Brute Force (暴力匹配) O(mn).

逐个暴力匹配，匹配不上则模式串下移一位，匹配串重来.

Algorithm: BruteForce(T, P):

```
n ← len(T), m ← len(P)
for i ← 0 to n-m-1
    for j ← 0 to m-1
        if P[j] != T[i+j] then
            break;
    if j = m-1
        pattern occurs with shift i
```

Rabin-Karp Algorithm (Hashing) O(mn)但表现比暴力好，通常能有 O(m+n).

先比较模数，若相等则逐个比较，否则不匹配

(对char处理常用ASCII)

Algorithm: Rabin-Karp(T, P, d, q):

d ← 基数, q ← 模数

```
n ← len(T), m ← len(P)
h ← d^(m-1) (mod q), p ← 0, t0 ← 0
for j ← 0 to m-1
    p ← (dp + P[j]) mod q,
    t_0 ← (dt_0 + T[j]) mod q
for i ← 0 to n-m
    if p != t_i then
        t_(i+1) ← (d(t_i-T[i]h)+T[i+m]) mod q
    else
        If P[0,..m-1]=T[i,i+m-1]
            pattern occurs with shift i
        Else
            t_(i+1) ← (d(t_i-T[i]h)+T[i+m]) mod q
```

(Lines 3–5: ⇒计算前m个表示数的模.)

(Line 8: 移位 →; line 13: 移位 →)

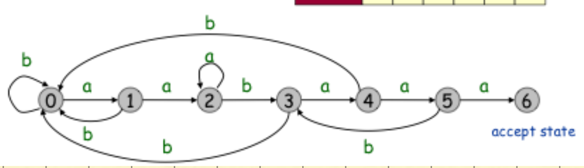
Finite State Automata (FSA)

- A finite State automaton is defined by:
  - *Q, a set of states*
  - $q_0 \in Q$*, the start state*
  - $A \subseteq Q$*, the accepting states*
  - Σ*, the input alphabet*
  - δ*, the transition function, from* $Q \times \Sigma$ *to Q*

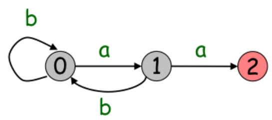
Search Pattern

a	a	b	a	a	a

	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3
next	0	0	2	0	0	3



	0	1
a	1	2
b	0	0



Transition function (状态转移函数)

一般用二维数组模拟.

dp[i][m] = j

表示在状态i处若遇到字符m则到状态j.

(对char的处理一般通过ASCII转成int)

Algorithm: Transition(P, Σ):

```
m ← len(P)
X ← 0
Initialize δ(0,a) for each a ∈ Σ
for j ← 1 to m-1
    for each character a ∈ Σ
        if P[j+1] = a then   // char match
            δ(j,a) ← j + 1
        else                 // char mismatch
            δ(j,a) ← δ(X,a)
    X ← δ(X,P[j+1])
return δ
```

FSA algorithm: $O(|\Sigma| \times m+n) = O(C \times m+n) = O(m+n)$

Algorithm: FSA(T, P):

```
1. n ← len(T), m ← len(P)
2. δ ← Transition(P, Σ)
3. q ← 0 // q is the state of the FSA.
4. for i ← 1 to n
5.       q ← δ(q,T[i])
6.       if q = m
7.             pattern occurs with shift i - m
```

Prefix Function (前缀函数 nextArray): 表示第 i 个字符之前的最长公共前后缀

given $P[1..m]$, the prefix function π for P is π : {1, 2 ..., m} -> {0, 1, ..., m-1} such that:

$\pi[i]=max\{k, k<i \text{ and } P[1,..,k]= P[i-k+1,...,i]\}$

Algorithm: NextArray(P):

```
1. m ← len(P)
2. Let π[1,…,m] be a new array
3. π[1] = 0, k ← 0
4. for q = 2 to m
5.       while k > 0 and P[k+1] != P[q]
6.             k ← π[k]
7.       if P[k+1] = P[q]
8.             k ← k + 1
9.       π[q] ← k
10. return π
```

在P中找最长的后缀与前缀相同.

