结点都加入到最小生成树中之后，就找出了连通图中的最小生成树了。

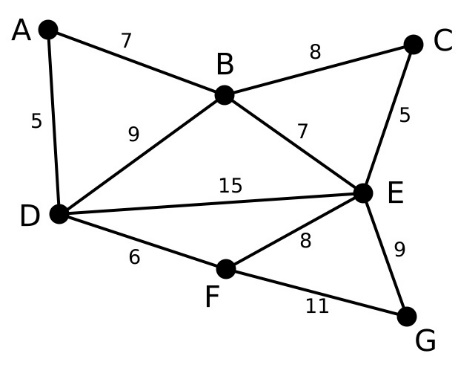
     Prim算法最小生成树查找过程：

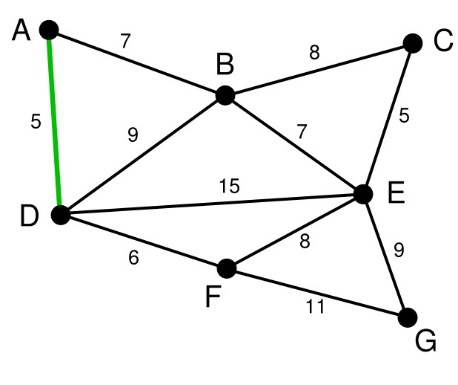


二、Kruskal算法：

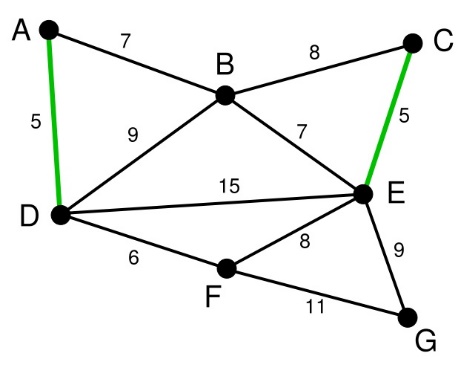
    Kruskal算法与Prim算法的不同之处在于，Kruskal在找最小生成树结点之前，需要对所有权重边做从小到大排序。将排序好的权重边依次加入到最小生成树中，如果加入时产生回路就跳过这条边，加入下一条边。当所有结点都加入到最小生成树中之后，就找出了最小生成树。

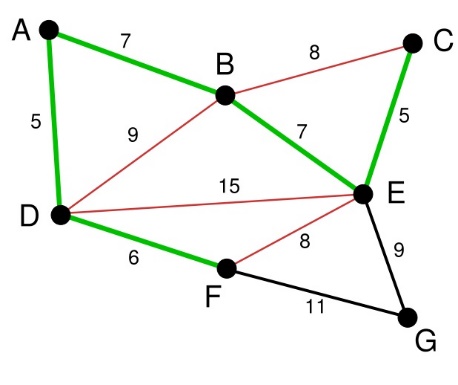
图例描述：

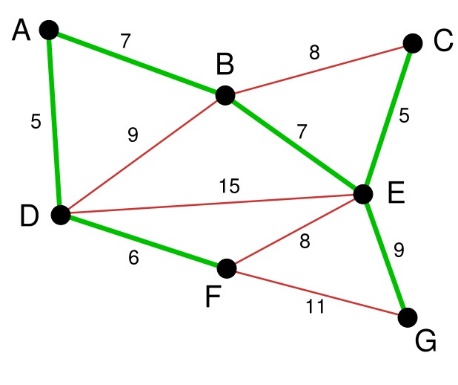
首先第一步，我们有一张图Graph，有若干点和边



将所有的边的长度排序，用排序的结果作为我们选择边的依据。这里再次体现了贪心算法的思想。资源排序，对局部最优的资源进行选择，排序完成后，我们率先选择了边AD。这样我们的图就变成了右图

在剩下的变中寻找。我们找到了CE。这里边的权重也是5

依次类推我们找到了6,7,7，即DF，AB，BE。



下面继续选择， BC或者EF尽管现在长度为8的边是最小的未选择的边。但是现在他们已经连通了（对于BC可以通过CE,EB来连接，类似的EF可以通过EB,BA,AD,DF来接连）。所以不需要选择他们。类似的BD也已经连通了（这里上图的连通线用红色表示了）。

最后就剩下EG和FG了。当然我们选择了EG。