KMP algorithm: $O(m+n)$

Algorithm: KMP(T, P):

```
1. n ← len(T), m ← len(P)
2. π ← NextArray(P)
3. q ← 0
4. for i = 1 to n
5.       while q > 0 and P[q+1] != T[i]
6.             q ← π[q]
7.       if (P[q+1] = T[i])
8.             q ← q + 1
9.       if q == m
10.            print "Pattern occurs with shift" i-m
11.            q ← π[q]
```

KMP的标定数组可用 nextArray, 也可用FSA. 最重要的是若匹配不上, 则要知道去回到何处.

最长回文子串

①中心扩散法：$O(N^2)$

```java
public class Solution {

    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2) {
            return s;
        }
        int maxLen = 1;
        String res = s.substring(0, 1);
        // 中心位置枚举到 len - 2 即可
        for (int i = 0; i < len - 1; i++) {
            String oddStr = centerSpread(s, i, i);
            String evenStr = centerSpread(s, i, i + 1);
            String maxLenStr = oddStr.length() > evenStr.length() ? oddStr : evenStr;
            if (maxLenStr.length() > maxLen) {
                maxLen = maxLenStr.length();
                res = maxLenStr;
            }
        }
        return res;
    }

    private String centerSpread(String s, int left, int right) {
        // left = right 的时候，此时回文中心是一个空隙，回文串的长度是奇数
        // right = left + 1 的时候，此时回文中心是任意一个字符，回文串的长度是偶数
        int len = s.length();
        int i = left;
        int j = right;
        while (i >= 0 && j < len) {
            if (s.charAt(i) == s.charAt(j)) {
                i--;
                j++;
            } else {
                break;
            }
        }
        // 这里要小心，跳出 while 循环时，恰好满足 s.charAt(i) != s.charAt(j)，因此不能取 i，不能取 j
        return s.substring(i + 1, j);
    }
}
```

②Manacher：

```java
/**
 * 创建预处理字符串
 *
 * @param s       原始字符串
 * @param divide 分隔字符
 * @return 使用分隔字符处理以后得到的字符串
 */
private String addBoundaries(String s, char divide) {
    int len = s.length();
    if (len == 0) {
        return "";
    }
    if (s.indexOf(divide) != -1) {
        throw new IllegalArgumentException("参数错误，您传递的分割字符，在输入字符串中存在！");
    }
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < len; i++) {
        stringBuilder.append(divide);
        stringBuilder.append(s.charAt(i));
    }
    stringBuilder.append(divide);
    return stringBuilder.toString();
}
```

下面是算法的实现，注意，为了避免更新P的时候导致越界，我们在字符串T的前增加一个特殊字符，比如说'$',所以算法中字符串是从1开始的。、

```java
string longestPalindrome(string s)
{
    string manaStr = "$#";
    for (int i=0;i<s.size();i++) //首先构造出新的字符串
    {
      manaStr += s[i];
      manaStr += '#';
    }
    vector<int> rd(manaStr.size(), 0);//用一个辅助数组来记录最大的回文串长度，注意这里记录的是新串的长度，原串的长度要减去1
    int pos = 0, mx = 0;
    int start = 0, maxLen = 0;
    for (int i = 1; i < manaStr.size(); i++)
    {
      rd[i] = i < mx ? min(rd[2 * pos - i], mx - i) : 1;
      while (i+rd[i]<manaStr.size() && i-rd[i]>0 && manaStr[i + rd[i]] == manaStr[i - rd[i]])//这里要注意数组越界的判断，源代码没有注意，release下没有报错
        rd[i]++;
      if (i + rd[i] > mx) //如果新计算的最右侧端点大于mx,则更新pos和mx
      {
        pos = i;
        mx = i + rd[i];
      }
      if (rd[i] - 1 > maxLen)
      {
        start = (i - rd[i]) / 2;
        maxLen = rd[i] - 1;
      }
    }
    return s.substr(start, maxLen);
}
```

一般：$O(N^2)$

```java
public String longestPalindrome(String s) {
    int len = s.length();
    if (len < 2) {
        return s;
    }
    String str = addBoundaries(s, '#');
    int sLen = 2 * len + 1;
    int maxLen = 1;

    int start = 0;
    for (int i = 0; i < sLen; i++) {
        int curLen = centerSpread(str, i);
        if (curLen > maxLen) {
            maxLen = curLen;
            start = (i - maxLen) / 2;
        }
    }
    return s.substring(start, start + maxLen);
}

private int centerSpread(String s, int center) {
    // left = right 的时候，此时回文中心是一个空隙，回文串的长度是奇数
    // right = left + 1 的时候，此时回文中心是任意一个字符，回文串的长度是偶数
    int len = s.length();
    int i = center - 1;
    int j = center + 1;
    int step = 0;
    while (i >= 0 && j < len && s.charAt(i) == s.charAt(j)) {
        i--;
        j++;
        step++;
    }
    return step;
}
```

优化：$O(N)$

```java
public String longestPalindrome(String s) {
    // 特判
    int len = s.length();
    if (len < 2) {
        return s;
    }

    // 得到预处理字符串
    String str = addBoundaries(s, '#');
    // 新字符串的长度
    int sLen = 2 * len + 1;

    // 数组 p 记录了扫描过的回文子串的信息
    int[] p = new int[sLen];

    // 双指针，它们是一一对应的，须同时更新
    int maxRight = 0;
    int center = 0;

    // 当前遍历的中心最大扩散步数，其值等于原始字符串的最长回文子串的长度
    int maxLen = 1;
    // 原始字符串的最长回文子串的起始位置，与 maxLen 必须同时更新
    int start = 0;

    for (int i = 0; i < sLen; i++) {
        if (i < maxRight) {
            int mirror = 2 * center - i;
            // 这一行代码是 Manacher 算法的关键所在，要结合图形来理解
            p[i] = Math.min(maxRight - i, p[mirror]);
        }

        // 下一次尝试扩散的左右起点，能扩散的步数直接加到 p[i] 中
        int left = i - (1 + p[i]);
        int right = i + (1 + p[i]);

        // left >= 0 && right < sLen 保证不越界
        // str.charAt(left) == str.charAt(right) 表示可以扩散 1 次
        while (left >= 0 && right < sLen && str.charAt(left) == str.charAt(right)) {
            p[i]++;
            left--;
            right++;
        }

        // 根据 maxRight 的定义，它是遍历过的 i 的 i + p[i] 的最大者
        // 如果 maxRight 的值越大，进入上面 i < maxRight 的判断的可能性就越大，这样就可以重复利用之前判断过的回文信息了
        if (i + p[i] > maxRight) {
            // maxRight 和 center 需要同时更新
            maxRight = i + p[i];
            center = i;
        }
        if (p[i] > maxLen) {
            // 记录最长回文子串的长度和相应它在原始字符串中的起点
            maxLen = p[i];
            start = (i - maxLen) / 2;
        }
    }
    return s.substring(start, start + maxLen);
}
```

③动态规划：

```
string longestPalindrome(string s)
{
    if (s.empty()) return "";
    int len = s.size();
    if (len == 1)return s;
    int longest = 1;
    int start=0;
    vector<vector<int> > dp(len,vector<int>(len));
    for (int i = 0; i < len; i++)
    {
        dp[i][i] = 1;
        if(i<len-1)
        {
            if (s[i] == s[i + 1])
            {
                dp[i][i + 1] = 1;
                start=i;
                longest=2;
            }
        }
    }
    for (int l = 3; l <= len; l++)//子串长度
    {
        for (int i = 0; i+l-1 < len; i++)//枚举子串的起始点
        {
            int j=l+i-1;//终点
            if (s[i] == s[j] && dp[i+1][j-1]==1)
            {
                dp[i][j] = 1;
                start=i;
                longest = l;
            }
        }
    }
    return s.substr(start,longest);
}
```

$$dp[i,j]=\begin{cases} dp[i-1,j+1], & \text{if } str[i]=str[j] \\ 0, & \text{if } str[i]\neq str[j] \end{cases}$$